T2R2 東京科学大学 リサーチリポジトリ
Science Tokyo Research Repository

論文 / 著書情報
Article / Book Information

題目(和文)	
Title(English)	Construction and Analysis of Post-Quantum Key Exchange Protocols for Secure Messaging
著者(和文)	橋本啓太郎
Author(English)	Keitarou Hashimoto
出典(和文)	学位:博士(工学), 学位授与機関:東京工業大学, 報告番号:甲第12390号, 授与年月日:2023年3月26日, 学位の種別:課程博士, 審査員:尾形 わかは,植松 友彦,山田 功,松本 隆太郎,田中 圭介,安永 憲司
Citation(English)	Degree:Doctor (Engineering), Conferring organization: Tokyo Institute of Technology, Report number:甲第12390号, Conferred date:2023/3/26, Degree Type:Course doctor, Examiner:,,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

Powered by T2R2 (Science Tokyo Research Repository)

TOKYO INSTITUTE OF TECHNOLOGY

DOCTORAL DISSERTATION

Construction and Analysis of Post-Quantum Key Exchange Protocols for Secure Messaging

---

Author:
Keitaro Hashimoto

Supervisor:
Professor Wakaha Ogata

Information and Communications Engineering

Tokyo Institute of Technology

March 2023

Abstract

Messaging applications such as WhatsApp, LINE, Slack, are widely used for both private and business communications. On the other hand, some service providers of such applications and governments are trying to collect information about application users (e.g., messages and social graphs). Such activities threaten users' privacy; thus people have become hesitant to use such messaging applications that do not protect their privacy. To ensure the privacy of users, some messaging apps implement secure group messaging (SGM) and SGM becomes popular around the world. SGM ensures the confidentiality of messages by encrypting messages with the shared secret key among conversation partners. That is, third parties other than conversation partners cannot access messages. In addition, SGM has strong post-compromise and forward secrecy (PCFS) security, which guarantees the confidentiality of past and future messages even if the key at a certain period is compromised. The most famous SGM protocol is Signal, which has been implemented in other SGM apps such as WhatsApp and Facebook Messenger and is currently used by over 2 billion people. However, since its efficiency deteriorates as the number of group members increases, it limits the number of group members to 1,000. To solve this problem, industries such as Google and Meta and academia such as universities and research institutes are collaborating to develop and standardize Message Layer Security (MLS) protocol. It can operate efficiently even in large groups of 50,000 members. In addition, from the other perspective, it is necessary to develop a new SGM secure against quantum computers since currently used cryptographic protocols are known to be broken by large-scale quantum computers. Thus, the National Institute of Standards and Technology (NIST) is working on the standardization of post-quantum cryptography. To ensure the privacy of conversations in the future, we need to start developing post-quantum secure group messaging protocols.

The objective of this work is to realize post-quantum secure group messaging. The contributions of this work are the following.

1. The first contribution of this work is developing a post-quantum authenticated key exchange protocol for Signal's initial key agreement. We formalize the security model for Signal's initial key agreement and propose a new generic construction based on key encapsulation mechanisms and signature schemes. That is, the proposed protocol can be instantiated from various well-studied post-quantum assumptions. Also, we implement the proposed protocol with the NIST PQC candidates and evaluate the communication and computation costs of each instantiation. This experimental result confirms that the proposed protocol works efficiently in a real-world environment. Moreover, we construct a deniable authenticated key exchange protocol from ring signatures and non-interactive zero-knowledge arguments. This allows users to deny the fact that they have exchanged session keys with another user.

2. The second contribution of this work is designing a new post-quantum continuous group key agreement protocol from multi-recipient public key encryptions. The proposed protocol achieves the most efficient total communication costs for each user when all group members update their key materials. We also formulate a new Universal Composability (UC) security model suitable for the proposed protocol and prove the security of the proposed protocol. In addition, we propose a new

lattice-based multi-recipient public key encryption that contributes to reducing the uploading costs of the proposed protocol. Moreover, the experimental result confirms that the proposed protocol works efficiently even for a group of 1000 members.

3. The third contribution of this work is proposing a metadata-hiding continuous group key agreement protocol. To do so, we formulate a UC security model for metadata-hiding secure group messaging, and a simple and generic wrapper protocol that converts any non-metadata-hiding continuous group key agreement into metadata-hiding one with minimum overhead. Then, we rigorously prove that the modified version of the protocol proposed in the second contribution plus our new wrapper protocol satisfies the desired metadata-hiding properties. This is the first provably-secure metadata-hiding SGM protocol. In contrast to existing secure group messaging that only ensures the confidentiality of messages, our protocol additionally hides the relationship between users. Thus, our protocol enhances users' privacy.

Acknowledgments

First and foremost, I would like to express my deepest and most sincere gratitude to my adviser, Professor Wakaha Ogata. I am delighted to have the opportunity to study cryptography under her guidance because, when I was a high school student, I hoped to study cryptography with her. Throughout the six years in the laboratory, she gave me a lot of helpful advice not only on my studies but also on my personal problems. She spent a lot of time discussing my research (and various other miscellaneous topics) and gave me useful clues when I was stuck in my research.

Shuichi Katsumata is my first co-author during my Ph.D. and was an excellent mentor to me. He has led me to the world of secure messaging. I am very grateful for his assistance during my Ph.D. My research could not have been accomplished without discussions with him. He persisted in discussions with me and helped me grow as a researcher: I learned how to find new research topics, how to write academic papers (especially how to write an introduction), and how to give presentations. He also introduced me to Thomas Prest.

Thomas Prest is one of the first foreign researchers I worked with. He noticed the importance of cryptography in the real-world, and allowed me to meet a variety of researchers working on real-world cryptography, and I gained more extensive knowledge about real-world cryptography. In addition, he offered to visit PQShield SAS in Paris. During the visit, I had a more in-depth discussion due to his multifaceted perspectives on things different from mine. I am sincerely grateful to him for giving me such a wonderful experience. Also, he provided great diagrams made by Tikz. The diagrams he created are all helpful in understanding this work.

I would also like to thank my co-authors during my Ph.D., Kris Kwiatkowski, Eamonn W. Postlethwaite, and Bas Westerbaan, for the great collaborations in the area of secure messaging. Kris Kwiatkowski and Bas Westerbaan contributed to implementing the proposed protocols and evaluating their concrete efficiency. Their works made the paper useful not only to cryptographers but also to engineers who are interested in secure messaging. Eamonn W. Postlethwaite is the profession of lattice-based cryptography. Thanks to him, I learned how to choose the parameters of lattice-based primitives.

I also appreciate members of Ogata Laboratory. Especially, I would like to thank Toi Tomita. I enjoyed discussions about cryptography and computer science with him. He gave me a lot of insights to find new research topics. He also invited me to Shin-Akarui-Angou-Benkyou-Kai, which is a cryptography study group led by researchers at the National Institute of Advanced Industrial Science and Technology (AIST). Thanks to this invitation, I obtained the opportunity to work as a research assistant at AIST.

During my Ph.D., I had the pleasure of doing research at AIST. Goichiro Hanaoka and Takahiro Matsuda have always been extremely generous and sincere to me, and the research environment they prepared for me at AIST has had a great positive influence on my research. I wish to thank all members of Shin-Akarui-Angou-Benkyou-Kai. I had a lot of experience in the discussion with them.

I was partially supported by JSPS KAKENHI Grant Number 22J13963. Thanks to their financial support, I was able to visit PQSheild SAS in Paris for a month.

Finally, I owe my deepest gratitude to my parents, Masao and Izumi, and my younger brother Masafumi. I am glad that my parents supported me wholeheartedly in my choice to pursue Ph.D. and become a researcher.

To Keiji Gonda and Keiji Hashimoto, my grandfathers.
To Keitaro Hashimoto, my great-great-grandfather.

Contents

Abstract **iii**

Acknowledgments **v**

1 Introduction **1**
1.1 Background . . . 1
1.2 History of Secure Messaging . . . 2
1.3 Quantum Computers and Their Impact on Cryptographic Protocols . . . 4
1.4 Contributions of Dissertation . . . 5
1.5 Organization of Dissertation . . . 7

2 Preliminaries **9**
2.1 Notations . . . 9
2.2 Key Encapsulation Mechanisms . . . 10
2.3 Secret Key Encryption . . . 12
2.4 Digital Signatures . . . 14
2.5 Pseudo-Random Functions and Pseudo-Random Permutations . . . 15
2.6 Strong Randomness Extractors . . . 15
2.7 Ring Signatures . . . 15
2.8 Non-Interactive Zero-Knowledge . . . 17
2.9 Decomposable Multi-Recipient PKE . . . 17
2.10 Message Authentication Codes . . . 19
2.11 Key Derivation Functions . . . 20

3 Post-Quantum Authenticated Key Exchange for Signal Protocol **21**
3.1 Introduction . . . 21
3.2 Security Model for Signal-Conforming AKE Protocols . . . 27
3.3 Generic Construction of Signal-Conforming AKE . . . 33
3.4 Post-Quantum Signal Handshake . . . 47
3.5 Instantiating Post-Quantum Signal Handshake . . . 50
3.6 Adding Deniability to Basic Signal-Conforming AKE . . . 55
3.7 Equivalence Between Designated Verifier Signature and Ring Signature . . . 73

4 Continuous Group Key Agreement via Post-Quantum Multi-Recipient PKEs **79**
4.1 Introduction . . . 79
4.2 Committing Multi-Recipient PKE . . . 83
4.3 Continuous Group Key Agreement . . . 93
4.4 Proposed Protocol: Chained CmPKE . . . 107

4.5 More Efficient Lattice-Based mPKEs . . . 171
4.6 Instantiation and Implementation of Chained CmPKE . . . 181
4.7 A Variant of GSD Security Tailored to Chained CmPKE . . . 186

5 MetaData-Hiding Continuous Group Key Agreement 193
5.1 Introduction . . . 193
5.2 Background about CGKA . . . 197
5.3 Static Metadata-Hiding CGKA: Define UC Security Model . . . 200
5.4 Static Metadata-Hiding CGKA: Construction and Security Proof . . . 221
5.5 Metadata-Hiding CGKA: Construct Wrapper Protocol W^{mh} . . . 257
5.6 Metadata-Hiding CGKA: Define UC Security Model . . . 271
5.7 Instantiation and Efficiency of Proposed Metadata-Hiding CGKA . . . 297
5.8 Limitation of Efficient Metadata-Hiding CGKA . . . 299

6 Conclusion 303

Bibliography 305

Author's Publications 325

Chapter 1

Introduction

1.1 Background

Online communications through messaging applications, such as WhatsApp, Facebook Messenger, LINE, Slack, etc., have become popular in both private and business situations. Such messaging apps realize *asynchronous* communication through the service provider's server. Consider the case that Alice and Bob want to communicate with each other. When Alice sends a message to Bob, Alice uploads the message to the server and the server stores it until Bob becomes online. When becoming online, he downloads the message and may upload responses. The responses are stored on the server until Alice becomes online. Thus, by communicating through the server, users can send and receive messages regardless of the other user's connection status.

While these messaging apps are convenient, it has become clear that users' privacy is being violated by service providers. Journarists of the Gurdian [GM13] and that of INSIDER [MG19] disclosed that service providers put on collected users' information including communicated messages to governments or other companies. Especially, governments are actively gathering personal information: they use surveillance systems e.g., PRISM [Wik22]. This invasion of privacy by service providers has led users to seek more secure communication methods.

Secure messaging (SM) ensures privacy and security by making sure that only the person you are sending the message to can read the message, a.k.a. end-to-end encryption. With the ever-growing awareness of mass surveillance of communications, people have become more privacy-aware and the demand for SM has been steadily increasing. The baseline security notion guaranteed by SM is that no adversary, including the server, should be able to read the messages sent among the users involved in the conversation. This security is achieved cryptographically. Roughly, exchanged messages are encrypted with an encryption key shared with conversation partners: When Alice and Bob start a conversation, they run a key exchange protocol to share a session key. Then, when they send messages, they encrypt the messages with the shared session key. Since the session key is known by only Alice and Bob, other entities, e.g., service providers and governments, cannot read the encrypted messages. Thus, SM ensures the confidentiality of messages.

SM should have stronger security properties, forward secrecy and post-compromise security, that strengthen the confidentiality of messages by updating session keys continuously. Forward secrecy (FS) guarantees the confidentiality of past messages even if the current session key is compromised; Post-compromise security (PCS) guarantees the confidentiality of future messages even if the current session key is compromised. In other words, the adversary is limited in accessible messages even if it obtains a session key. Therefore, SM minimizes messages leaked by key compromise.

All SM ensures the confidentiality of messages, but it is not sufficient to protect users' privacy from the server. This is because the server may be able to collect *metadata*, such as the identities of the sender and

of other group members, which can both be leaked from the exchanged encrypted contents. It has been shown in numerous real-world scenarios [Sav13; Fli15; Ops13; Moh21; Car15] that knowledge of metadata alone can cause damaging repercussions, sometimes enough to defeat the purpose of using SM. These also have negative impacts on the activity and safety of some users, e.g., journalists and activists [McG+15; MRC16]. Recent media articles [Bre22; Kro21] report that, in the United States, metadata collection by law enforcement agencies on *SM applications* is a widespread practice, supported by a legal [Kro21] and technical framework [Bre22] that gives it a wide reach.

1.2 History of Secure Messaging

This section summarizes the history and related works of secure messaging.

1.2.1 Secure Two-Party Messaging

Secure two-party messaging is the one specialized for two-party conversations. While there has been a range of secure two-party messaging protocols, the Signal protocol [Sig] is widely regarded as the gold standard. Not only is it used by the Signal app[1], the Signal protocol is also used by WhatsApp, Skype, and Facebook Messenger among many others, where the number of active users is well over 2 billion. One of the reasons for such popularity is due to the simplicity and the strong security properties it provides, while simultaneously allowing for the same user experience as any (non-cryptographically secure) messaging apps.

Signal protocol consists of two sub-protocols: the X3DH protocol [MP16b][2] and the double ratchet protocol [MP16a]. The former protocol can be viewed as a type of key exchange protocol allowing two parties to exchange a secure initial session key. The latter protocol is executed after the X3DH protocol, and it allows two parties to perform a secure back-and-forth message delivery. Each time a message is sent, the double ratchet protocol internally runs a key exchange to establish a new shared key for encrypting the next message.

The Double Ratchet Protocol. The first attempt at a full security analysis of the Signal protocol was made by Cohn-Gordon et al. [Coh+17; Coh+20]. They considered the Signal protocol as one large protocol and analyzed the security guarantees in its entirety. Since the double ratchet protocol was understood to be the root of the complexity, many subsequent works aimed at further abstracting and formalizing (and enhancing in some cases) the security of the double ratchet protocol by viewing it as a stand-alone protocol. Bellare et al. [Bel+17] partially abstracted the double ratchet protocol as "ratcheted key exchange" (RKE), and the subsequent works updated the model of RKE to capture a lot of properties [PR18; DV19; JMM19b; JMM19a; BRV20; CDV21; Dow+22]. Notably, Alwen et al. [ACD19] fully abstracted the complex Diffie-Hellman-based double ratchet protocol used by Signal as "continuous key agreement" (CKA) and provided a concrete security model along with a generic construction based on simple building blocks. Since these blocks are instantiable from versatile assumptions, including post-quantum ones, their work resulted in the first *post-quantum secure* double ratchet protocol. Recently, Canettie et al. [Can+22] and Biestock et al. [Bie+22b] formalized the Universally Composable (UC) security model for the Signal protocol. Thanks to these works, the double ratchet protocol has been understood well. Here, we elucidate that all the aforementioned works analyze the double ratchet protocol as a stand-alone primitive, and hence, it is

[1] The name Signal is used to point to the app *and* the protocol.

[2] X3DH stands for Extended Triple Diffie-Hellman.

assumed that any two parties can securely share an initial session key, for instance, by executing a "secure" X3DH protocol.

The X3DH Protocol. In contrast, works focusing on the X3DH protocol seem to be limited. Cohn-Gordon et al. [Coh+17; Coh+20] indirectly considered the X3DH protocol, but they did not provide security models of it. Brendel et al. [Bre+20] abstract the X3DH protocol and provide the first generic construction based on a new primitive they call a *split key encapsulation mechanism* (split KEM). However, so far, instantiations of split KEMs with strong security guarantees required for the X3DH protocol are limited to Diffie-Hellman style assumptions. In fact, the recent result of Guo et al. [Guo+20] implies that it would be difficult to construct them from one of the promising post-quantum candidates: lattice-based assumptions (and presumably coded-based assumptions). On the other hand, Vatandas et al. [Vat+20] studied one of the security guarantees widely assumed for the X3DH protocol called (off-line) deniability [MP16b, Section 4.4] and showed that a strong knowledge-type assumption would be necessary to formally prove it. Unger and Goldberg [UG15; UG18] construct several protocols that can be used as drop-in replacements of the X3DH protocol that achieve a strong flavor of (online) *deniability* from standard assumptions, albeit by making a noticeable sacrifice in the security against key-compromise attacks: a type of attack that exploits leaked secret information of a party. For instance, while the X3DH protocol is secure against key-compromise impersonation (KCI) attacks [BJM97],[3] the protocols of Unger and Goldberg are no longer secure against such attacks.[4]

Metadata Protection for Two-Party SM. To hide metadata, Signal app implements sender-anonymouse SM called *Sealed Sender* [Sig18]. Using it, message senders can hide their information from the server. Martiny et al. [Mar+21] analyzed its security and proposed an improved version of Sealed Sender which uses blind signatures [Cha82]. Tyagi et al. [Tya+21] pointed out the drawback of Signal's Sealed Sender and proposed a new sender-anonymous SM based on group signatures to overcome the drawback.

1.2.2 Secure Group Messaging

Secure group messaging (SGM) realizes secure conversations among two or more users, i.e., groups. Signal realizes group messaging with *pairwise channels*: each member in a group establishes two-party SM channels with all other group members, and a group member sends messages with each channel. This protocol is simple, but the sender requires $O(N)$ computation and communication costs where N is the number of group members. Since messages are sometimes large (e.g., the size of a high-resolution picture is several Megabytes), this protocol does not scale well due to the large messaging costs. In fact, Signal limits the maximum number of group members to one thousand.

WhatsApp uses another SGM protocol called *Sender Key*: Sender Key also uses pairwise channels, but instead of sending messages via pairwise channels, the seder sneds the encryption key via the channels and broadcasts messages encrypted with the shared encryption key. Although Sender Key requires $O(N)$ costs for sharing encryption keys, it achieves $O(1)$ costs for sending messages. Since the size of encryption keys is much smaller than that of messages, Sender Key reduces the cost of sending messages. However, the cost of key updates still remains $O(N)$. The cost for key updates should be small since they are important to achieve strong security (FS and PCS).

To reduce the costs of key updates, *Asynchronous Ratcheting Trees* (ART) was developed by Cohn-Gordon et al. [Coh+18]. ART is a group key agreement protocol to share a group session key, which is designed to

[3] Although [MP16b, Section 4.6] states that the X3DH protocol is susceptible to KCI attacks, this is only because they consider the scenario where the *session-specific* secret is compromised. If we consider the standard KCI attack scenario where the long-term secret is the only information being compromised [BJM97], then the X3DH protocol is secure.

[4] Being vulnerable against KCI attacks seems to be intrinsic to online deniability [UG15; UG18; MP16b].

be able to update session keys with $O(\log N)$ costs (in the best case; in the worst case, the costs become $O(N)$). ART is based on the specific Diffie-Hellman problem.

Based on the ideas of ART [Coh+18], TreeKEM [BBR18] was constructed. TreeKEM can be seen as a generalized version of ART based on standard building blocks. It enjoys good efficiency like ART and, thanks to its generality, can be instantiable from post-quantum assumptions. TreeKEM is used as the core protocol in Message Layer Security (MLS), which is standardizing in Internet Engineering Task Force (IETF) working group (WG)[5].

To date, TreeKEM discussed in MLS WG has gone through 16 versions, some of which have undergone formal security analysis. TreeKEM version 7 was analyzed by Alwen et al. [Alw+20a] and Bhargavan et al. [BBN19a]. The former abstracts TreeKEM as "continuous group key agreement" (CGKA) (i.e., group version of CKA) and proved its security based on a game-based security model for CGKA. The latter presented mechanized security proof. TreeKEM version 10, which adopts the 'parent hash' and 'tree-signing' mechanisms to enhance security, was analyzed by Alwen et al. [AJM22]. They proved that it is secure against active and insider adversaries with a newly-defined universally-composable security model. The key scheduling in MLS version 11 was analyzed by Brzuska et al. [BCK22] via the State Separating Proofs methotology [Brz+18] proposed by Brzuska et al. In contrast to TreeKEM, the MLS protocol including message encryption is not well analyzed. Alwen et al. [Alw+21a] proved the security of the non-metadata-hiding version of MLS (called MLSPlaintext) in version 11.

In addition to the standard TreeKEM discussed in MLS WG, variants of TreeKEM have been proposed. Re-randomized TreeKEM [Alw+20a] and TreeKEM with active security [Alw+20b] improve the forward security property against passive and active adversaries, respectively, but require relatively heavy cryptographic primitives. *Tainted* TreeKEM [Kle+21] enjoys efficiency advantages for large groups maintained by a small number of 'administrators.' *Causal* TreeKEM [Wei19] is an initial attempt to support concurrent changes to the group states but it depends on Diffie-Hellman problems and currently has no accompanying formal security proof. Alwen et al. proposed CoCoA [Alw+22c] and DeCAF [Alw+22a] supporting concurrent changes to the group states based on generic primitives. They allow updating keys with $O(\log N)$ costs in all cases. Weidner et al. [Wei+21] proposed *decentralized CGKA*, which does not require a centralized server responsible for message delivery, and its security proof. Theoretical analysis for CGKA has also been conducted. Bienstock et al. investigated the trade-offs and limits on the efficiency of CGKA [BDR20; Bie+22a].

Metadata Protection for SGM. Signal app considers metadata protection in group messaging. It implements metadata-hiding SM called Private Groups [Sig19]. It allows group members to exchange messages anonymously against the server. Its security was analyzed by Chase et al. [CPZ20]. MLS WG is discussing a metadata-hiding version called MLSCiphertext, but its security has not been analyzed in contrast MLSPlaintext was analyzed.

1.3 Quantum Computers and Their Impact on Cryptographic Protocols

The existing key exchange protocols deployed in widely used SM apps are based on the hardness of factoring-related or discrete logarithm-related problems (e.g., RSA or Diffie-Hellman problem). However, quantum computers can solve these problems in polynomial time [Sho94]. In other words, SM based on such problems is no longer secure if large-scale quantum computers will be developed. To resist the threat of quantum computers, post-quantum cryptographic primitives are being prepared. For example, the U.S. National Institute of Standards and Technology (NIST) is working on the standardization of

[5] https://datatracker.ietf.org/wg/mls/about/

post-quantum cryptography(PQC)[6]. However, it is known that post-quantum primitives are less efficient than pre-quantum primitives based on RSA or Diffie-Hellman problems. For example, the size of the ciphertext/signature of the NIST selected PQC algorithms Kyber PKE [Sch+22]/Falcon signature [Pre+22] is about 1 Kilobyte. This is about 15 times larger than the Diffie-Hellman-based ElGamal PKE [ElG85]/Schnorr signature [Sch90], which sizes 64 bytes. To overcome such efficiency-related issues, researchers have proposed new versions of real-world cryptographic protocols suitable to the post-quantum era. The examples of such considerations are KEMTLS [SSW20], McTiny [BL20] and Post-Quantum WireGuard [Hül+21].

1.4 Contributions of Dissertation

This dissertation proposes practical post-quantum authenticated (group) key exchange protocols, which are important to construct practical and post-quantum secure (group) messaging. Below, we provide a brief summary of the contributions.

1.4.1 Post-Quantum Authenticated Key Exchange for Signal Protocol

Although we have a rough understanding of what the X3DH protocol offers [MP16b; Coh+17; Coh+20], the current state of affairs is unsatisfactory for the following reasons, and making progress on these issues will be the focus of this work:

- It is difficult to formally understand the security guarantees offered by the X3DH protocol or to make a meaningful comparison among different protocols achieving the same functionality as the X3DH protocol without a clearly defined security model.
- The X3DH protocol is so far only instantiable from Diffie-Hellman style assumptions [Bre+20] and it is unclear whether such assumptions are inherent to the Signal protocol.
- Ideally, similarly to what Alwen et al. [ACD19] did for the double ratchet protocol, we would like to abstract the X3DH protocol and have a generic construction based on simple building blocks that can be instantiated from versatile assumptions, including but not limited to post-quantum ones. However, no such model or generic construction is known.
- No matter how secure the double ratchet protocol is, we cannot completely secure the Signal protocol if the initial X3DH protocol is the weakest link in the chain (e.g., insecure against state leakage and only offering security against classical adversaries).

To construct post-quantum replacements of the X3DH protocol, we cast it as a specific type of authenticated key exchange (AKE) protocol, which we call a *Signal-conforming AKE* protocol, and formally define its security model based on the vast prior works on AKE protocols. We then provide the first efficient generic construction of a Signal-conforming AKE protocol based on standard cryptographic primitives such as key encapsulation mechanisms (KEM) and signature schemes. Specifically, this results in the first post-quantum secure replacement of the X3DH protocol based on well-established assumptions. Similar to the X3DH protocol, the proposed Signal-conforming AKE protocol offers a strong (or stronger) flavor of security, where the exchanged key remains secure even when all the non-trivial combinations of the long-term secrets and session-specific secrets are compromised. Moreover, we further show how to progressively strengthen

[6] `https://csrc.nist.gov/projects/post-quantum-cryptography`

its deniability using ring signatures and/or non-interactive zero-knowledge proof systems. Finally, we provide a full-fledged, generic C implementation of the basic (weakly deniable) protocol. We instantiate it with several Round 3 candidates (finalists and alternates) to the NIST post-quantum standardization process and compare the resulting bandwidth and computation performances. The implementation of the basic protocol is publicly available and thus anyone can test its efficiency.

1.4.2 Continuous Group Key Agreement via Post-Quantum Multi-Recipient Public Key Encryption

Continuous group key agreements (CGKAs) [Alw+20a; Alw+20b] are a class of group key exchange protocols that provide functionalities and strong security guarantees required for secure group messaging protocols such as Signal and MLS. Protection against device compromise is provided by *commit messages*: at a regular rate, each group member may refresh their key material by uploading a commit message, which is then downloaded and processed by all the other group members. In practice, propagating commit messages dominates the bandwidth consumption of existing CGKAs.

We propose Chained CmPKE, a CGKA with an asymmetric bandwidth cost: in a group of N members, a commit message costs $O(N)$ to upload and $O(1)$ to download, for a total bandwidth cost of $O(N)$. In contrast, TreeKEM [BBR18; Oma+21; Bar+20] costs $\Omega(\log N)$ in both directions, for a total cost $\Omega(N \log N)$. The proposed protocol relies on generic primitives and is therefore readily post-quantum.

We go one step further and propose post-quantum primitives that are tailored to Chained CmPKE, which allows us to cut the growth rate of *uploaded* commit messages by two or three orders of magnitude compared to naive instantiations. Finally, we realize a software implementation of Chained CmPKE. The experiments show that even for groups with a size as large as $N = 2^{10}$, commit messages can be computed and processed in less than 100 ms.

1.4.3 MetaData-Hiding Continuous Group Key Agreement

Although robust techniques have been developed to protect the *contents* of conversations in this context, it is in general more challenging to protect *metadata* (e.g. the identity and social relationships of group members), since their knowledge is often needed by the server to ensure the proper function of the SGM protocol.

We provide a simple and generic wrapper protocol that upgrades non-metadata-hiding CGKAs into metadata-hiding CGKAs. The key insight is to leverage the existence of a unique continuously evolving group secret key shared among the group members. We use this key to perform a group membership authentication protocol that convinces the server in an *anonymous* manner that a user is a legitimate group member. Our technique only uses a standard signature scheme, and thus, the wrapper protocol can be instantiated from a wide range of assumptions, including post-quantum ones. It is also very efficient, as it increases the bandwidth cost of the underlying CGKA operations by at most a factor of two.

To formally prove the security of the proposed protocol, we use the universal composability (UC) framework [Can01] and model a new ideal functionality $\mathcal{F}_{\text{CGKA}}^{\text{mh}}$ capturing the correctness and security guarantee of metadata-hiding CGKA based on the state-of-the-art UC security model on secure messaging [AJM22; Has+21b]. To capture the above intuition of a "wrapper" protocol, we also define a restricted ideal functionality $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$, which roughly captures a non-metadata-hiding CGKA and, as a concrete example, proves that the modified version of Chained CmPKE [Has+21b] called Chained CmPKE$^{\text{ctxt}}$ realizes $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$. We then show that our wrapper protocol UC-realizes $\mathcal{F}_{\text{CGKA}}^{\text{mh}}$ in the $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$-hybrid model, which in

particular formalizes the intuition that any non-metadata-hiding CGKA can be modularly converted into metadata-hiding CGKA.

1.5 Organization of Dissertation

This dissertation is organized as follows. Chapter 2 introduces basic notations and definitions of cryptographic primitives used in this dissertation. Chapter 3 proposes a new post-quantum authenticated key exchange protocol that can be used as a drop-in replacement of the Signal's initial key agreement protocol. Chapter 4 proposes a continuous group key agreement protocol based on multi-recipient public key encryption and more efficient lattice-based multi-recipient public key encryption. Chapter 5 considers metadata protection on secure group messaging and proposes security models and generic constructions of metadata-hiding continuous group key agreement that can be instantiable from various post-quantum assumptions. Concluding remarks are given in Chapter 6.

Chapter 2

Preliminaries

In this chapter, we prepare notations and cryptographic primitives that will be used throughout the dissertation.

2.1 Notations

We denote the set of natural numbers (non-negative integers) by $\mathbb{N}$ and the security parameter by $\kappa \in \mathbb{N}$. The set of κ-bit strings is denoted by $\{0,1\}^\kappa$. The κ-bit string consisting of only $a \in \{0,1\}$ is denoted by a^κ.

The operator $\oplus$ denotes bit-wise exclusive-or ("XOR"), and $\|$ denotes string concatenation. We denote by $[\![cond]\!]$ the bit that is 1 if the boolean statement *cond* is true, and 0 otherwise.

For $n \in \mathbb{N}$, we write $[n]$ to denote the set $[n] := \{ 1, \ldots, n \}$. For $j \in [n]$, we write $[n \backslash j]$ to denote the set $[n \backslash j] := \{ 1, \ldots, n \} \setminus \{ j \}$. For $a, b \in \mathbb{N}$ ($a < b$), we write $[a : b]$ to denote the set $[a : b] := \{ a, \ldots, b \}$.

We use $v \leftarrow x$ and $v := x$ to denote assigning the value x to the variable v, and use $v \leftarrow\!\!\$\, S$ to denote sampling an element v uniformly and randomly from a finite set S.

Let $\mathcal{A}$ be an algorithm. By $y \leftarrow \mathcal{A}(x)$, we denote that y is obtained by running $\mathcal{A}$ on input x. With $y \in \mathcal{A}(x)$, we denote a possible output y of the execution of $\mathcal{A}$ on input x. When we want to make the randomness explicit, we use the notation $y \leftarrow \mathcal{A}(x; \mathsf{r})$ meaning that the randomized algorithm $\mathcal{A}$ is executed with the explicit randomness r. We use the notation $\mathcal{A}^{\mathcal{O}}$, where $\mathcal{O} = \{ O_1(\cdot), \ldots, O_w(\cdot) \}$, to denote the algorithm $\mathcal{A}$ has access to oracle machines $O_1, \ldots, O_w$ which will answer queries to it.

Let ϕ be a boolean function. We write

$$\Pr\left[y_1 \leftarrow \mathcal{A}_1(x_1), \ldots, y_t \leftarrow \mathcal{A}_t(x_t) : \phi(y_1, \ldots, y_t) \right]$$

to denote the probability of the event that $\phi(y_1, \ldots, y_t)$ is `true` after the value $y_1, \ldots, y_t$ was obtained by (orderly) running algorithm $\mathcal{A}_1, \ldots, \mathcal{A}_t$ on inputs $x_1, \ldots, x_t$, respectively. See, for example, Definition 2.3.5.

We denote polynomial functions with respect to κ by $\mathsf{poly}(\kappa)$. A non-negative function $f(\kappa)$ is said to be negligible with respect to κ if for any positive polynomial $\mathsf{poly}(\kappa)$ and for all sufficiently large κ, we have $f(\kappa) < \frac{1}{\mathsf{poly}(\kappa)}$. We write $\mathsf{negl}(\kappa)$ to denote negligible functions with respect to κ. PPT (resp. QPT) stands for probabilistic (resp. quantum) polynomial time. We say that the algorithm $\mathcal{A}$ is *efficient* if its running time is polynomial in its input length. For example, the running time of $\mathcal{A}(1^\kappa)$ is $\mathsf{poly}(|1^\kappa|) = \mathsf{poly}(\kappa)$. By definition, both PPT and QPT algorithms are efficient.

Data structure. For a set V, we write $V \xleftarrow{+} x$ and $V \xleftarrow{-} x$ as shorthand for $V \leftarrow V \cup \{ x \}$ and $V \leftarrow V \setminus \{ x \}$, respectively. For another set W, we write $V \xleftarrow{+} W$ and $V \xleftarrow{-} W$ as shorthand for $V \leftarrow V \cup W$ and $V \leftarrow V \setminus W$, respectively. For lists (vectors) $x := (x_1, \ldots, x_n)$ and $y := (y_1, \ldots, y_m)$, we denote the concatenation of x and y by $x \| y = (x_1, \ldots, x_n, y_1, \ldots, y_m)$ and use $x \xleftarrow{\shortparallel} v$ as a shorthand for $x \leftarrow x \| (v)$ for a value v. We further use associative arrays and use $A[i] \leftarrow x$ and $y \leftarrow A[i]$ to assign and

retrieval an element i, respectively. We denote by $A[*] \leftarrow v$ the initialization of the array to the default value v. For simplicity, we use the wildcard notation when dealing with sets of tuples and multi-argument associative arrays. For example, for an array with domain $\mathcal{I} \times \mathcal{J}$, we write $A[*, j] := \{ A[i, j] \mid i \in \mathcal{I} \}$ and for a set $S \subseteq \mathcal{I} \times \mathcal{J}$, we write $(i, *) \in S$ as a shorthand for the condition $\exists j \in \mathcal{J} : (i, j) \in S$.

Keywords. We use the following keywords in pseudo-codes:

- **req** *cond* denotes that if the condition *cond* is false, then the current function unwinds all state changes and immediately returns $\perp$.
- **parse** $(m_1, \ldots, m_n) \leftarrow m$ denotes an attempt to parse a message m as a tuple. If m is not of the correct format, the current function unwinds all state changes and immediately returns $\perp$.
- **try** $y \leftarrow *\texttt{func}(x)$ is a shorthand notation for calling a helper function $*\texttt{func}$ and executing **req** $y \neq \perp$.
- **assert** *cond* is only used to describe ideal functionalities. It denotes that if *cond* is false, then the functionality permanently halts, making the real and ideal worlds trivially distinguishable (this is used to validate inputs of the simulator).

2.2 Key Encapsulation Mechanisms

The syntax and the correctness of key encapsulation mechanism (KEM) schemes are defined as follows.

Definition 2.2.1 (KEM Schemes). *A key encapsulation mechanism (KEM) scheme* KEM *with session key space* $\mathcal{KS}$ *consists of the following algorithms:*

- $\mathsf{Setup}(1^\kappa) \to \mathsf{pp}$: *The setup algorithm takes the security parameter* 1^κ *as input and outputs a public parameter* pp*. In the following, we assume* pp *is provided to all the algorithms and may omit it for simplicity.*
- $\mathsf{KeyGen}(\mathsf{pp}) \to (\mathsf{ek}, \mathsf{dk})$: *The key generation algorithm takes a public parameter* pp *as input and outputs a pair of keys* $(\mathsf{ek}, \mathsf{dk})$*.*
- $\mathsf{Encap}(\mathsf{ek}) \to (\mathsf{K}, \mathsf{C})$: *The encapsulation algorithm takes an encapsulation key* ek *as input and outputs a session key* $\mathsf{K} \in \mathcal{KS}$ *and a ciphertext* C*.*
- $\mathsf{Decap}(\mathsf{dk}, \mathsf{C}) \to \mathsf{K} / \perp$: *The decapsulation algorithm takes a decapsulation key* dk *and a ciphertext* C *as input and outputs either a session key* $\mathsf{K} \in \mathcal{KS}$ *or* $\perp \notin \mathcal{KS}$*.*

Definition 2.2.2 ($(1-\delta)$-Correctness for KEM). *We say a KEM scheme* KEM *is* $(1-\delta)$-correct *if for all* $\kappa \in \mathbb{N}$ *and* $\mathsf{pp} \in \mathsf{Setup}(1^\kappa)$*,*

$$(1-\delta) \leq \Pr\left[\begin{array}{c} (\mathsf{ek}, \mathsf{dk}) \leftarrow \mathsf{KeyGen}(\mathsf{pp}), \\ (\mathsf{K}, \mathsf{C}) \leftarrow \mathsf{Encap}(\mathsf{ek}) \end{array} : \mathsf{Decap}(\mathsf{dk}, \mathsf{C}) = \mathsf{K} \right].$$

We define the min-entropy of KEM encapsulation keys and ciphertexts.

Definition 2.2.3 (Min-Entropy of KEM Encapsulation Key). *We say a KEM scheme* KEM *has* ν-high encapsulation key min-entropy *if for all* $\kappa \in \mathbb{N}$ *and* $\mathsf{pp} \in \mathsf{Setup}(1^\kappa)$*,*

$$\nu \leq -\log_2 \left(\max_{\mathsf{ek}^*} \Pr\left[(\mathsf{ek}, \mathsf{dk}) \leftarrow \mathsf{KeyGen}(\mathsf{pp}) : \mathsf{ek} = \mathsf{ek}^* \right] \right).$$

$\mathsf{Game}^{\mathsf{IND\text{-}ATK}}_{\mathsf{KEM},\mathcal{A}}(\kappa)$

1 : $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\kappa)$
2 : $(\mathsf{ek}^*, \mathsf{dk}^*) \leftarrow \mathsf{KeyGen}(\mathsf{pp})$
3 : $\mathsf{state} \leftarrow \mathcal{A}_1^{\mathcal{O}}(\mathsf{pp}, \mathsf{ek}^*)$
4 : $b \leftarrow\$ \{0,1\}$
5 : $(\mathsf{K}_0^*, \mathsf{C}_0^*) \leftarrow \mathsf{Encap}(\mathsf{ek}^*)$
6 : $\mathsf{K}_1^* \leftarrow\$ \mathcal{KS}$
7 : $b' \leftarrow \mathcal{A}_2^{\mathcal{O}}(\mathsf{pp}, \mathsf{ek}^*, (\mathsf{K}_b^*, \mathsf{C}_0^*), \mathsf{state})$
8 : **return** $[\![b = b']\!]$

Available oracles

if $\mathsf{ATK} = \mathsf{CPA} : \mathcal{O} = \emptyset$
if $\mathsf{ATK} = \mathsf{CCA} : \mathcal{O} = \{\, \mathtt{Dec}(\cdot) \,\}$

Decapsulation Oracle $\mathtt{Dec}(\mathsf{C})$

1 : **req** $\mathsf{C} \neq \mathsf{C}_0^*$
2 : $\mathsf{K} \leftarrow \mathsf{Decap}(\mathsf{dk}^*, \mathsf{C})$
3 : **return** K

FIGURE 2.1: Security games for defining the security notions of KEM.

Definition 2.2.4 (Min-Entropy of KEM Ciphertext). *We say a KEM scheme* KEM *has* χ-high ciphertext min-entropy *if for all* $\kappa \in \mathbb{N}$ *and* $\mathsf{pp} \in \mathsf{Setup}(1^\kappa)$,

$$\chi \leq -\log_2 \left(\mathbb{E} \left[\max_{\mathsf{C}^*} \Pr\left[(\mathsf{K}, \mathsf{C}) \leftarrow \mathsf{Encap}(\mathsf{ek}) : \mathsf{C} = \mathsf{C}^* \right] \right] \right),$$

where the expectation is taken over the randomness used to sample $(\mathsf{ek}, \mathsf{dk}) \leftarrow \mathsf{KeyGen}(\mathsf{pp})$.

We recall the standard security notions of KEM schemes: indistinguishability against chosen plaintext attacks (IND-CPA) and indistinguishability against chosen ciphertext attacks (IND-CCA) [Sho00; CS03]. The difference between IND-CPA and IND-CCA is the adversary in IND-CCA game can access the decapsulation oracle.

Definition 2.2.5 (IND-CPA **and** IND-CCA **Security for KEM).** *Let* $\mathsf{ATK} \in \{\, \mathsf{CPA}, \mathsf{CCA} \,\}$. *We define the game* $\mathsf{Game}^{\mathsf{IND\text{-}ATK}}_{\mathsf{KEM},\mathcal{A}}$ *in Figure 2.1 that an adversary* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ *runs in and the advantage of* $\mathcal{A}$ *as*

$$\mathsf{Adv}^{\mathsf{IND\text{-}ATK}}_{\mathsf{KEM},\mathcal{A}}(\kappa) := \left| \Pr\left[\mathsf{Game}^{\mathsf{IND\text{-}ATK}}_{\mathsf{KEM},\mathcal{A}}(\kappa) = 1 \right] - \frac{1}{2} \right|.$$

We say that KEM *is* IND-ATK secure *if* $\mathsf{Adv}^{\mathsf{IND\text{-}ATK}}_{\mathsf{KEM},\mathcal{A}}(\kappa) \leq \mathsf{negl}(\kappa)$ *for any efficient adversaries* $\mathcal{A}$.

We define plaintext-awareness (PA) for KEM schemes [BR95; BP04] where multiple keys are considered [MSs12].[1] We consider strengthening the (already strong) PA security where the efficient extractor $\mathcal{E}_\mathcal{C}$ for the ciphertext creator $\mathcal{C}$ can be constructed efficiently given the description of $\mathcal{C}$. This is required in the proof of deniability as the simulator must construct such $\mathcal{E}_\mathcal{C}$ given the description of the adversary.

Definition 2.2.6 (Plaintext-Awareness for KEM). *Let* $t = \mathsf{poly}(\kappa)$ *be an integer. We say a KEM scheme* KEM *is* plaintext-aware (PA_t-1) secure *if for all* $\kappa \in \mathbb{N}$ *and (non-uniform)* PPT *ciphertext creator* $\mathcal{C}$, *there exists a* PPT *extractor* $\mathcal{E}_\mathcal{C}$ *such that for any efficient distinguisher* $\mathcal{D}$, *the two experiments* $\mathsf{Exp}^{\mathsf{dec}}_{\mathcal{C},\mathcal{D}}$ *and* $\mathsf{Exp}^{\mathsf{ext}}_{\mathcal{C},\mathcal{E}_\mathcal{C},\mathcal{D}}$ *defined in Figure 2.2 are indistinguishable.*

[1] This property is required to prove the deniability of the proposed authenticated key exchange protocol in Section 3.6. We observe that the standard PA security defined for a single key does not immediately imply a multi-key variant and that the original proof of deniability by Di Raimondo et al. [DGK06, Theorem 2 and 3] crucially relies on the multi-key variant.

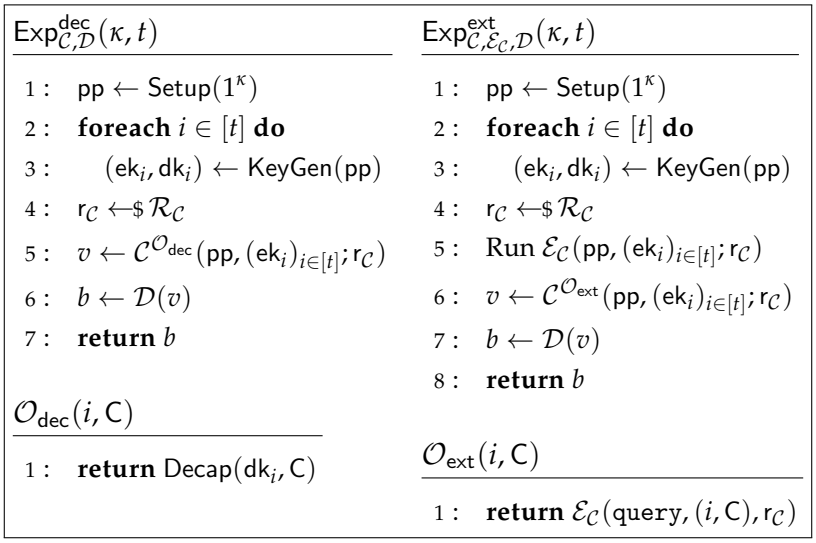


FIGURE 2.2: Security experiments for defining the plaintext-awareness of KEM. $\mathcal{R}_\mathcal{C}$ denotes the randomness space of the KEM scheme. We assume the algorithms $\mathcal{C}$ and $\mathcal{E}_\mathcal{C}$ are stateful.

Moreover, we say the extractor $\mathcal{E}_\mathcal{C}$ is efficiently constructible *if the description of $\mathcal{E}_\mathcal{C}$ can be efficiently computed from the description of $\mathcal{C}$.*

2.3 Secret Key Encryption

The syntax and the correctness of secret key encryption (SKE) schemes are defined as follows.

Definition 2.3.1 (Secret-Key Encryption). *A secret-key encryption (SKE) scheme* SKE *over a key space $\mathcal{K}$ and message space $\mathcal{M}$ consists of the following algorithms:*

- $\mathsf{Enc}_\mathsf{s}(\mathsf{k}, \mathsf{m}) \to \mathsf{c}$: *The encryption algorithm takes a secret key* $\mathsf{k} \in \mathcal{K}$ *and a message* $\mathsf{m} \in \mathcal{M}$ *as input and outputs a ciphertext* c.
- $\mathsf{Dec}_\mathsf{s}(\mathsf{k}, \mathsf{c}) \to \mathsf{m}/\perp$: *The decryption algorithm takesa secret key* k *and a ciphertext* c *as input and (deterministically) outputs either* $\mathsf{m} \in \mathcal{M}$ *or* $\perp \notin \mathcal{M}$.

Definition 2.3.2 (Correctness for SKE). *We say that a* SKE *is* correct *if for all* $\mathsf{m} \in \mathcal{M}$ *and* $\mathsf{k} \in \mathcal{K}$,

$$\Pr\left[\mathsf{Dec}_\mathsf{s}(\mathsf{k}, \mathsf{Enc}_\mathsf{s}(\mathsf{k}, \mathsf{m})) = \mathsf{m}\right] = 1.$$

We recall the IND-CPA and IND-CCA security for SKE by the left-or-right version of game-based indistinguishability [BN00].

Definition 2.3.3 (IND-CPA and IND-CCA security for SKE). *Let* $\mathsf{ATK} \in \{\mathsf{CPA}, \mathsf{CCA}\}$. *We define the game* $\mathsf{Game}^{\mathsf{IND\text{-}ATK}}_{\mathsf{SKE},\mathcal{A}}$ *illustrated in Figure 2.3 and the advantage of $\mathcal{A}$ as*

$$\mathsf{Adv}^{\mathsf{IND\text{-}ATK}}_{\mathsf{SKE},\mathcal{A}}(\kappa) := \left|\Pr\left[\mathsf{Game}^{\mathsf{IND\text{-}ATK}}_{\mathsf{SKE},\mathcal{A}}(\kappa) = 1\right] - \frac{1}{2}\right|.$$

We say that a SKE *is* IND-ATK secure *if* $\mathsf{Adv}^{\mathsf{IND\text{-}ATK}}_{\mathsf{SKE},\mathcal{A}}(\kappa) \leq \mathsf{negl}(\kappa)$ *for any efficient adversaries $\mathcal{A}$.*

$\mathsf{Game}^{\text{OT-IND-CCA}}_{\mathsf{SKE},\mathcal{A}}(\kappa)$

1 : $\mathsf{k} \leftarrow\!\$\, \mathcal{K}$
2 : $b \leftarrow\!\$\, \{0,1\}$
3 : $(\mathsf{m}_0, \mathsf{m}_1, \mathsf{state}) \leftarrow \mathcal{A}(1^\kappa)$
4 : $\mathsf{c}^* \leftarrow \mathsf{Enc}_\mathsf{s}(\mathsf{k}, \mathsf{m}_b)$
5 : $\mathsf{EL} \leftarrow \{\, \mathsf{c}^* \,\}$
6 : $b' \leftarrow \mathcal{A}^{\mathtt{Dec}_\mathsf{s}(\cdot)}(\mathsf{c}^*, \mathsf{state})$
7 : **return** $[\![b = b']\!]$

$\mathsf{Game}^{\text{IND-ATK}}_{\mathsf{SKE},\mathcal{A}}(\kappa)$

1 : $\mathsf{EL} \leftarrow \emptyset$
2 : $\mathsf{k} \leftarrow\!\$\, \mathcal{K}$
3 : $b \leftarrow\!\$\, \{0,1\}$
4 : $b' \leftarrow \mathcal{A}^{\mathcal{O}}(1^\kappa)$
5 : **return** $[\![b = b']\!]$

Available oracles

if $\mathsf{ATK} = \mathsf{CPA} : \mathcal{O} = \{\, \mathtt{LoR}_\mathsf{s}(\cdot,\cdot) \,\}$
if $\mathsf{ATK} = \mathsf{CCA} : \mathcal{O} = \{\, \mathtt{LoR}_\mathsf{s}(\cdot,\cdot), \mathtt{Dec}_\mathsf{s}(\cdot) \,\}$

Left-or-Right Oracle $\mathtt{LoR}_\mathsf{s}(\mathsf{m}_0, \mathsf{m}_1)$

1 : $\mathsf{c} \leftarrow \mathsf{Enc}_\mathsf{s}(\mathsf{k}, \mathsf{m}_b)$
2 : $\mathsf{EL} \mathrel{+}\!\leftarrow \mathsf{c}$
3 : **return** c

Decryption Oracle $\mathtt{Dec}_\mathsf{s}(\mathsf{c})$

1 : **req** $\mathsf{c} \notin \mathsf{EL}$
2 : $\mathsf{m} \leftarrow \mathsf{Dec}_\mathsf{s}(\mathsf{k}, \mathsf{c})$
3 : **return** m

FIGURE 2.3: Security games for defining the security notions of SKE. If the condition following **req** does not hold, the game terminates by returning a random bit.

For convenience, we define the one-time IND-CCA security for SKE separately.

Definition 2.3.4 (One-time IND-CCA security for SKE). *We define the game* $\mathsf{Game}^{\text{OT-IND-CCA}}_{\mathsf{SKE},\mathcal{A}}$ *illustrated in Figure 2.3 and the advantage of* $\mathcal{A}$ *as*

$$\mathsf{Game}^{\text{OT-IND-CCA}}_{\mathsf{SKE},\mathcal{A}}(\kappa) := \left| \Pr\left[\mathsf{Game}^{\text{OT-IND-CCA}}_{\mathsf{SKE},\mathcal{A}}(\kappa) = 1\right] - \frac{1}{2} \right|.$$

We say that a SKE *is* one-time IND-CCA secure *if* $\mathsf{Adv}^{\text{OT-IND-CCA}}_{\mathsf{SKE},\mathcal{A}}(\kappa) \leq \mathsf{negl}(\kappa)$ *for any efficient adversaries* $\mathcal{A}$*.*

We define *key-committing* property for SKE schemes [FOR17] which roughly states that it is difficult to find two secret keys that correctly decrypt the same ciphertext (to possibly different messages). As in prior works [FOR17; GLR17; Dod+18; Alb+22], we define this notion by providing the (non-uniform) adversary oracle access to Enc_s and Dec_s, where we implicitly assume these two algorithms are implemented using an internal hash function modeled as a random oracle.

Definition 2.3.5 (Key-Committing for SKE). *We say that a* SKE *has* key-committing *property if for any efficient adversaries* $\mathcal{A}$*, we have*

$$\Pr\left[\begin{array}{ll} (\mathsf{k}_0, \mathsf{k}_1, \mathsf{c}) \leftarrow \mathcal{A}(1^\kappa), & \\ \mathsf{m}_0 \leftarrow \mathsf{Dec}_\mathsf{s}(\mathsf{k}_0, \mathsf{c}), & : \mathsf{m}_0 \neq \perp \wedge \mathsf{m}_1 \neq \perp \\ \mathsf{m}_1 \leftarrow \mathsf{Dec}_\mathsf{s}(\mathsf{k}_1, \mathsf{c}) & \end{array}\right] \leq \mathsf{negl}(\kappa).$$

Viewing SKE as (a weakened version of) AEAD, we can use [Alb+22, Sec. 5.2.] to generically transform any IND-CCA secure SKE, regardless of it being one-time secure or not, to one with the key-committing property. The transform only adds κ bits of overhead to the original ciphertext: to encrypt, the key committing scheme expands $\mathsf{k}_{\mathsf{enc}} \leftarrow \mathsf{H}_{\mathsf{enc}}(\mathsf{key})$ and $\mathsf{k}_{\mathsf{com}} \leftarrow \mathsf{H}_{\mathsf{com}}(\mathsf{key})$, runs $\mathsf{Enc}_\mathsf{s}(\mathsf{k}_{\mathsf{enc}}, \mathsf{m})$ and outputs the

$\mathsf{Game}^{\mathsf{GOAL\text{-}CMA}}_{\mathsf{SIG},\mathcal{A}}(\kappa)$

1 : $\mathsf{SL} \leftarrow \emptyset$
2 : $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\kappa)$
3 : $(\mathsf{vk}^*, \mathsf{sk}^*) \leftarrow \mathsf{KeyGen}(\mathsf{pp})$
4 : $(\mathsf{m}^*, \sigma^*) \leftarrow \mathcal{A}^{\mathtt{Sign}(\cdot)}(\mathsf{pp}, \mathsf{vk}^*)$
5 : **if** $\mathsf{GOAL} = \mathsf{EUF}$ **then**
6 : **return** $[\![\mathsf{Verify}(\mathsf{vk}^*, \mathsf{m}^*, \sigma^*) = 1 \wedge (\mathsf{m}^*, *) \notin \mathsf{SL}]\!]$
7 : **if** $\mathsf{GOAL} = \mathsf{sEUF}$ **then**
8 : **return** $[\![\mathsf{Verify}(\mathsf{vk}^*, \mathsf{m}^*, \sigma^*) = 1 \wedge (\mathsf{m}^*, \sigma^*) \notin \mathsf{SL}]\!]$

Signing Oracle $\mathtt{Sign}(\mathsf{m})$

1 : $\sigma \leftarrow \mathsf{Sign}(\mathsf{sk}^*, \mathsf{m})$
2 : $\mathsf{SL} \mathrel{+}\!\leftarrow (\mathsf{m}, \sigma)$
3 : **return** σ

FIGURE 2.4: Security games for defining the security notions of signature scheme.

ciphertext as $(\mathsf{c}, \mathsf{k}_{\mathsf{com}})$. Here $\mathsf{H}_{\mathsf{enc}}$ and $\mathsf{H}_{\mathsf{com}}$ are modeled as random oracles. Key committing simply follows from the collision resistance of $\mathsf{H}_{\mathsf{com}}$.

2.4 Digital Signatures

The syntax and the correctness of signature schemes are defined as follows.

Definition 2.4.1 (Signature Schemes). *A signature scheme with message space $\mathcal{M}$ consists of the following algorithms:*

- $\mathsf{Setup}(1^\kappa) \to \mathsf{pp}$: *The setup algorithm takes a security parameter 1^κ as input and outputs a public parameter* pp*. In the following, we assume* pp *is provided to all the algorithms and may omit it for simplicity.*
- $\mathsf{KeyGen}(\mathsf{pp}) \to (\mathsf{vk}, \mathsf{sk})$: *The key generation algorithm takes a public parameter* pp *as input and outputs a pair of verification and singing keys* $(\mathsf{vk}, \mathsf{sk})$*.*
- $\mathsf{Sign}(\mathsf{sk}, \mathsf{m}) \to \sigma$: *The signing algorithm takes a signing key* sk *and a message* $\mathsf{m} \in \mathcal{M}$ *as input and outputs a signature σ.*
- $\mathsf{Verify}(\mathsf{vk}, \mathsf{m}, \sigma) \to 1/0$: *The verification algorithm takes a verification key* vk*, a message* m *and a signature σ as input and outputs* 1 *or* 0*.*

Definition 2.4.2 ($(1-\delta)$-Correctness for SIG). *We say a signature scheme* SIG *is* $(1-\delta)$-correct *if for all $\kappa \in \mathbb{N}$, all messages $\mathsf{m} \in \mathcal{M}$ and all $\mathsf{pp} \in \mathsf{Setup}(1^\kappa)$,*

$$(1-\delta) \leq \Pr\left[(\mathsf{vk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(\mathsf{pp}), \sigma \leftarrow \mathsf{Sign}(\mathsf{sk}, \mathsf{m}) : \mathsf{Verify}(\mathsf{vk}, \mathsf{m}, \sigma) = 1\right].$$

We recall the existential unforgeability against chosen message attacks (EUF-CMA) [GMR88] and strongly existential unforgeability against chosen message attacks (sEUF-CMA) [ADR02].

Definition 2.4.3 (EUF-CMA and sEUF-CMA Security for SIG). *Let* $\mathsf{GOAL} \in \{\mathsf{EUF}, \mathsf{sEUF}\}$*. We define the game* $\mathsf{Game}^{\mathsf{GOAL\text{-}CMA}}_{\mathsf{SIG},\mathcal{A}}$ *in Figure 2.4 that an adversary $\mathcal{A}$ runs in and the advantage of $\mathcal{A}$ as*

$$\mathsf{Adv}^{\mathsf{GOAL\text{-}CMA}}_{\mathsf{SIG},\mathcal{A}}(\kappa) := \left|\Pr\left[\mathsf{Game}^{\mathsf{GOAL\text{-}CMA}}_{\mathsf{SIG},\mathcal{A}}(\kappa) = 1\right]\right|.$$

We say a signature scheme SIG *is* GOAL-CMA secure *if* $\mathsf{Adv}^{\mathsf{GOAL\text{-}CMA}}_{\mathsf{SIG},\mathcal{A}}(\kappa) \leq \mathsf{negl}(\kappa)$ *for any efficient adversaries* $\mathcal{A}$.

2.5 Pseudo-Random Functions and Pseudo-Random Permutations

Let $\mathsf{F} : \mathcal{K} \times \mathcal{D} \to \mathcal{R}$ be an efficiently computable function family with key space $\mathcal{K}$, domain $\mathcal{D}$ and finite range $\mathcal{R}$. The security notions of pseudo-random functions [GGM84] and pseudo-random permutations [LR88] are defined as follows. Below, we note that the adversary $\mathcal{A}$ is only allowed to make classical queries to the oracles.

Definition 2.5.1 (Pseudo-Random Function and Permutation Family). *Let* $\mathcal{A}$ *be an adversary and* $\mathsf{Func}_{\mathcal{D}\to\mathcal{R}}$ *be a set of all functions whose domain and range are* $\mathcal{D}$ *and* $\mathcal{R}$*, respectively. We define the advantage of* $\mathcal{A}$ *as*

$$\mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F},\mathcal{A}}(\kappa) := \left|\Pr\left[\mathsf{K} \leftarrow\!\$\, \mathcal{K} : 1 \leftarrow \mathcal{A}^{\mathsf{F}(\mathsf{K},\cdot)}(1^\kappa)\right] - \Pr\left[\mathsf{RF} \leftarrow\!\$\, \mathsf{Func}_{\mathcal{D}\to\mathcal{R}} : 1 \leftarrow \mathcal{A}^{\mathsf{RF}(\cdot)}(1^\kappa)\right]\right|.$$

We say that F *is a* pseudo-random function (PRF) family *if* $\mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F},\mathcal{A}}(\kappa) \leq \mathsf{negl}(\kappa)$ *for any efficient adversaries* $\mathcal{A}$.
In particular, when $\mathcal{D} = \mathcal{R}$ *and* F *is bijective, we say that* F *is a* pseudo-random permutation (PRP) family.

2.6 Strong Randomness Extractors

The *statistical distance* between random variables X, Y over a finite domain S is defined by

$$\mathsf{SD}(X, Y) := \frac{1}{2}\sum_{s\in S} \left|\Pr[X = s] - \Pr[Y = s]\right|.$$

Strong randomness extractors are defined as follows [DRS04; NZ96].

Definition 2.6.1 (Strong Randomness Extractors). *Let* $\mathsf{Ext} : \mathcal{S} \times \mathcal{D} \to \mathcal{R}$ *be a family of efficiently computable functions with set* $\mathcal{S}$*, domain* $\mathcal{D}$ *and range* $\mathcal{R}$*, all with finite size. A function family* Ext *is a* strong $(\lambda, \varepsilon_{\mathsf{Ext}})$-extractor *if for any random variable* X *over* $\mathcal{D}$ *with* $\Pr[X = x] \leq 2^{-\lambda}$ *(i.e.,* X *has min-entropy at least* λ*), if* s *and* R *are chosen uniformly at random from* $\mathcal{S}$ *and* $\mathcal{R}$*, respectively, the two distributions* $(s, \mathsf{Ext}_s(X))$ *and* (s, R) *are within statistical distance* $\varepsilon_{\mathsf{Ext}}$*, that is*

$$\mathsf{SD}((s, \mathsf{Ext}_s(X)), (s, R)) \leq \varepsilon_{\mathsf{Ext}}.$$

2.7 Ring Signatures

The syntax of ring signature schemes is defined as follows.

Definition 2.7.1 (Ring Signature Schemes). *A ring signature scheme* RS *consists of the following algorithms:*

- $\mathsf{Setup}(1^\kappa) \to \mathsf{pp}$: *The setup algorithm takes a security parameter* 1^κ *as input and outputs a public parameters* pp *used by the scheme. In the following, we assume* pp *is provided to all the algorithms and may omit it for simplicity.*
- $\mathsf{KeyGen}(\mathsf{pp}) \to (\mathsf{vk}, \mathsf{sk})$: *The key generation algorithm takes the public parameters* pp *as input and outputs a pair of verification and signing keys* $(\mathsf{vk}, \mathsf{sk})$*. We denote* $\mathcal{R}_{\mathsf{RS}}$*by the randomness space of* KeyGen *algorithm.*

$\mathsf{Game}^{\text{RS-Unf}}_{\mathsf{RS},\mathcal{A}}(\kappa, N)$

$\mathsf{SL}, \mathsf{CL} \leftarrow \emptyset$
$\mathsf{pp} \leftarrow \mathsf{Setup}(1^\kappa)$
foreach $i \in [N]$ **do**
  $\mathsf{r}_i \leftarrow\!\$\, \mathcal{R}_{\mathsf{RS}}$
  $(\mathsf{vk}_i, \mathsf{sk}_i) \leftarrow \mathsf{KeyGen}(\mathsf{pp}; \mathsf{r}_i)$
$\mathsf{VK} := \{\, \mathsf{vk}_i \mid i \in [N] \,\}$
$(\mathsf{R}^*, \mathsf{m}^*, \sigma^*) \leftarrow \mathcal{A}^{\mathtt{Sign}(\cdot,\cdot,\cdot),\mathtt{Corr}(\cdot)}(\mathsf{pp}, \mathsf{VK})$
$b_1 \leftarrow [\![\mathsf{R}^* \subset \mathsf{VK} \backslash \mathsf{CL}]\!]$
$b_2 \leftarrow [\![(*, \mathsf{m}^*, \mathsf{R}^*) \notin \mathsf{SL}]\!]$
$b_3 \leftarrow [\![\mathsf{Verify}(\mathsf{R}^*, \mathsf{m}^*, \sigma^*) = 1]\!]$
return $b_1 \wedge b_2 \wedge b_3$

Singing Oracle $\mathtt{Sign}(i, \mathsf{m}, \mathsf{R})$

if $\mathsf{vk}_i \in \mathsf{R}$ **then**
  return $\perp$
$\sigma \leftarrow \mathsf{Sign}(\mathsf{sk}_i, \mathsf{m}, \mathsf{R})$
$\mathsf{SL} \mathrel{+}\!\leftarrow (i, \mathsf{m}, \mathsf{R})$
return σ

Corruption Oracle $\mathtt{Corr}(i)$

$\mathsf{CL} \mathrel{+}\!\leftarrow \mathsf{vk}_i$
return r_i

FIGURE 2.5: Security game for defining the unforgeability of ring signature schemes.

- $\mathsf{Sign}(\mathsf{sk}, \mathsf{m}, \mathsf{R}) \to \sigma$: *The signing algorithm takes a signing key* sk*, a message* m*, and a list of verification keys, i.e., a* ring, $\mathsf{R} = \{\mathsf{vk}_1, \ldots, \mathsf{vk}_N\}$ *as input and outputs a signature* σ*.*
- $\mathsf{Verify}(\mathsf{R}, \mathsf{m}, \sigma) \to 1/0$: *The verification algorithm takes a ring* $\mathsf{R} = \{\mathsf{vk}_1, \ldots, \mathsf{vk}_N\}$*, a message* m*, and a signature* σ *as input and outputs either* 1 *or* 0*.*

We require a ring signature scheme RS to satisfy the following properties: correctness, anonymity, and unforgeability [BKM06].

Definition 2.7.2 ($(1-\delta)$-Correctness for RS). *We say a ring signature scheme* RS *is* $(1-\delta)$-correct *if for all* $\kappa \in \mathbb{N}$*,* $N = \mathsf{poly}(\kappa)$*,* $j \in [N]$*, and every message* m*,*

$$(1-\delta) \leq \Pr\left[\begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\kappa), \\ (\mathsf{vk}_i, \mathsf{sk}_i) \leftarrow \mathsf{KeyGen}(\mathsf{pp})\ \forall i \in [N], \\ \mathsf{R} := (\mathsf{vk}_1, \cdots, \mathsf{vk}_N), \\ \sigma \leftarrow \mathsf{Sign}(\mathsf{sk}_j, \mathsf{m}, \mathsf{R}) \end{array} \;:\; \mathsf{Verify}(\mathsf{R}, \mathsf{m}, \sigma) = 1\right].$$

Definition 2.7.3 (Anonymity for RS). *We say a ring signature scheme* RS *is* anonymous *if, for any* $\kappa \in \mathbb{N}$*,* $\mathsf{pp} \in \mathsf{Setup}(1^\kappa)$*,* $(\mathsf{vk}_0, \mathsf{sk}_0), (\mathsf{vk}_1, \mathsf{sk}_1) \in \mathsf{KeyGen}(\mathsf{pp})$*, and message* m*, the two distributions*

$$D_0 := \{\, \sigma : \sigma \leftarrow \mathsf{Sign}(\mathsf{sk}_0, \mathsf{m}, \{\, \mathsf{vk}_0, \mathsf{vk}_1 \,\}) \,\} \quad \textit{and} \quad D_1 := \{\, \sigma : \sigma \leftarrow \mathsf{Sign}(\mathsf{sk}_1, \mathsf{m}, \{\, \mathsf{vk}_0, \mathsf{vk}_1 \,\}) \,\}$$

are indistinguishable for any efficient distinguishers $\mathcal{A}$

Definition 2.7.4 (Unforgeability for RS). *Let* $N = \mathsf{poly}(\kappa)$ *be an integer. We define the game* $\mathsf{Game}^{\text{RS-Unf}}_{\mathsf{RS},\mathcal{A}}$ *in Figure 2.5 that an adversary* $\mathcal{A}$ *runs in and the advantage of* $\mathcal{A}$ *as*

$$\mathsf{Adv}^{\text{RS-Unf}}_{\mathsf{RS},\mathcal{A}}(\kappa) := \Pr\left[\mathsf{Game}^{\text{RS-Unf}}_{\mathsf{RS},\mathcal{A}}(\kappa, N) = 1\right].$$

We say a ring signature scheme RS *is* unforgeable *if* $\mathsf{Adv}^{\text{RS-Unf}}_{\mathsf{RS},\mathcal{A}}(\kappa) \leq \mathsf{negl}(\kappa)$ *for any efficient adversaries* $\mathcal{A}$*.*

2.8 Non-Interactive Zero-Knowledge

Let $\mathcal{R} \subseteq \{0,1\}^* \times \{0,1\}^*$ be a polynomial time recognizable binary relation. For $(x,w) \in \mathcal{R}$, we call x the statement and w the witness. Let $\mathcal{L}$ be the corresponding **NP** language $\mathcal{L} = \{x \mid \exists w \text{ s.t. } (x,w) \in \mathcal{R}\}$. Below, we define non-interactive zero-knowledge arguments for **NP** languages.

The syntax of NIZK arguments is defined as follows.

Definition 2.8.1 (NIZK Arguments). *A non-interactive zero-knowledge (NIZK) argument* NIZK *for the relation* $\mathcal{R}$ *consists of the following algorithms.*

- $\mathsf{Setup}(1^\kappa) \to \mathsf{crs}$: *The setup algorithm takes the security parameter* 1^κ *as input and outputs a common reference string* crs*.*
- $\mathsf{Prove}(\mathsf{crs}, x, w) \to \pi$: *The prover's algorithm takes as input a common reference string* crs*, a statement* x*, and a witness* w *and outputs a proof* π*.*
- $\mathsf{Verify}(\mathsf{crs}, x, \pi) \to 1/0$: *The verifier's algorithm takes as input a common reference string, a statement* x*, and a proof* π *and outputs* 1 *to indicate acceptance of the proof and* 0 *otherwise.*

We require a NIZK argument NIZK to satisfy the following properties: (perfect) correctness, soundness, and zero-knowledge [BFM88; FLS90].

Definition 2.8.2 (Correctness for NIZK argument). *We say a NIZK argument* NIZK *is* correct *if for all pairs* $(x,w) \in \mathcal{R}$*, if we run* $\mathsf{crs} \leftarrow \mathsf{Setup}(1^\kappa)$*, then we have*

$$\Pr[\pi \leftarrow \mathsf{Prove}(\mathsf{crs}, x, w) : \mathsf{Verify}(\mathsf{crs}, x, \pi) = 1] = 1.$$

Definition 2.8.3 (Soundness for NIZK argument). *We say a* NIZK *argument* NIZK *is* sound *if for any efficient adversaries* $\mathcal{A}$*, if we run* $\mathsf{crs} \leftarrow \mathsf{Setup}(1^\kappa)$*, then we have*

$$\Pr[(x,\pi) \leftarrow \mathcal{A}(1^\kappa, \mathsf{crs}) : x \notin \mathcal{L} \wedge \mathsf{Verify}(\mathsf{crs}, x, \pi) = 1] = \mathsf{negl}(\kappa).$$

Definition 2.8.4 (Zero-Knowledge for NIZK argument). *We say a* NIZK *argument* NIZK *is* zero-knowledge *if for any efficient adversaries* $\mathcal{A}$*, there exists a* PPT *simulator* $\mathsf{Sim} = (\mathsf{Sim}_1, \mathsf{Sim}_2)$ *such that if we run* $\mathsf{crs} \leftarrow \mathsf{Setup}(1^\kappa)$ *and* $(\overline{\mathsf{crs}}, \bar{\tau}) \leftarrow \mathsf{Sim}_1(1^\kappa)$*, then we have*

$$\left|\Pr\left[\mathcal{A}^{\mathcal{O}_0(\mathsf{crs},\cdot,\cdot)}(1^\kappa, \mathsf{crs}) = 1\right] - \Pr\left[\mathcal{A}^{\mathcal{O}_1(\overline{\mathsf{crs}},\bar{\tau},\cdot,\cdot)}(1^\kappa, \overline{\mathsf{crs}}) = 1\right]\right| = \mathsf{negl}(\kappa),$$

where $\mathcal{O}_0(\mathsf{crs}, x, w)$ *outputs* $\mathsf{Prove}(\mathsf{crs}, x, w)$ *if* $(x,w) \in \mathcal{R}$ *and* 0 *otherwise, and* $\mathcal{O}_1(\overline{\mathsf{crs}}, \bar{\tau}, x, w)$ *outputs* $\mathsf{Sim}_2(\overline{\mathsf{crs}}, \bar{\tau}, x)$ *if* $(x,w) \in \mathcal{R}$ *and* 0 *otherwise.*

2.9 Decomposable Multi-Recipient PKE

Decomposable multi-recipient PKE (mPKE) was introduced in [Kat+20]. Similar to standard mPKEs [Kur02; Sma05; BF07], a decomposable mPKE allows a user to send a message to multiple recipients more efficiently than naively running a standard PKE to the individual recipients. The main difference between a decomposable and non-decomposable mPKE is whether the encryption algorithm can be decomposed into a recipient-dependent and recipient-independent part. In [Kat+20], it was shown that many assumptions

known to imply PKE (e.g., Decisional Diffie-Hellman (DDH), Learning with Errors (LWE), Supersingular Isogeny Diffie-Hellman (SIDH)) can naturally be used to construct an IND-CPA secure decomposable mPKE. In this work, we introduce a stronger security notion than those provided in [Kat+20] where we allow the adversary to adaptively corrupt users during the IND-CPA security game. Looking ahead, this notion will be important when we target an *adaptively* secure CGKA.

Definition 2.9.1 (Decomposable Multi-Recipient Public Key Encryption). *A (single-message) decomposable multi-recipient public key encryption (decomposable mPKE) scheme* mPKE *over a message space* $\mathcal{M}$ *consists of the following algorithms:*

- $\mathsf{mSetup}(1^\kappa) \to \mathsf{pp}$: *The setup algorithm takes the security parameter* 1^κ *as input and outputs a public parameter* pp*. In the following, we assume* pp *is provided to all the algorithms and may omit it for simplicity.*
- $\mathsf{mGen}(\mathsf{pp}) \to (\mathsf{ek}, \mathsf{dk})$: *The key generation algorithm takes a public parameter* pp *as input and outputs a pair of encryption key and decryption key* $(\mathsf{ek}, \mathsf{dk})$*.*
- $\mathsf{mEnc}((\mathsf{ek}_i)_{i\in[N]}, \mathsf{m}; \mathsf{r}_0, (\mathsf{r}_i)_{i\in[N]}) \to \vec{\mathsf{ct}} = (\mathsf{ct}_0, (\widehat{\mathsf{ct}}_i)_{i\in[N]})$: *The (decomposable) encryption algorithm running with randomness* $(\mathsf{r}_0, \mathsf{r}_1, \cdots, \mathsf{r}_N)$*, splits into a pair of algorithms* $(\mathsf{mEnc}^{\mathsf{i}}, \mathsf{mEnc}^{\mathsf{d}})$:
  - $\mathsf{mEnc}^{\mathsf{i}}(\mathsf{pp}; \mathsf{r}_0) \to \mathsf{ct}_0$: *On input a public parameter* pp *and randomness* r_0*, it outputs an (encryption key* <u>i</u>ndependent*) ciphertext* ct_0*.*
  - $\mathsf{mEnc}^{\mathsf{d}}(\mathsf{pp}, \mathsf{ek}_i, \mathsf{m}; \mathsf{r}_0, \mathsf{r}_i) \to \widehat{\mathsf{ct}}_i$: *On input a public parameter* pp*, an encryption key* ek_i*, a message* $\mathsf{m} \in \mathcal{M}$*, and randomness* $(\mathsf{r}_0, \mathsf{r}_i)$*, it outputs an (encryption key* <u>d</u>ependent*) ciphertext* $\widehat{\mathsf{ct}}_i$*.*
- $\mathsf{mDec}(\mathsf{dk}, \mathsf{ct}_i) \to \mathsf{m}$ *or* $\perp$: *The decryption algorithm takes a decryption key* dk *and a ciphertext* $\mathsf{ct}_i = (\mathsf{ct}_0, \widehat{\mathsf{ct}}_i)$ *as input and outputs either* $\mathsf{m} \in \mathcal{M}$ *or* $\perp \notin \mathcal{M}$*.*

Remark 2.9.2 (decomposable mPKE from standard PKE). Observe that any standard PKE can be used to construct a decomposable mPKE in the obvious way where $\mathsf{mEnc}^{\mathsf{i}}$ is the null function and $\mathsf{mEnc}^{\mathsf{d}}$ is the encryption algorithm of the PKE. So naturally, the main motivation for mPKE will be to reuse a large portion of the encryption randomness r_0 for all recipients and to obtain a more efficient scheme compared to the obvious solution. The asymptotic behavior will be the same as the obvious solution (i.e., the total ciphertext size is $O(N)$) but the concrete size can be drastically reduced (see Section 4.5 for more details).

We require the standard notion of correctness and ciphertext-spreadness [FO99], where the latter roughly states that the probability of generating an identical ciphertext is negligibly small if we use proper randomness.

Definition 2.9.3 (Correctness for mPKE). *We say an* mPKE *is* correct *if*

$$1 - \mathsf{negl}(\kappa) \leq \mathbb{E}\left[\max_{\mathsf{m}\in\mathcal{M}} \Pr\left[\begin{array}{c} \mathsf{ct}_0 \leftarrow \mathsf{mEnc}^{\mathsf{i}}(\mathsf{pp}), \\ \widehat{\mathsf{ct}} \leftarrow \mathsf{mEnc}^{\mathsf{d}}(\mathsf{pp}, \mathsf{ek}, \mathsf{m}) \end{array} : \mathsf{m} = \mathsf{mDec}(\mathsf{dk}, (\mathsf{ct}_0, \widehat{\mathsf{ct}}))\right]\right], \tag{2.1}$$

where the expectation is taken over $\mathsf{pp} \leftarrow \mathsf{mSetup}(1^\kappa)$ *and* $(\mathsf{ek}, \mathsf{dk}) \leftarrow \mathsf{mGen}(\mathsf{pp})$*.*

Definition 2.9.4 (Ciphertext-Spreadness for mPKE). *We say an* mPKE *is* ciphertext-spread *if for all* $\mathsf{pp} \in \mathsf{mSetup}(1^\kappa)$*, and* $(\mathsf{ek}, \mathsf{dk}) \in \mathsf{mGen}(\mathsf{pp})$*,*

$$\mathbb{E}\left[\max_{\mathsf{ct}^*, \mathsf{m}\in\mathcal{M}} \Pr_{\mathsf{r}_0, \mathsf{r}}\left[\mathsf{ct} \leftarrow \left(\mathsf{mEnc}^{\mathsf{i}}(\mathsf{pp}; \mathsf{r}_0), \mathsf{mEnc}^{\mathsf{d}}(\mathsf{pp}, \mathsf{ek}, \mathsf{m}; \mathsf{r}_0, \mathsf{r})\right) : \mathsf{ct}^* = \mathsf{ct}\right]\right] \leq \mathsf{negl}(\kappa),$$

where the expectation is taken over $\mathsf{pp} \leftarrow \mathsf{mSetup}(1^\kappa)$ *and* $(\mathsf{ek}, \mathsf{dk}) \leftarrow \mathsf{mGen}(\mathsf{pp})$*.*

$\mathsf{Game}^{\mathsf{IND\text{-}CPA}}_{\mathsf{mPKE},\mathcal{A}}(\kappa, N)$

1 : $\mathsf{CL} \leftarrow \emptyset$
2 : $\mathsf{pp} \leftarrow \mathsf{mSetup}(1^\kappa)$
3 : **foreach** $i \in [N]$ **do**
4 : $\quad(\mathsf{ek}_i, \mathsf{dk}_i) \leftarrow \mathsf{mGen}(\mathsf{pp})$
5 : $(\mathsf{m}_0, \mathsf{m}_1, S \subseteq [N]) \leftarrow \mathcal{A}^{\mathtt{Corr}(\cdot)}(\mathsf{pp}, (\mathsf{ek}_i)_{i\in[N]})$
6 : $b \leftarrow\$ \{0,1\}$
7 : $\vec{\mathsf{ct}}^* \leftarrow \mathsf{mEnc}(\mathsf{pp}, (\mathsf{ek}_i)_{i\in S}, \mathsf{m}_b)$
8 : $b' \leftarrow \mathcal{A}^{\mathtt{Corr}(\cdot)}(\mathsf{pp}, (\mathsf{ek}_i)_{i\in[N]}, \vec{\mathsf{ct}}^*)$
9 : **if** $\mathsf{CL} \cap S \neq \emptyset$ **then**
10 : $\quad$ **return** b
11 : **return** $[\![b = b']\!]$

Corruption Oracle $\mathtt{Corr}(i)$

1 : $\mathsf{CL} \mathrel{+}\!\leftarrow i$
2 : **return** dk_i

FIGURE 2.6: Security games for defining the IND-CPA with adaptive corruption security of mPKE.

We also define indistinguishability of chosen plaintext attacks (IND-CPA) *with adaptive corruption* for a decomposable mPKE following [Kat+20].

Definition 2.9.5 (IND-CPA security for mPKE). *We define the game* $\mathsf{Game}^{\mathsf{IND\text{-}CPA}}_{\mathsf{mPKE},\mathcal{A}}$ *illustrated in Figure 2.6. A decomposable* mPKE *is* IND-CPA secure with adaptive corruption *if for any efficient adversaries* $\mathcal{A}$*, we have*

$$\left|\Pr\left[\mathsf{Game}^{\mathsf{IND\text{-}CPA}}_{\mathsf{mPKE},\mathcal{A}}(\kappa) = 1\right] - \frac{1}{2}\right| \leq \mathsf{negl}(\kappa)$$

If $\mathcal{A}$ *is not given access to the corruption oracle* `Corr`*, this game corresponds to the standard* IND-CPA *security.*

2.10 Message Authentication Codes

We provide the standard notion of (deterministic) message authentication codes (MACs).

Definition 2.10.1 (Message Authentication Code). *A (deterministic) message authentication code* MAC *over a key space* $\mathcal{K}$ *and a message space* $\mathcal{M}$ *consists of the following algorithms:*

- $\mathsf{Gen}(\mathsf{k}, \mathsf{m}) \to \mathsf{tag}$: *On input a key* $\mathsf{k} \in \mathcal{K}$ *and a message* $\mathsf{m} \in \mathcal{M}$*, it (deterministically) outputs a tag* tag*.*
- $\mathsf{Verify}(\mathsf{k}, \mathsf{m}, \mathsf{tag}) \to 1/0$: *On input a key* k*, a message* m *and a tag* tag*, it (deterministically) outputs* 1 *or* 0*.*

Since the Gen algorithm is deterministic, we can simply define Verify to run Gen on (k, m) and check if the generated tag' is identical to the provided tag.

Definition 2.10.2 (Correctness for MAC). *A* MAC *is correct if for all keys* $\mathsf{k} \in \mathcal{K}$ *and all messages* $\mathsf{m} \in \mathcal{M}$*,*

$$\Pr\left[\mathsf{Verify}(\mathsf{k}, \mathsf{m}, \mathsf{Gen}(\mathsf{k}, \mathsf{m})) = 1\right] = 1.$$

We define collision resistance of MAC by providing the (non-uniform) adversary oracle access to Gen and Verify, where we implicitly assume these two algorithms are implemented using an internal hash function modeled as a random oracle. We note that natural and practical constructions of a MAC based on a hash function modeled as a random oracle possess this property.

Definition 2.10.3 (Collision-Resistance for MAC). *A* MAC *is collision-resistant if for any efficient adversaries* $\mathcal{A}$*, we have*

$$\Pr\left[(\mathsf{k},\mathsf{m},\mathsf{k}',\mathsf{m}',\mathsf{tag}) \leftarrow \mathcal{A}(1^\kappa) : \begin{array}{c}(\mathsf{k},\mathsf{m}) \neq (\mathsf{k}',\mathsf{m}') \wedge \\ \mathsf{Verify}(\mathsf{k},\mathsf{m},\mathsf{tag}) = \mathsf{Verify}(\mathsf{k}',\mathsf{m}',\mathsf{tag})\end{array}\right] \leq \mathsf{negl}(\kappa).$$

2.11 Key Derivation Functions

A key derivation function (KDF) is a cryptographic algorithm that derives one or more secret keys from a secret seed. In this work, HKDF, a KDF based on HMAC [Kra10], is used. It consists of the two algorithms HKDF.Extract and HKDF.Expand. The extraction algorithm $\mathsf{k} \leftarrow \mathsf{HKDF.Extract}(s_0, s_1)$ outputs an uniform and random key k if either s_0 or s_1 has high min-entropy. The expansion algorithm $\mathsf{k}_{\mathsf{lbl}} \leftarrow \mathsf{HKDF.Expand}(\mathsf{k}, \mathsf{lbl})$, on input a key k, outputs a random key $\mathsf{k}_{\mathsf{lbl}}$ for (public) label lbl. In the security proof, we model both HKDF.Extract and HKDF.Expand as a random oracle.

Chapter 3

Post-Quantum Authenticated Key Exchange for Signal Protocol[1]

3.1 Introduction

3.1.1 Contribution of This Work

In this work, we cast the X3DH protocol (see Figure 3.1) as a specific type of authenticated key exchange (AKE) protocol, which we call a *Signal-conforming AKE* protocol, and define its security model based on the vast prior work on AKE protocols (see Section 3.2). We then provide an efficient generic construction of a Signal-conforming AKE protocol based on standard cryptographic primitives: an (IND-CCA secure) KEM, a signature scheme, and a pseudorandom function (PRF) (see Section 3.3). Similar to the X3DH protocol, our Signal-conforming AKE protocol offers a strong flavor of key-compromise security. Borrowing terminologies from AKE-related literature, the proposed protocol is proven secure in the strong Canetti-Krawczyk (CK) type security models [CK01; Kra05; Fuj+12; LLM07], where the exchanged session key remains secure even if all the non-trivial combinations of the long-term secrets and session-specific secrets of the parties are compromised. In fact, the proposed protocol is more secure than the X3DH protocol since it is even secure against KCI-attacks where the parties' session-specific secrets are compromised (see Footnote 3).[2] We believe the level of security offered by our Signal-conforming AKE protocol aligns with the level of security guaranteed by the double ratchet protocol where (a specific notion of) security still holds even when such secrets are compromised.

We then provide details on how to recast our Signal-conforming AKE protocol into a key agreement protocol similar to what is used in the Signal protocol. We call this the Signal handshake protocol (see Section 3.4). Unlike standard AKE protocols, the Signal handshake protocol makes several different design choices for efficiency reasons. The most prominent difference is that informally, the Signal handshake protocol reuses the same first message of the AKE protocol for a certain period of time. While this reduces communication and computation complexity and the storage size required by the server, this negatively affects the level of forward secrecy of the underlying Signal-conforming AKE protocol. We discuss in detail the trade-off between security and efficiency incurred when transforming our Signal-conforming AKE protocol into a Signal handshake protocol in Section 3.4.2.

[1] The contents of this chapter are based on the work published in the Journal of Cryptology under the title "An Efficient and Generic Construction for Signal's Handshake (X3DH): Post-Quantum, State Leakage Secure, and Deniable" [Has+22]. The preliminary work of [Has+22] was presented at PKC 2021 [Has+21a].

[2] Although the X3DH protocol can naturally be made secure against leakage of session-specific secrets (including randomness generated within the session) by using the generic NAXOS trick, e.g., [LLM07; Fuj+12; KF14; YCL18], it typically requires additional computation. Since this negatively affects efficiency, we target AKE protocols without using the NAXOS trick. See Section 3.1.3 for more detail.

In addition, the proposed post-quantum Signal handshake protocol is implemented in C, building on the open source libraries PQClean and LibTomCrypt (see Section 3.5). The implementation [Kwi20] is fully generic and can thus be instantiated with a wide range of KEMs and signature schemes. We instantiate it with several Round 3 candidates (finalists and alternates) to the NIST post-quantum standardization process and compare the bandwidth and computation costs that result from these choices. The proposed protocol performs best with "balanced" schemes, for example, most lattice-based schemes. The isogeny-based scheme SIKE offers good bandwidth performance but entails a significant computation cost. Finally, schemes with large public keys (Classic McEliece, Rainbow, etc.) do not seem to be a good match for the proposed protocol, since these keys are transferred at each run of the protocol.

Finally, while our Signal-conforming AKE already provides a weak form of deniability, we show how to progressively strengthen its deniability by using a ring signature instead of a signature scheme and adding a non-interactive zero-knowledge proof system (NIZK) (see Section 3.6). We propose one protocol that only uses ring signatures while only being deniable against semi-honest adversaries. We then add an NIZK on top of this protocol to make it secure even against malicious adversaries. Although the construction seemingly offers (off-line) deniability against malicious adversaries similar to the X3DH protocol [Vat+20], the formal proof relies on a strong knowledge-type assumption. However, relying on such assumptions seems unavoidable considering that all known deniable AKE protocols secure against key-compromise attacks, including the X3DH protocol, rely on them [DGK06; YZ10; Vat+20]. We briefly discuss the efficiency of our Signal-conforming AKE protocol using ring signatures in Remark 3.6.14.

3.1.2 Technical Overview

We first review the X3DH protocol and abstract its required properties by viewing it through the lens of AKE protocols. We then provide an overview of how to construct a Signal-conforming AKE protocol from standard assumptions.

Recap on the X3DH Protocol. At a high level, the X3DH protocol allows for an asynchronous key exchange where two parties, say Alice and Bob, exchange a session key without having to be online at the same time. Even more, the party, say Bob, that wishes to send a secure message to Alice can do so without Alice even knowing Bob. For instance, imagine the scenario where you send a friend request and a message at the same time before being accepted as a friend. At first glance, it seems what we require is a non-interactive key exchange (NIKE) since Bob needs to exchange a key with Alice who is offline, while Alice does not yet know that Bob is trying to communicate with her. Unfortunately, solutions based on NIKEs are undesirable since they either provide weaker guarantees than standard (interactive) AKE or exhibit inefficient constructions [Ber06; CKS08; Fre+13; PS14].

The X3DH protocol circumvents this issue by considering an *untrusted server* (e.g., the Signal server) to sit in the middle between Alice and Bob to serve as a public bulletin board. That is, the parties can store and retrieve information from the server while the server is not assumed to act honestly. A simplified description of the X3DH protocol based on the classical Diffie-Hellman (DH) key exchange is provided in Figure 3.1.[3] As the first step, Alice sends her DH component $g^x \in \mathbb{G}$ and its signature σ_A[4] to the server and then possibly goes offline. We point out that Alice does *not* need to know who she will be communicating with at this point. Bob, who may ad-hocly decide to communicate with Alice, then fetches Alice's first message from the server and uploads its DH component g^y to the server. As in a typical DH key exchange,

[3] We assume Alice and Bob know each other's long-term keys. In practice, this can be enforced by "out-of-bound" authentications (see [MP16b, Section 4.1]).

[4] In the actual protocol [MP16b; Per16], XEdDSA is used as the signature scheme, and the same long-term key (a, g^a) is used for both key exchange and signing.

Alice		Server		Bob
$(\mathsf{lpk}_\mathsf{A} = (g^a, \mathsf{vk}_\mathsf{A}), \mathsf{lsk}_\mathsf{A} = (a, \mathsf{sk}_\mathsf{A}))$				$(\mathsf{lpk}_\mathsf{B} = (g^b, \mathsf{vk}_\mathsf{B}), \mathsf{lsk}_\mathsf{B} = (b, \mathsf{sk}_\mathsf{B}))$
$x \leftarrow\$ \mathbb{Z}_p$ $\sigma_\mathsf{A} \leftarrow \mathsf{Sign}(\mathsf{sk}_\mathsf{A}, g^x)$ Store x Upload g^x, σ_A to server `-- go offline --`	$\xrightarrow{g^x, \sigma_\mathsf{A}}$	Store $(\text{Alice}, g^x, \sigma_\mathsf{A})$	$\xrightarrow{g^x, \sigma_\mathsf{A}}$	Fetch $(\text{Alice}, g^x, \sigma_\mathsf{A})$ $\mathsf{Verify}(\mathsf{vk}_\mathsf{A}, g^x, \sigma_\mathsf{A}) \stackrel{?}{=} 1$ $y \leftarrow\$ \mathbb{Z}_p$ $\mathsf{k}_\mathsf{B} := \mathsf{KDF}((g^x)^b, (g^a)^y, (g^x)^y)$ Upload g^y to server Erase y
`-- come online --` Fetch ((Alice, Bob), g^y) $\mathsf{k}_\mathsf{A} := \mathsf{KDF}((g^b)^x, (g^y)^a, (g^y)^x)$	$\xleftarrow{g^y}$	Store $((\text{Alice}, \text{Bob}), g^y)$	$\xleftarrow{g^y}$	

FIGURE 3.1: Simplified description of the X3DH Protocol. Alice and Bob have the long-term key pairs $(\mathsf{lpk}_\mathsf{A}, \mathsf{lsk}_\mathsf{A})$ and $(\mathsf{lpk}_\mathsf{B}, \mathsf{lsk}_\mathsf{B})$, respectively. Alice and Bob agree on a session key $\mathsf{k}_\mathsf{A} = \mathsf{k}_\mathsf{B}$, where KDF denotes a key derivation function.

Bob computes the session key k_B using the long-term secret exponent $b \in \mathbb{Z}_p$ and session-specific secret exponent $y \in \mathbb{Z}_p$. Since Bob can compute the session key k_B while Alice is offline, he can begin executing the subsequent double ratchet protocol without waiting for Alice to come online.[5] Whenever Alice comes online, she can fetch whatever message Bob sent from the server.

Casting the X3DH Protocol as an AKE Protocol. It is not difficult to see that the X3DH protocol can be cast as a specific type of AKE protocol. In particular, we can think of the server as an adversary that tries to mount a person-in-the-middle attack in a standard AKE protocol. Viewing the server as a malicious adversary, rather than some semi-honest entity, has two benefits: the parties do not need to put trust in the server since the protocol is supposed to be secure even against a malicious server, while the server or the company providing the app is relieved from having to "prove" that it is behaving honestly. One distinguishing feature required by the X3DH protocol when viewed as an AKE protocol is that it needs to be a two-round protocol where the initiator message is generated *independently* from the responder. That is, Alice needs to be able to store her first message to the server without knowing whom she will be communicating. In this work, we define an AKE protocol with such functionality as a *Signal-conforming* AKE protocol.

Regarding the security model for a Signal-conforming AKE protocol, we base it on the vast prior works on AKE protocols. Specifically, we build on the recent formalizations of [GJ18; Coh+19] that study the tightness of efficient AKE protocols (including a slight variant of the X3DH protocol) and strengthen the model to also incorporate *state leakage* compromise; a model where an adversary can obtain session-specific information called *session-state*. Since the double ratchet protocol considers a very strong form of state leakage security, we believe it would be the most rational design choice to discuss the X3DH protocol in a security model that captures such leakage as well. Informally, we consider our Signal-conforming AKE protocol in the Canetti-Krawczyk (CK) type security model [CK01; Kra05; Fuj+12; LLM07], which is a

[5] In practice, Bob may initiate the double ratchet protocol using k_B and send his message to Alice along with g^y to the server before Alice responds.

strengthening of the Bellare-Rogaway security model [BR94] considered by [GJ18; Coh+19]. A detailed discussion and comparison between the model used in this work and the numerous other security models of AKE protocols are provided in Section 3.2.

Lack of Signal-Conforming AKE Protocol. The main feature of a Signal-conforming AKE protocol is that the initiator's message is *independent* of the responder. Although this seems like a very natural feature considering DH-type AKE protocols, it turns out that they are quite unique (see Brendel et al. [Bre+20] for some discussion). For instance, as far as we are aware, the only other assumption that allows for a clean analog of the X3DH protocol is based on the *gap* CSIDH assumption recently introduced by De Kock et al. [KGV20] and Kawashima et al. [Kaw+20]. Considering the community is still in the process of assessing the concrete parameter selection for *standard* CSIDH [BS20b; Pei20], it would be desirable to base the X3DH protocol on more well-established and versatile assumptions. On the other hand, the known generic constructions of AKE protocols [Fuj+12; Fuj+13; KF14; YCL18; Xue+18; Höv+20; Xue+20] that can be instantiated from versatile assumptions, including post-quantum ones, can be observed that they are either not Signal-conforming or require the NAXOS trick [LLM07] (see Section 3.1.3) to be made secure against leakage of session-specific secrets.

Proposed Construction. To this end, in this work, we provide a new practical generic construction of a Signal-conforming AKE protocol from an (IND-CCA secure) KEM and a signature scheme. We believe this may be of independent interest in other scenarios where we require an AKE protocol that has a flavor of "receiver obliviousness."[6] The construction is simple: let us assume Alice and Bob's long-term keys consist of KEM key pairs $(\mathsf{ek}_\mathsf{A}, \mathsf{dk}_\mathsf{A})$ and $(\mathsf{ek}_\mathsf{B}, \mathsf{dk}_\mathsf{B})$ and signature key pairs $(\mathsf{vk}_\mathsf{A}, \mathsf{sk}_\mathsf{A})$ and $(\mathsf{vk}_\mathsf{B}, \mathsf{sk}_\mathsf{B})$, respectively. The Signal-conforming AKE protocol then starts by Alice (i.e., the initiator) generating a session-specific KEM key $(\mathsf{ek}_T, \mathsf{dk}_T)$, creating a signature $\sigma_\mathsf{A} \leftarrow \mathsf{SIG.Sign}(\mathsf{sk}_\mathsf{A}, \mathsf{ek}_T)$, and sending $(\mathsf{ek}_T, \sigma_\mathsf{A})$ to Bob (i.e., the responder). Here, observe that Alice's message does not depend on who she will be communicating with. Bob then verifies the signature and then constructs two ciphertexts: one using Alice's long-term key $(\mathsf{K}_\mathsf{A}, \mathsf{C}_\mathsf{A}) \leftarrow \mathsf{KEM.Encap}(\mathsf{ek}_\mathsf{A})$ and another using the session-specific key $(\mathsf{K}_T, \mathsf{C}_T) \leftarrow \mathsf{KEM.Encap}(\mathsf{ek}_T)$. It then signs these ciphertext $\mathsf{m} := (\mathsf{C}_\mathsf{A}, \mathsf{C}_T)$ as $\sigma_\mathsf{B} \leftarrow \mathsf{SIG.Sign}(\mathsf{sk}_\mathsf{B}, \mathsf{m})$, where we include other session-specific components in m in the actual construction. Since sending σ_B in the clear may serve as public evidence that Bob communicated with Alice, Bob will hide this. To this end, he derives two keys, a session key k_AKE and a one-time pad key k_OTP, by running a key derivation function on input the random KEM keys $(\mathsf{K}_\mathsf{A}, \mathsf{K}_T)$. Bob then sends $(\mathsf{C}_\mathsf{A}, \mathsf{C}_T, \mathsf{c} := \sigma_\mathsf{B} \oplus \mathsf{k}_\mathsf{OTP})$ to Alice and sets the session key as k_AKE. Here, note that we do not require Alice to hide her signature σ_A since this can only reveal that she was using the Signal app, unlike σ_B that may reveal who Bob was talking to. Once Alice receives the message from Bob, she decrypts the ciphertexts $(\mathsf{C}_\mathsf{A}, \mathsf{C}_T)$, derives the two keys $(\mathsf{k}_\mathsf{AKE}, \mathsf{k}_\mathsf{OPT})$, and checks if $\sigma := \mathsf{c} \oplus \mathsf{k}_\mathsf{OTP}$ is a valid signature of Bob's. If so, she sets the session key as k_AKE. We provide a formal proof and show that the proposed protocol satisfies a strong flavor of security where the shared session key remains pseudorandom even to an adversary that can obtain any non-trivial combinations of the long-term private keys (i.e., $\mathsf{dk}_\mathsf{A}, \mathsf{dk}_\mathsf{B}, \mathsf{sk}_\mathsf{A}, \mathsf{sk}_\mathsf{B}$) and session-specific secret keys (i.e., dk_T). Notably, the proposed protocol satisfies a stronger notion of security compared to the X3DH protocol since it prevents an adversary to impersonate Alice even if her session-specific secret key is compromised [MP16b, Section 4.6].

Finally, our Signal-conforming AKE protocol already satisfies a limited form of deniability where the publicly exchanged messages do not directly leak the participant of the protocol. However, if Alice at a later point gets compromised or turns malicious, she can publicize the signature σ_B sent from Bob to cryptographically prove that Bob was communicating with Alice. This is in contrast to the X3DH protocol which does not allow such a deniability attack. We, therefore, show that we can protect Bob from such

[6] This property has also been called *post-specified peers* [CK02] in the context of Internet Key Exchange (IKE) protocols.

attacks by replacing the signature scheme with a *ring* signature scheme. In particular, Alice now further sends a session-specific ring signature verification key vk_T, and Bob signs to the ring $\{ \mathsf{vk}_T, \mathsf{vk}_\mathsf{B} \}$. Effectively, when Alice outputs a signature from Bob $\sigma_{\mathsf{B},T}$, she cannot fully convince third parties whether it originates from Bob since she could have signed $\sigma_{\mathsf{B},T}$ using her signing key sk_T corresponding to vk_T. Since we only require a ring of two users, we can use existing efficient post-quantum ring signatures to instantiate this idea. For example, targeting NIST security level 1, we have 2.5 KiB for Raptor [LAZ19] (based on NTRU), 4.4 KiB for DualRing [Yue+21] (based on M-LWE/SIS), and 3.5 KiB for Calamari [BKP20] (based on CSIDH).

Although the intuition is clear, it turns out that turning this into a formal proof is quite difficult and we observe that for some practical ring signature schemes, this method only provides deniability against *semi-honest* adversaries, which are types of adversaries that follow the protocol description honestly. We provide a concrete attack where a malicious Alice that registers malformed key packages to the server can later (informally) prove to a third party that Bob was trying to communicate with her even if Bob used a ring signature to sign. We thus propose another protocol that additionally uses NIZKs to make it secure even against malicious adversaries. Similar to all previous works on AKE protocols satisfying a strong flavor of key-compromise security [DGK06; YZ10] (including the X3DH protocol [Vat+20]), the proof of deniability against malicious adversaries relies on a strong knowledge-type assumption.

3.1.3 Related and Subsequent Work

On the NAXOS trick. The NAXOS trick is a generic/artificial method to boost AKE protocols to be secure with respect to randomness exposure attacks. At a high level, whenever a party is generating a message for its peer, it will not simply use fresh randomness but extract randomness by feeding fresh randomness sampled within that session *and* its long-term secret key into a randomness extractor. Intuitively, this makes the protocol secure against randomness exposure attacks since even if the fresh randomness sampled during the session is completely exposed, the extracted randomness remains random as long as the long-term secret key is not compromised.

The NAXOS trick was originally proposed by LaMacchia et al. [LLM07] in the random oracle model. Fujioka et al. [Fuj+12] proposed a new primitive called twisted pseudo-random functions (PRF) in the standard model to mimic its properties and showed how to construct a twisted PRF from any PRF. Alawatugoda et al. [ASB14] also showed that we can mimic the NAXOS trick in the standard model by using a KEM that satisfies a non-standard notion of pair-generation indistinguishability.

In a two-round AKE protocol, the initiator and responder can both use the NAXOS trick but with different consequences. While the NAXOS trick adds a noticeable overhead in the computation time for the initiator, it adds almost none for the responder. The reason for this asymmetry is that the initiator must perform a wasteful recomputation of (part of) the first message it sent in order to process the second message sent by the responder. For instance, the initiator may generate a public key and a secret key for a KEM using the derived randomness with the NAXOS trick. In order to be secure even when the session-specific secret (i.e., the secret key for the KEM) is exposed, the initiator must securely erase the secret key from its memory after it sends the first message and only store the fresh randomness sampled within the session. Later, when the initiator receives a message from the responder, it must recompute the key generation algorithm again using the randomness derived from the NAXOS trick in order to decrypt the ciphertext included in the message. Since the NAXOS trick assumes that secure erasure of memory is possible and adds a possibly wasteful recomputation step, we simply aim for a two-round AKE protocol that does not use this.

Signal-Conforming AKE Protocol using the NAXOS Trick. Kurosawa and Furukawa [KF14] generalized the Signed Diffie-Hellman key exchange to work using an IND-CPA secure KEM and a signature scheme. Since the initiator's first message does not depend on the responder's identity, the protocol is Signal-conforming. Unfortunately, the protocol is insecure against standard KCI attacks and exposure of session-specific secrets (i.e., KEM secret key). Later, Yang et al. [YCL18] showed how to strengthen the security of Kurosawa and Furukawa's protocol by using an IND-CCA secure KEM instead and by further applying the NAXOS trick developed by Alawatugoda et al. [ASB14]. Their protocol results in a secure Signal-conforming AKE protocol that uses the NAXOS trick.

Subsequent Work. After the preliminary work [Has+21a] appeared, Brendel et al. [Bre+22] posted on ePrint a post-quantum key exchange protocol that can be used in place of X3DH similar to the proposed protocol in [Has+21a]. Dobson and Galbraith [DG22] proposed an X3DH-style protocol tailored to SIDH (SI-X3DH). Details follow.

Brendel et al. [Bre+22]. We summarize their contributions and compare their protocol with the proposed protocol.

1. Brendel et al. provide a generic construction of a *deniable* Signal-conforming AKE protocol based on a *designated verifier signature* (DVS) and a KEM. They show that DVS can be instantiated from a ring signature, in which case, their core AKE protocol illustrated in [Bre+22, Figure 2] becomes almost identical to the construction in Section 3.6.2, Figure 3.5. The followings are the main minor differences.

    - We additionally encrypt the signature generated by the ring signature by a one-time pad, while they send it in the clear. This additional layer of encryption offers anonymity with almost no overhead since the transcript no longer leaks information regarding neither the sender nor the responder to a passive eavesdropper.[7] The same idea can be applied to their protocol as well.
    - They use the NAXOS trick to generate the *second message* sent by the responder. Effectively, the protocol remains secure even if the randomness sampled by the responder (i.e., Bob in Figure 3.1) is exposed to the adversary. This trick can be used generically in any AKE protocol, including the proposed one for the same net effect. Unfortunately, similar to the proposed protocol, their protocol is insecure when the randomness used to generate the *first message* is exposed. As explained above, applying the NAXOS trick on the first message requires secure erasure of memory and a wasteful recomputation step. Making the proposed protocols secure against randomness exposure of the initiator without using the NAXOS trick remains open. We note that both protocols are secure even if the session-specific secrets (i.e., the secret that is stored by the initiator in order to process the responder's second message) are exposed.

2. In their work, they show that a DVS is implied by a ring signature (for a ring of two users) and left the other implication as an open problem. In particular, this left open the possibility that a generic construction based on DVS may be more general than those based on ring signatures. In Section 3.7, we show that a DVS can be used to construct a ring signature and thus show that a generic construction based on DVS and ring signature are theoretically equivalent.[8]

[7] To be more precise, we additionally assume that the KEM ciphertext is anonymous (i.e., indistinguishable from random) as well. This is often the case for standard encryption schemes such as those based on lattices.

[8] Note that the definition of DVS and ring signature come in various flavors. Thus, this work only shows equivalence under the security properties that Brendel et al. [Bre+22] required to construct their AKE protocol. Namely, this implication relies on the fact that their DVS assumes the signature is *publicly* verifiable.

3. In their work, they depart from prior definitions of simulation-based deniability [DGK06; Dod+09; YZ10; UG15; UG18; Vat+20] and introduce a new notion of indistinguishability-based deniability. As the new definition is incomparable to the prior definitions, we provide a detailed comparison between the two definitions in Remark 3.6.5. Very roughly, their definition considers the setting where all the users honestly follow the protocol and only register valid keys to the server. The adversary (i.e., a non-user) then tries to break the deniability of the AKE protocol while giving access to the secret keys of all the users. The restriction on users behaving honestly in their definition can loosely be captured by prior definitions of deniability by restricting the adversary to be semi-honest.

4. We provide a concrete attack in Remark 3.6.15 that breaks the deniability of the proposed AKE protocol in Section 3.6.2, which we show to be secure against semi-honest adversaries. The same attack works against the protocol by Brendel et al. that is proven secure in their new deniability definition. The attack exploits the fact that malicious parties (i.e., non-honest users) can register malicious long-term keys.

5. Finally, we construct an AKE protocol secure even against malicious adversaries in Section 3.6.3 by additionally using NIZKs and strong knowledge-type assumptions, including a variant of the *plaintext-awareness* (PA) for the KEM scheme [BR95; Bel+98; BP04]. It is an interesting problem if there is a reasonable definition of deniability that suffices to use in the real world, which also allows for constructions based on more natural assumptions.

Dobson and Galbraith [DG22]. Dobson and Galbraith [DG22] proposed an X3DH-style protocol tailored to SIDH (SI-X3DH). Their construction can be seen as a replacement of the DH key exchange in the X3DH protocol with the SIDH key exchange. Their main contribution is showing that SIDH key exchanges, which are in general insecure against adaptive attacks, can be used securely by adding a zero-knowledge proof that the long-term SIDH public keys are generated honestly. They explained that the SI-X3DH protocol satisfies the same notions of security as those satisfied by the X3DH protocol. Unlike the proposed protocol and Brendel et al.'s protocol [Bre+22], their protocol does not require a ring signature to argue deniability. They require a strong knowledge-type assumption to prove the deniability of the SI-X3DH protocol, which follows similar arguments by [Vat+20] that establish the deniability of the X3DH protocol. Note that after their paper was published, the SIDH key exchange is broken by a polynomial time attack [CD22]. Thus, their protocol needs to be reconsidered against this attack.

3.2 Security Model for Signal-Conforming AKE Protocols

In this section, we define a security model for *Signal-conforming* authenticated key exchange (AKE) protocols: AKE protocols that can be used as a drop-in replacement of the X3DH protocol. We first provide in Sections 3.2.1 to 3.2.3 a game-based security model building on the recent formalization of [GJ18; Coh+19] targeting general AKE protocols. We then discuss in Section 3.2.4 the modifications needed to make it Signal-conforming. A detailed comparison and discussion between ours and other various security models for AKE protocols are provided in Section 3.2.5.

3.2.1 Execution Environment

We consider a system of μ parties $P_1, \ldots, P_\mu$. Each party P_i is represented by a set of ℓ oracles $\{ \pi_i^1, \ldots, \pi_i^\ell \}$, where each oracle corresponds to a single execution of a protocol, and $\ell \in \mathbb{N}$ is the maximum number of

protocol sessions per party. Each oracle is equipped with fixed randomness but is otherwise deterministic. Each oracle π_i^s has access to the long-term key pair $(\mathsf{lpk}_i, \mathsf{lsk}_i)$ of P_i and the public keys of all other parties, and maintains a list of the following local variables:

- rand_i^s is the randomness hard-wired to π_i^s;
- sid_i^s ("session identifier") stores the identity of the session specified by the protocol;
- Pid_i^s ("peer id") stores the identity of the intended communication partner;
- $\mathsf{result}_i^s \in \{ \perp, \texttt{accept}, \texttt{reject} \}$ indicates whether oracle π_i^s has successfully completed the protocol execution and "accepted" the resulting key;
- k_i^s stores the session key computed by π_i^s;
- state_i^s holds the (secret) session-state values and intermediary results required by the session;
- $\mathsf{role}_i^s \in \{ \perp, \texttt{init}, \texttt{resp} \}$ indicates π_i^s's role during the protocol execution.

For each oracle π_i^s, these variables, except the randomness, are initialized to $\perp$. An AKE protocol is executed interactively between two oracles. An oracle that first sends a message is called an *initiator* ($\mathsf{role} = \texttt{init}$) and a party that first receives a message is called a *responder* ($\mathsf{role} = \texttt{resp}$). The computed session key is assigned to the variable k_i^s if and only if π_i^s reaches the `accept` state, that is, $\mathsf{k}_i^s \neq \perp \iff \mathsf{result}_i^s = \texttt{accept}$.

Partnering. To exclude trivial attacks in the security game, we need to define a notion of "partnering" between two oracles. Intuitively, this dictates which oracles can be corrupted without trivializing the security game. We define the notion of partnering via session identifiers following the work of [CK01; dFW20]. Discussions on other possible choices of the definition for partnering are provided in Section 3.2.5.

Definition 3.2.1 (Partner Oracles). *For any $(i, j, s, t) \in [\mu]^2 \times [\ell]^2$ with $i \neq j$, we say that oracles π_i^s and π_j^t are* partners *if (1) $\mathsf{Pid}_i^s = j$ and $\mathsf{Pid}_j^t = i$; (2) $\mathsf{role}_i^s \neq \mathsf{role}_j^t$; and (3) $\mathsf{sid}_i^s = \mathsf{sid}_j^t$.*

For correctness, we require that two oracles executing the AKE protocol faithfully (i.e., without adversarial interaction) derive identical session identifiers. We also require that two such oracles reach the `accept` state and derive identical session keys except with all but a negligible probability. We call a set $S \subseteq ([\mu] \times [\ell])^2$ to have a *valid pairing* if the following properties hold:

- For all $((i, s), (j, t)) \in S$, $i \leq j$.
- For all $(i, s) \in [\mu] \times [\ell]$, there exists a unique $(j, t) \in [\mu] \times [\ell]$ such that $i \neq j$ and either $((i, s), (j, t)) \in S$ or $((j, t), (i, s)) \in S$.

In other words, a set with a valid pairing S partners off each oracle π_i^s and π_j^t in a way that the pairing is unique and no oracle is left out without a pair. We define the correctness of an AKE protocol as follows.

Definition 3.2.2 ($(1-\delta)$-Correctness). *We say an AKE protocol Π_{AKE} is $(1-\delta)$-*correct *if for any set with a valid pairing $S \subseteq ([\mu] \times [\ell])^2$, when we execute the AKE protocol faithfully between all the oracle pairs included in S, it holds that*

$$1 - \delta \leq \Pr \left[\begin{array}{l} \pi_i^s \text{ and } \pi_j^t \text{ are partners} \wedge \mathsf{result}_i^s = \mathsf{result}_j^t = \texttt{accept} \\ \wedge \mathsf{k}_i^s = \mathsf{k}_j^t \neq \perp \text{ for all } ((i, s), (j, t)) \in S \end{array} \right],$$

where the probability is taken over the randomness used in the oracles.

3.2.2 Security Game

We define the security of an AKE protocol Π_{AKE} via a game played between an adversary $\mathcal{A}$ and a challenger $\mathcal{C}$. We consider two slightly different variants, each denoted as $\mathsf{Game}^{\mathsf{AKE\text{-}FS}}_{\Pi_{\mathsf{AKE}}}$ and $\mathsf{Game}^{\mathsf{AKE\text{-}weakFS}}_{\Pi_{\mathsf{AKE}}}$. The former and latter capture a *weakly* and *perfect* forward secure AKE protocol, respectively. Roughly, when the long-term secret key is exposed, the former only ensures the security of past sessions where the adversary did not modify the exchanged messages. In contrast, the latter ensures the security of all past sessions regardless of the adversary actively modifying the exchanged messages. Further details on the difference are provided in Section 3.2.3. Looking ahead, the main AKE protocol in Section 3.3 achieves perfect forward secrecy, and its variants that satisfy deniability in Section 3.6 achieve weak forward secrecy.

More formally, the security game is parameterized by two integers $\mu = \mathsf{poly}(\kappa)$ (the number of honest parties) and $\ell = \mathsf{poly}(\kappa)$ (the maximum number of protocol executions per party), and proceeds as follows, where the freshness clauses Item 5a and Item 5b is used to define $\mathsf{Game}^{\mathsf{AKE\text{-}FS}}_{\Pi_{\mathsf{AKE}}}(\mu, \ell)$ and $\mathsf{Game}^{\mathsf{AKE\text{-}weakFS}}_{\Pi_{\mathsf{AKE}}}(\mu, \ell)$, respectively:

Setup: $\mathcal{C}$ first chooses a challenge bit $b \in \{0,1\}$ at random. $\mathcal{C}$ then generates the public parameter of Π_{AKE} and μ long-term key pair $\{ (\mathsf{lpk}_i, \mathsf{lsk}_i) \mid i \in [\mu] \}$, and initializes the collection of oracles $\{ \pi_i^s \mid i \in [\mu], s \in [\ell] \}$. $\mathcal{C}$ runs $\mathcal{A}$ providing the public parameter and all the long-term public keys $\{ \mathsf{lpk}_i \mid i \in [\mu] \}$ as input.

Phase 1: $\mathcal{A}$ adaptively issues the following queries any number of times in an arbitrary order:

- $\texttt{Send}(i, s, m)$: This query allows $\mathcal{A}$ to send an arbitrary message m to oracle π_i^s. The oracle will respond according to the protocol specification and its current internal state. To start a new oracle, the message m takes a special form:
  $\langle \texttt{START} : \mathsf{role}, j \rangle$; $\mathcal{C}$ initializes π_i^s in the role role, having party P_j as its peer, that is, $\mathcal{C}$ sets $\mathsf{Pid}_i^s := j$ and $\mathsf{role}_i^s := \mathsf{role}$. If π_i^s is an initiator (i.e., $\mathsf{role} = \texttt{init}$), then $\mathcal{C}$ returns the first message of the protocol.[9]
- $\texttt{RevLTK}(i)$: For $i \in [\mu]$, this query allows $\mathcal{A}$ to learn the long-term secret key lsk_i of party P_i. After this query, P_i is said to be *corrupted*.
- $\texttt{RegisterLTK}(i, \mathsf{lpk}_i)$: For $i \in \mathbb{N} \setminus [\mu]$, this query allows $\mathcal{A}$ to register a new party P_i with public key lpk_i. We do not require that the adversary knows the corresponding secret key. After the query, the pair (i, lpk_i) is distributed to all other oracles. Parties registered by `RegisterLTK` are corrupted by definition.
- $\texttt{RevState}(i, s)$: This query allows $\mathcal{A}$ to learn the session-state state_i^s of oracle π_i^s. After this query, state_i^s is said to be *revealed*.
- $\texttt{RevSessKey}(i, s)$: This query allows $\mathcal{A}$ to learn the session key k_i^s of oracle π_i^s.

Test: Once $\mathcal{A}$ decides that Phase 1 is over, it issues the following special `Test`-query which returns a real or random key depending on the challenge bit b.

- $\texttt{Test}(i, s)$: If $(i, s) \notin [\mu] \times [\ell]$ or $\mathsf{result}_i^s \neq \texttt{accept}$, $\mathcal{C}$ returns $\perp$. Else, $\mathcal{C}$ returns k_b, where $k_0 := \mathsf{k}_i^s$ and $k_1 \leftarrow\!\!\$\, \mathcal{K}$ (where $\mathcal{K}$ is the session key space).

After this query, π_i^s is said to be *tested*.

[9] Looking ahead, when the first message is independent of party P_j (i.e., $\mathcal{C}$ can first create the first message without knowledge of P_j and then set $\mathsf{Pid}_i^s := j$), we call the scheme *receiver oblivious*. See Section 3.2.4 for more details.

Phase 2: $\mathcal{A}$ adaptively issues queries as in Phase 1.

Guess: Finally, $\mathcal{A}$ outputs a guess $b' \in \{0,1\}$. At this point, the tested oracle must be *fresh*. Here, an oracle π_i^s with $\mathsf{Pid}_i^s = j$[10] is *fresh* if all the following conditions hold:

1. $\mathtt{RevSessKey}(i,s)$ has not been issued;
2. if π_i^s has a partner π_j^t for some $t \in [\ell]$, then $\mathtt{RevSessKey}(j,t)$ has not been issued;
3. P_i is not corrupted or state_i^s is not revealed;
4. if π_i^s has a partner π_j^t for some $t \in [\ell]$, then P_j is not corrupted or state_j^t is not revealed;
5. if π_i^s has no partner, then
   (a) in game $\mathsf{Game}_{\Pi_{\mathsf{AKE}}}^{\mathsf{AKE\text{-}FS}}$, P_j is corrupted only after π_i^s finishes the protocol execution.
   (b) in game $\mathsf{Game}_{\Pi_{\mathsf{AKE}}}^{\mathsf{AKE\text{-}weakFS}}$, P_j is not corrupted.

If the tested oracle is not fresh, $\mathcal{C}$ aborts the game and outputs a random bit b' on behalf of $\mathcal{A}$.

Definition 3.2.3 (Security of AKE Protocols). *Let* $\mathtt{xxx} \in \{\mathsf{weakFS}, \mathsf{FS}\}$. *The advantage of* $\mathcal{A}$ *in the security game* $\mathsf{Game}_{\Pi_{\mathsf{AKE}}}^{\mathsf{AKE\text{-}xxx}}(\mu,\ell)$ *is defined as*

$$\mathsf{Adv}_{\Pi_{\mathsf{AKE}},\mathcal{A}}^{\mathsf{AKE\text{-}xxx}}(\kappa) := \left| \Pr\left[b = b'\right] - \frac{1}{2} \right|.$$

We say an AKE protocol Π_{AKE} *is* secure with perfect (resp. weak) forward secrecy *if* $\mathsf{Adv}_{\Pi_{\mathsf{AKE}},\mathcal{A}}^{\mathsf{AKE\text{-}FS}}(\kappa) \leq \mathsf{negl}(\kappa)$ *(resp.* $\mathsf{Adv}_{\Pi_{\mathsf{AKE}},\mathcal{A}}^{\mathsf{AKE\text{-}weakFS}}(\kappa) \leq \mathsf{negl}(\kappa)$*) for any efficient adversaries* $\mathcal{A}$.

3.2.3 Security Properties

In this section, we explain the security properties captured by our security model. Comparison between other protocols is deferred to Section 3.2.5.

The freshness clauses Items 1 and 2 imply that we only exclude the reveal of session keys for the tested oracle and its partner oracles. These capture *key independence*: if the revealed session keys are different from the tested oracle's key, then such session keys must not enable computing the session key of the tested oracle. Note that key independence implies resilience to "no-match attacks" presented by Li and Schäge [LS17]. This is because revealed keys have no information on the tested oracle's key. Moreover, the two items capture *implicit authentication* between the involved parties. This is because an oracle π that computes the same session key as the tested oracle but disagrees with the peer would not be a partner of the tested oracle, and hence, an adversary can obtain the tested oracle's key by querying the session key computed by π. Specifically, our model captures resistance to *unknown key-share* (UKS) attacks [BM99]: a successful UKS attack is a specific type of attack that breaks implicit authentication where two parties compute the same session key but have different views on whom they are communicating with.

The freshness clauses Items 3 to 5 indicate that the game allows the adversary to reveal any subset of the four secret items of information — the long-term secret keys and the session-states of the two parties (where one party is the party defined by the tested oracle and the other its peer) — except for the combination where both the long-term secret key and session-state of one of the party is revealed. In particular, Item 5a captures *perfect forward secrecy* [DVW92; CK01]: the adversary can obtain the long-term secret keys of

[10] Note that by definition, the peer id Pid_i^s of a tested oracle π_i^s is always defined.

both parties once the tested oracle finishes the protocol and generates a session key. On the other hand, Item 5b captures *weak forward secrecy* [Kra05]: the adversary can obtain the long-term secret keys of both parties only when it has been passive during the protocol run of the tested oracle. In other words, if an adversary is active (i.e., inject malicious messages) during the protocol execution and further corrupts both long-term secret keys after the oracles evaluate some session keys, then the adversary can test the oracle in $\mathsf{Game}^{\mathsf{AKE\text{-}FS}}_{\Pi_{\mathsf{AKE}}}$, while it cannot in $\mathsf{Game}^{\mathsf{AKE\text{-}weakFS}}_{\Pi_{\mathsf{AKE}}}$. Another property captured by our model is resistance to *key-compromise impersonation* (KCI) attacks [BJM97]. Recall that KCI attacks are those where the adversary uses a party P_i's long-term secret key to impersonate other parties towards P_i. This is captured by our model because the adversary can learn the long-term secret key of a tested oracle without any restrictions. Most importantly, our model captures resistance to *state leakage* [CK01; Kra05; LLM07; Fuj+12] where an adversary is allowed to obtain session-states of both parties. We point out that our security model is strictly stronger than the recent models [GJ18; Coh+19] that do not allow the adversary to learn session-states. More discussion on state leakage is provided in Section 3.2.5.

3.2.4 Property for Signal-Conforming AKE: Receiver Obliviousness

In this work, we care for a specific type of (two-round) AKE protocol that is compatible with the X3DH protocol [MP16b] used by the Signal protocol [Sig]. As explained in Section 3.1.2, the X3DH protocol can be viewed as a special type of AKE protocol where the Signal server acts as an (untrusted) bulletin board, where parties can store and retrieve information. More specifically, the Signal server can be viewed as an adversary of an AKE protocol that controls the communication channel between the parties. When casting the X3DH protocol as an AKE protocol, one crucial property is that the first message of the initiator is generated *independently* of the communication partner. This is because, in secure messaging, parties are often *offline* during the key agreement so if the first message depended on the communication partner, then we must wait until they come online to complete the key agreement. Since we cannot send messages without agreeing on a session key, such an AKE protocol where the first message depends on the communication partner cannot be used as an alternative to the X3DH protocol.

We abstract this crucial yet implicit property achieved by the X3DH protocol as *receiver obliviousness*. As noted in Footnote 6, this property has also been called as *post-specified peers* [CK02] in the context of Internet Key Exchange (IKE) protocols.

Definition 3.2.4 (Receiver Obliviousness/Signal-Conforming). *An AKE protocol is* receiver oblivious *(or* Signal-conforming*) if it is two rounds and the initiator can compute the first message without knowledge of the peer identity and long-term public key of the communication peer.*

Many Diffie-Hellman type AKE protocols (e.g., the X3DH protocol used in Signal and some CSIDH-based AKE protocols [KGV20; Kaw+20]) can be checked to be receiver oblivious.

3.2.5 Relation to Other Security Models

In the literature of AKE protocols, many security models have been proposed: the Bellare-Rogaway (BR) model [BR94], the Canetti-Krawczyk (CK) model [CK01], the CK+ model [Kra05; Fuj+12], the extended CK (eCK) model [LLM07], and variants therein [CF12; Bad+15; GJ18; Coh+19; Höv+20; Jag+21]. Although many of these security models are built based on similar motivations, there are subtle differences. We point out the notable similarities and differences between our model and the models listed above.

Long-Term Key Reveal. We first compare the models with respect to the secret information the adversary is allowed to obtain. All models including ours allow the adversary to obtain the party's long-term secret

key $\{ \mathsf{lsk}_i \mid i \in [\mu] \}$. In some models such as the BR model [BR94] and its variants (e.g., [Bad+15; GJ18; Coh+19])[11], this will be the only information given to an adversary. Although this may be a restricted model, it often serves as an initial step in proving the security of an AKE protocol.

Session-State Reveal. We can also consider a stronger and more realistic security model where the adversary is allowed to obtain the *secret session-states* of the parties. Unlike a party's long-term secret key where the definition is clear from context, the notion of secret session-states is rather unclear, and this is one of the main reasons for the various incomparable security models. In the original CK model [CK01], the session-state can depend arbitrarily on the long-term secret and the randomness used by the party. More formally, using the terminology from Section 3.2.1, an adversary can query an oracle π_i^s for a secret session-state $f(\mathsf{lsk}_i, \mathsf{r}_i^s)$ for an arbitrary function f, where r_i^s is the randomness hardwired to the oracle π_i^s, and we say the AKE protocol is secure with respect to the session-state defined by f.[12] The eCK model [LLM07] and the CK+ model [Kra05; Fuj+12] made the CK model more accessible by only considering a specific but natural set of functions.[13] The eCK model defines the secret session-state as the randomness used by the oracle (i.e., $f(\mathsf{lsk}_i, \mathsf{r}_i^s) := \mathsf{r}_i^s$). On the other hand, the CK+ model defines the session-state to be what we called session-state in Section 3.2.1. More specifically, the model allows the adversary to obtain the session-state state_i^s (defined at the implementation level) for all oracles except for the tested oracle and allows the adversary to only obtain the randomness $\mathsf{r}_{i^*}^{s^*}$ of the tested oracle. As Cremers [Cre11; Cre09] points out, depending on how we define the function f, state_i^s and r_i^s, these notions provide incomparable security guarantees. For instance, we can always artificially modify the scheme so that $\mathsf{state}_i^s := \mathsf{r}_i^s$ but this usually results in an unnatural and less efficient implementation. Recent works [Höv+20; Jag+21] consider an arguably more simple and natural definition compared to the CK+ model where the adversary can obtain all the session-state state_i^s *including* the tested oracle. This seems to align with the type of state leakage considered by the double ratchet protocol and we choose to follow this formalization in our work.

Partnering. Another point of difference is how to define the partnering of two oracles, where recall that this is used to capture attacks that trivialize the security game. One popular method to define partnering of two oracles is by the so-called *matching conversations* used for instance by [BR94; Kra05; Fuj+12; LLM07; CF12; Bad+15; Coh+19; Höv+20; Jag+21]. As the name indicates, two oracles are partnered when the input and output (i.e., the conversation between the two oracles) match. One benefit of using matching conversations is that they are simple to handle; given a particular instantiation of an AKE protocol, a matching conversation is uniquely defined. However, it was recently observed by Li and Schäge [LS17] that some protocols using matching conversations are vulnerable against *no-match attacks*, where two oracles compute the same session key but do not have matching conversations. A protocol with a no-match attack allows the adversary to trivially win the security game since it can query the oracle that is not a partner of the tested oracle but computes the same session key as the tested oracle. It was noted by Li and Schäge that this is only a hypothetical attack that takes advantage of the security model and has no meaningful consequence in the real world. Therefore, in this work, we chose to use a more robust definition based on session identifiers [CK01; dFW20]. Unlike matching conversations, session identifiers must be explicitly defined for each AKE protocol and we note that if a session identifier is defined to be the concatenation of sent and received messages, then defining partnering via session identifiers and matching conversations become equivalent. Finally, we note that Li and Schäge [LS17] proposed another method

[11] We note that the subsequent variants differ from the original BR model [BR94] as they also model forward secrecy and KCI attacks.

[12] Note that the meaning of the session-state is different from those we defined in Section 3.2.1 (i.e., state_i^s). In the CK model, a "session-state" is only defined in the security model and does not capture the state_i^s specified by the implementation.

[13] These variants also strengthen the CK model by allowing the adversary to obtain the session-state of the tested oracle and by further modeling KCI attacks.

to define partnering called *original key partnering*. This has been used in [GJ18]. The *original key* of two oracles is defined as the session key that is computed when the oracles are executed faithfully. Then, in the security game (i.e., in the presence of an adversary), if two oracles compute their original key, they are said to be partners. The original key partnering is conceptually cleaner but arguably harder to handle since we need to consider two session keys for each oracle: the original key and the actual key, in the security game. Therefore, in this work, we use partnering based on session identifiers.

Number of `Test` queries. Finally, we allow the adversary to issue only one `Test`-query in the security game. This *single-challenge* setting has been widely used in the literature. However, recently, in order to evaluate the tightness of the security proof, [Bad+15; GJ18; Coh+19; Jag+21] consider the *multi-challenge* setting, where an adversary is allowed to make multiple `Test`-queries.

Remark 3.2.5 (Implicit and Explicit Authentication). Our model captures *implicit authentication*, where each party is assured that no other party aside from the intended peer can gain access to the session key. Here, note that implicit authentication does *not* guarantee that the intended peer holds the same key. What it guarantees is that although your intended peer may be computing a different key, that peer is the only possible party that can have information on your computed session key. On the other hand, the property that also guarantees that the intended peer has computed the same session key is called *explicit authentication*. In (mutual) explicit authentication protocols, if both parties reach the `accept` state, then they are guaranteed to share the same session key. In practice, the distinction between implicit and explicit authentication is a minor issue since we can always add a key confirmation step to enhance an implicit authentication AKE protocol into an explicit one [Yan14; Coh+19; dFW20]. For instance, we can send an encrypted message or a MAC tag under the established session key to check if the peer computed the same key without compromising security. In the context of Signal, the double ratchet protocol that comes after the X3DH protocol can be viewed as adding an explicit authentication step.

3.3 Generic Construction of Signal-Conforming AKE

In this section, we propose our Signal-conforming AKE protocol $\Pi_{\mathsf{SC\text{-}AKE}}$ that can be used to construct a Signal's initial key agreement (Signal handshake) protocol such as the X3DH protocol. Unlike the X3DH protocol, the proposed protocol can be instantiated from post-quantum assumptions, and moreover, it also provides stronger security against state leakage.

3.3.1 Description of Signal-Conforming AKE $\Pi_{\mathsf{SC\text{-}AKE}}$

The protocol description is presented in Figure 3.2. We emphasize that our AKE protocol is receiver oblivious since the first message is independent of the communication partner. Details follow.

Building Blocks. Our Signal-conforming AKE protocol $\Pi_{\mathsf{SC\text{-}AKE}}$ consists of the following building blocks.

- $\mathsf{KEM} = (\mathsf{KEM.Setup}, \mathsf{KEM.KeyGen}, \mathsf{KEM.Encap}, \mathsf{KEM.Decap})$ is a KEM scheme that is IND-CCA secure and assume we have $(1-\delta_{\mathsf{KEM}})$-correctness, ν_{KEM}-high encapsulation key min-entropy and χ_{KEM}-high ciphertext min-entropy.
- $\mathsf{wKEM} = (\mathsf{wKEM.Setup}, \mathsf{wKEM.KeyGen}, \mathsf{wKEM.Encap}, \mathsf{wKEM.Decap})$ is a KEM schemes that is IND-CPA secure (and not IND-CCA secure) and assume we have $(1-\delta_{\mathsf{wKEM}})$-correctness, ν_{wKEM}-high encapsulation key min-entropy, and χ_{wKEM}-high ciphertext min-entropy. In the following, for simplicity of presentation and without loss of generality, we assume $\delta_{\mathsf{wKEM}} = \delta_{\mathsf{KEM}}$, $\nu_{\mathsf{wKEM}} = \nu_{\mathsf{KEM}}$, $\chi_{\mathsf{wKEM}} = \chi_{\mathsf{KEM}}$.

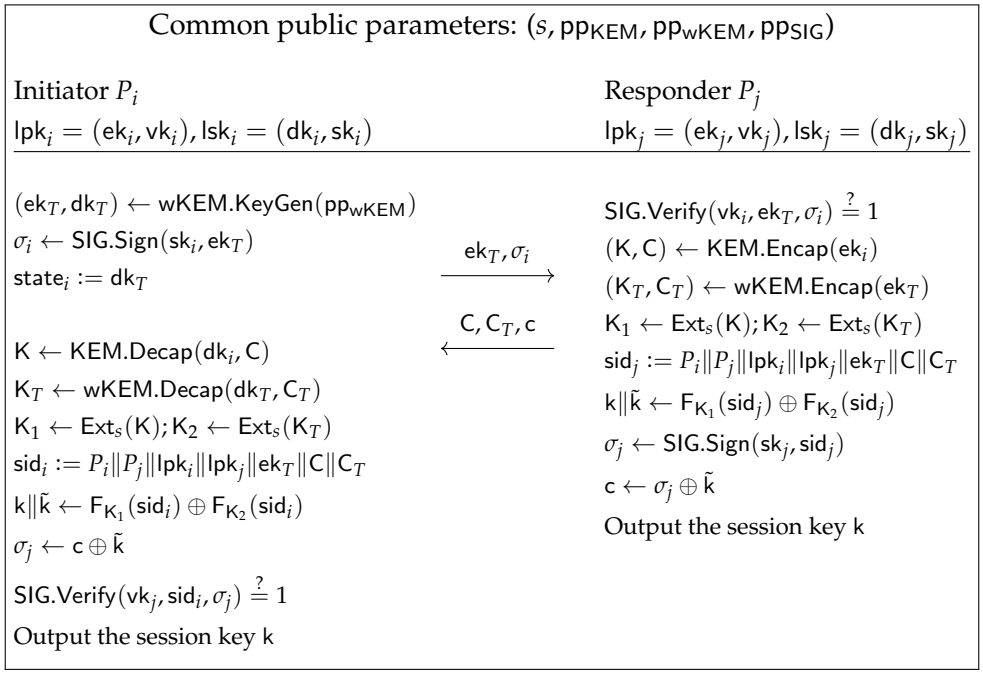


FIGURE 3.2: Our Signal-conforming AKE protocol $\Pi_{\mathsf{SC\text{-}AKE}}$.

- $\mathsf{SIG} = (\mathsf{SIG.Setup}, \mathsf{SIG.KeyGen}, \mathsf{SIG.Sign}, \mathsf{SIG.Verify})$ is a signature scheme that is EUF-CMA secure and $(1 - \delta_{\mathsf{SIG}})$-correctness. We denote d as the bit length of the signature generated by SIG.Sign.
- $\mathsf{F} : \mathcal{K} \times \{0,1\}^* \to \{0,1\}^{\kappa+d}$ is a pseudo-random function family with key space $\mathcal{K}$.
- $\mathsf{Ext} : \mathcal{S} \times \mathcal{KS} \to \mathcal{K}$ is a strong $(\gamma_{\mathsf{KEM}}, \varepsilon_{\mathsf{Ext}})$-extractor.

Public Parameters. All the parties in the system are provided with the following public parameters as input: $(s, \mathsf{pp}_{\mathsf{KEM}}, \mathsf{pp}_{\mathsf{wKEM}}, \mathsf{pp}_{\mathsf{SIG}})$. Here, s is a random seed chosen uniformly from $\mathcal{S}$ for the strong randomness extractor, and pp_{X} for $\mathsf{X} \in \{\mathsf{KEM}, \mathsf{wKEM}, \mathsf{SIG}\}$ are public parameters generated by X.Setup.

Long-Term Public and Secret Keys. Each party P_i runs $(\mathsf{ek}_i, \mathsf{dk}_i) \leftarrow \mathsf{KEM.KeyGen}(\mathsf{pp}_{\mathsf{KEM}})$ and $(\mathsf{vk}_i, \mathsf{sk}_i) \leftarrow \mathsf{SIG.KeyGen}(\mathsf{pp}_{\mathsf{SIG}})$. Party P_i's long-term public key and secret key are set as $\mathsf{lpk}_i = (\mathsf{ek}_i, \mathsf{vk}_i)$ and $\mathsf{lsk}_i = (\mathsf{dk}_i, \mathsf{sk}_i)$, respectively.

Construction. A key exchange between an initiator P_i in the s-th session (i.e., π_i^s) and responder P_j in the t-th session (i.e., π_j^t) is executed as in Figure 3.2. More formally, we have the following.

1. Party P_i sets $\mathsf{Pid}_i^s := j$ and $\mathsf{role}_i^s := \texttt{init}$. P_i computes $(\mathsf{dk}_T, \mathsf{ek}_T) \leftarrow \mathsf{wKEM.KeyGen}(\mathsf{pp}_{\mathsf{wKEM}})$ and $\sigma_i \leftarrow \mathsf{SIG.Sign}(\mathsf{sk}_i, \mathsf{ek}_T)$. Then it sends $(\mathsf{ek}_T, \sigma_i)$ to party P_j. P_i stores the ephemeral decapsulation key dk_T as the session-state, i.e., $\mathsf{state}_i^s := \mathsf{dk}_T$.[14]
2. Party P_j sets $\mathsf{Pid}_j^t := i$ and $\mathsf{role}_j^t := \texttt{resp}$. Upon receiving $(\mathsf{ek}_T, \sigma_i)$, P_j first checks whether $\mathsf{SIG.Verify}(\mathsf{vk}_i, \mathsf{ek}_T, \sigma_i) = 1$ holds. If not, P_j sets $(\mathsf{result}_j^t, \mathsf{k}_j^t, \mathsf{state}_j^t) := (\texttt{reject}, \perp, \perp)$ and stops. Otherwise, it computes $(\mathsf{K}, \mathsf{C}) \leftarrow \mathsf{KEM.Encap}(\mathsf{ek}_i)$ and $(\mathsf{K}_T, \mathsf{C}_T) \leftarrow \mathsf{wKEM.Encap}(\mathsf{ek}_T)$. Then P_j derives two

[14] Notice the protocol is receiver oblivious since the first message is computed independently of the receiver.

PRF keys $\mathsf{K}_1 \leftarrow \mathsf{Ext}_s(\mathsf{K})$ and $\mathsf{K}_2 \leftarrow \mathsf{Ext}_s(\mathsf{K}_T)$. It then defines the session identifier as $\mathsf{sid}_j^t := P_i \| P_j \| \mathsf{lpk}_i \| \mathsf{lpk}_j \| \mathsf{ek}_T \| \mathsf{C} \| \mathsf{C}_T$ and computes $\mathsf{k} \| \tilde{\mathsf{k}} \leftarrow \mathsf{F}_{\mathsf{K}_1}(\mathsf{sid}_j) \oplus \mathsf{F}_{\mathsf{K}_2}(\mathsf{sid}_j)$, where $\mathsf{k} \in \{0,1\}^\kappa$ and $\tilde{\mathsf{k}} \in \{0,1\}^d$, and sets the session key as $\mathsf{k}_j^t := \mathsf{k}$. P_j then signs $\sigma \leftarrow \mathsf{SIG.Sign}(\mathsf{sk}_j, \mathsf{sid}_j^t)$ and encrypts it as $\mathsf{c} \leftarrow \sigma \oplus \tilde{\mathsf{k}}$. Finally, it sends $(\mathsf{C}, \mathsf{C}_T, \mathsf{c})$ to P_i and sets $\mathsf{result}_j^t := \mathtt{accept}$. Here, note that P_j does not require to store any session-state, i.e., $\mathsf{state}_j^t = \bot$.

3. Upon receiving $(\mathsf{C}, \mathsf{C}_T, \mathsf{c})$, P_i first decrypts $\mathsf{K} \leftarrow \mathsf{KEM.Decap}(\mathsf{dk}_i, \mathsf{C})$ and $\mathsf{K}_T \leftarrow \mathsf{wKEM.Decap}(\mathsf{dk}_T, \mathsf{C}_T)$, and derives two PRF keys $\mathsf{K}_1 \leftarrow \mathsf{Ext}_s(\mathsf{K})$ and $\mathsf{K}_2 \leftarrow \mathsf{Ext}_s(\mathsf{K}_T)$. It then sets the session identifier as $\mathsf{sid}_i^s := P_i \| P_j \| \mathsf{lpk}_i \| \mathsf{lpk}_j \| \mathsf{ek}_T \| \mathsf{C} \| \mathsf{C}_T$ and computes $\mathsf{k} \| \tilde{\mathsf{k}} \leftarrow \mathsf{F}_{\mathsf{K}_1}(\mathsf{sid}_i) \oplus \mathsf{F}_{\mathsf{K}_2}(\mathsf{sid}_i)$, where $\mathsf{k} \in \{0,1\}^\kappa$ and $\tilde{\mathsf{k}} \in \{0,1\}^d$. P_i then decrypts $\sigma \leftarrow \mathsf{c} \oplus \tilde{\mathsf{k}}$ and checks whether $\mathsf{SIG.Verify}(\mathsf{vk}_j, \mathsf{sid}_i^s, \sigma) = 1$ holds. If not, P_i sets $(\mathsf{result}_i^s, \mathsf{k}_i^s, \mathsf{state}_i^s) := (\mathtt{reject}, \bot, \bot)$ and stops. Otherwise, it sets $(\mathsf{result}_i^s, \mathsf{k}_i^s, \mathsf{state}_i^s) := (\mathtt{accept}, \mathsf{k}, \bot)$. Here, note that P_i deletes the session-state $\mathsf{state}_i^s = \mathsf{dk}_T$ at the end of the key exchange (i.e., sets $\mathsf{state}_i^s := \bot$).

Remark 3.3.1 (A Note on Session-State). The session-state of the initiator P_i contains the ephemeral decryption key dk_T, and P_i must store it until the peer responds. Any other information that is computed after receiving the message from the peer is erased after the session key is established. In contrast, the responder P_j has no session-state because the responder directly computes the session key after receiving the initiator's message and does not need to store any session-specific information. That is, all states can be erased as soon as the session key is computed.

3.3.2 Security of Signal-Conforming AKE $\Pi_{\mathsf{SC\text{-}AKE}}$

We first prove the correctness of the proposed protocol $\Pi_{\mathsf{SC\text{-}AKE}}$.

Theorem 3.3.2 (Correctness of $\Pi_{\mathsf{SC\text{-}AKE}}$). *Assume* KEM *and* wKEM *are* $(1-\delta_{\mathsf{KEM}})$*-correct and* SIG *is* $(1-\delta_{\mathsf{SIG}})$*-correct. Then,* $\Pi_{\mathsf{SC\text{-}AKE}}$ *is* $(1-\mu\ell(\delta_{\mathsf{SIG}}+2\delta_{\mathsf{KEM}})/2)$*-correct.*

Proof. It is clear that an initiator oracle and a responder oracle become partners when they execute the protocol faithfully. Moreover, if no correctness error occurs in the underlying KEM and signature scheme, the partner oracles compute an identical session key. Since each oracle is assigned to uniform randomness, the probability that a correctness error occurs in one of the underlying schemes is bounded by $\delta_{\mathsf{SIG}} + 2\delta_{\mathsf{KEM}}$. Since there are at most $\mu\ell/2$ responder oracles, the AKE protocol is correct except with probability $\mu\ell(\delta_{\mathsf{SIG}} + 2\delta_{\mathsf{KEM}})/2$. □

We then prove the security of $\Pi_{\mathsf{SC\text{-}AKE}}$.

Theorem 3.3.3 (Security of $\Pi_{\mathsf{SC\text{-}AKE}}$). *For any* QPT *adversaries* $\mathcal{A}$ *that plays the game* $\mathsf{Game}^{\mathsf{AKE\text{-}FS}}_{\Pi_{\mathsf{SC\text{-}AKE}}}(\mu,\ell)$ *with* μ *parties that establishes at most* ℓ *sessions per party, there exist* QPT *algorithms* $\mathcal{B}_1$ *breaking the* IND-CPA *security of* wKEM, $\mathcal{B}_2$ *breaking the* IND-CCA *security of* KEM, $\mathcal{B}_3$ *breaking the* EUF-CMA *security of* SIG, *and* $\mathcal{D}_1$ *and* $\mathcal{D}_2$ *breaking the security of PRF* F *such that*

$$\mathsf{Adv}^{\mathsf{AKE\text{-}FS}}_{\Pi_{\mathsf{SC\text{-}AKE}},\mathcal{A}}(\kappa) \le \max\left\{\begin{array}{l} \mu^2\ell^2 \cdot (2\mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{wKEM},\mathcal{B}_1}(\kappa) + \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F},\mathcal{D}_1}(\kappa) + \varepsilon_{\mathsf{Ext}}), \\ \mu^2\ell \cdot (2\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{KEM},\mathcal{B}_2}(\kappa) + \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F},\mathcal{D}_2}(\kappa) + \varepsilon_{\mathsf{Ext}}) + \mu\ell^2 \cdot \left(\frac{1}{2^{2\chi_{\mathsf{KEM}}}} + \frac{1}{2^{v_{\mathsf{KEM}}}}\right), \\ \mu \cdot \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{SIG},\mathcal{B}_3}(\kappa), \end{array}\right\} + \frac{\mu\ell}{2} \cdot (\delta_{\mathsf{SIG}} + 2\delta_{\mathsf{KEM}})$$

TABLE 3.1: The strategy taken by the adversary in the security game when the tested oracle is fresh. "Yes" means the tested oracle has some (possibly non-unique) partner oracles and "No" means it has none. "✔✔" means the adversary can reveal the secret-key/session-state at any time, "✔" means the adversary can reveal the secret-key after the tested oracle completes its execution, "✘" means the adversary does not reveal secret-key/session-state. "-" means the session-state is not defined.

Strategy	Role of tested oracle	Partner oracle	$\mathsf{lsk}_{\mathtt{init}}$	$\mathsf{state}_{\mathtt{init}}$	$\mathsf{lsk}_{\mathtt{resp}}$	$\mathsf{state}_{\mathtt{resp}}$
Type-1	`init` or `resp`	Yes	✔✔	✘	✔✔	✘
Type-2	`init` or `resp`	Yes	✔✔	✘	✘	✔✔
Type-3	`init` or `resp`	Yes	✘	✔✔	✔✔	✘
Type-4	`init` or `resp`	Yes	✘	✔✔	✘	✔✔
Type-5	`init`	No	✔✔	✘	✔	-
Type-6	`init`	No	✘	✔✔	✔	-
Type-7	`resp`	No	✔	-	✔✔	✘
Type-8	`resp`	No	✔	-	✘	✔✔

where ν_{KEM} *(resp.* χ_{KEM}*) is the encapsulation key (resp. ciphertext) min-entropy of* wKEM *and* KEM*. The running time of* $\mathcal{B}_1$*,* $\mathcal{B}_2$*,* $\mathcal{B}_3$*,* $\mathcal{D}_1$*, and* $\mathcal{D}_2$ *are about that of* $\mathcal{A}$*.*

We first provide an overview of the proof.

Proof sketch. Let $\mathcal{A}$ be an adversary that plays the security game $\mathsf{Game}^{\mathsf{AKE\text{-}FS}}_{\Pi_{\mathsf{SC\text{-}AKE}}}(\mu, \ell)$. We distinguish between all possible strategies that can be taken by $\mathcal{A}$. Specifically, $\mathcal{A}$'s strategy can be divided into the eight types of strategies listed in Table 3.1. Here, each strategy is mutually independent and covers all possible (non-trivial) strategies. We point out that for our specific AKE construction we have $\mathsf{state}_{\mathtt{resp}} := \perp$ since the responder does not maintain any states (see Remark 3.3.1). Therefore, the Type-1 (resp. Type-3, Type-7) strategy is strictly stronger than the Type-2 (resp. Type-4, Type-8) strategy. Concretely, for our proof, we only need to consider the following four cases and to show that $\mathcal{A}$ has no advantage in each case: (a) $\mathcal{A}$ uses the Type-1 strategy; (b) $\mathcal{A}$ uses the Type-3 strategy; (c) $\mathcal{A}$ uses the Type-5 or Type-6 strategy; (d) $\mathcal{A}$ uses the Type-7 strategy.

In cases (a) and (b), the session key is informally protected by the security properties of KEM, PRF, and randomness extractor Ext. In case (a), since the ephemeral decapsulation key dk_T is not revealed, K_T is indistinguishable from a random key due to the IND-CPA security of wKEM. On the other hand, in case (b), since the initiator's decapsulation key $\mathsf{dk}_{\mathtt{init}}$ is not revealed, K is indistinguishable from a random key due to the IND-CCA security of KEM. Here, we require IND-CCA security because there are initiator oracles other than the tested oracle that uses $\mathsf{dk}_{\mathtt{init}}$, which the reduction algorithm needs to simulate. This is in contrast to case (a) where dk_T is only used by the tested oracle. Then, in both cases, since either K_T or K has sufficient high min-entropy from the view of the adversary, Ext on input K_T or K outputs a uniformly random PRF key. Finally, we can invoke the pseudo-randomness of the PRF and argue that the session key in the tested oracle is indistinguishable from a random key.

In cases (c) and (d), the session key is informally protected by the security property of the signature scheme. More concretely, in both cases, the peer's signing key sk of the tested oracle is not revealed when the tested oracle runs the protocol. Thus, due to the EUF-CMA security of SIG, $\mathcal{A}$ cannot forge the signature for the session identifier of the tested oracle $\mathsf{sid}_{\mathsf{test}}$ (in case (c)) or for the ephemeral encapsulation key ek_T (in case (d)). In addition, since the tested oracle has no partner oracles, no oracle ever signs $\mathsf{sid}_{\mathsf{test}}$ (in case (c)) or ek_T (in case (d)). Therefore, combining these two facts, we conclude that the tested oracle cannot be

in the accept state unless $\mathcal{A}$ breaks the signature scheme. In other words, when $\mathcal{A}$ issues Test-query, the tested oracle always returns $\perp$ even if $\mathcal{A}$ corrupts the long-term key of the peer. Thus, the session key of the tested oracle is hidden from $\mathcal{A}$. □

The full proof of Theorem 3.3.3 is as follows.

Proof of Theorem 3.3.3. Let $\mathcal{A}$ be an adversary that plays the security game $\mathsf{Game}^{\mathsf{AKE\text{-}FS}}_{\Pi_{\mathsf{SC\text{-}AKE}}}(\mu, \ell)$ with the challenger $\mathcal{C}$ with advantage $\mathsf{Adv}^{\mathsf{AKE\text{-}FS}}_{\Pi_{\mathsf{SC\text{-}AKE}},\mathcal{A}}(\kappa) = \epsilon$. In order to prove Theorem 3.3.3, we distinguish between the strategy that can be taken by the $\mathcal{A}$. Specifically, $\mathcal{A}$'s strategy can be divided into the eight types of strategies listed in Table 3.1. Here, each strategy is mutually independent and covers all possible (non-trivial) strategies.[15] We point out that for our specific AKE construction we have $\mathsf{state}_{\mathtt{resp}} := \perp$ since the responder does not maintain any states (see Remark 3.3.1). Therefore, the Type-1 (resp. Type-3, Type-7) strategy is strictly stronger than the Type-2 (resp. Type-4, Type-8) strategy. We only include the full types of strategies in Table 3.1 as we believe it would be helpful when proving other AKE protocols and note that our proof implicitly handles both strategies at the same time.

For each possible strategy taken by $\mathcal{A}$, we construct an algorithm that breaks one of the underlying assumptions by using such an adversary $\mathcal{A}$ as a subroutine. More formally, we construct six algorithms $\mathcal{B}_1$, $\mathcal{B}_2$, $\mathcal{B}_{3,0}$, $\mathcal{B}_{3,1}$, $\mathcal{D}_1$ and $\mathcal{D}_2$ satisfying the following:

1. If $\mathcal{A}$ uses the Type-1 (or Type-2) strategy, then $\mathcal{B}_1$ succeeds in breaking the IND-CPA security of wKEM with advantage $\approx \frac{1}{\mu^2\ell^2}\epsilon$ or $\mathcal{D}_1$ succeeds in breaking the security of PRF F with advantage $\approx \frac{1}{\mu^2\ell^2}\epsilon$.
2. If $\mathcal{A}$ uses the Type-3 (or Type-4) strategy, then $\mathcal{B}_2$ succeeds in breaking the IND-CCA security of KEM with advantage $\approx \frac{1}{\mu^2\ell}\epsilon$ or $\mathcal{D}_2$ succeeds in breaking the security of PRF F with advantage $\approx \frac{1}{\mu^2\ell}\epsilon$.
3. If $\mathcal{A}$ uses the Type-5 or Type-6 strategy, then $\mathcal{B}_{3,0}$ succeeds in breaking the EUF-CMA security of SIG with advantage $\approx \frac{1}{\mu}\epsilon$.
4. If $\mathcal{A}$ uses the Type-7 (or Type-8) strategy, then $\mathcal{B}_{3,1}$ succeeds in breaking the EUF-CMA security of SIG with advantage $\approx \frac{1}{\mu}\epsilon$.

We present a security proof structured as a sequence of games. Without loss of generality, we assume that $\mathcal{A}$ always issues a Test-query. In the following, let S_j denote the event that $b = b'$ occurs in Game j and let $\epsilon_j := \left|\Pr\left[\mathsf{S}_j\right] - 1/2\right|$ denote the advantage of the adversary in Game j. Regardless of the strategy taken by $\mathcal{A}$, all proofs share the common game sequences Game 0 and Game 1 as described below.

Game 0**.** This game is identical to the original security game. We thus have

$$\epsilon_0 = \epsilon.$$

Game 1**.** This game is identical to Game 0, except that we add an abort condition. Let $\mathsf{E}_{\mathsf{corr}}$ be the event that there exist two partner oracles π_i^s and π_j^t that do not agree on the same session key. If $\mathsf{E}_{\mathsf{corr}}$ occurs, then $\mathcal{C}$ aborts (i.e., sets $\mathcal{A}$'s output to be a random bit) at the end of the game.

There are at most $\mu\ell/2$ responder oracles and each oracle is assigned uniform randomness. From Theorem 3.3.2, the probability of error occurring during the security game is at most $\mu\ell(\delta_{\mathsf{SIG}} + 2\delta_{\mathsf{KEM}})/2$.

[15] We note that although we can consider an adversary $\mathcal{A}$ that makes no reveal queries (i.e., all lsk and state are either p or "-"), we can exclude them without loss of generality since such $\mathcal{A}$ can always be modified into an adversary $\mathcal{A}'$ that follows one of the strategies listed in Table 3.1.

Therefore, $\mathsf{E}_{\mathsf{corr}}$ occurs with probability at most $\mu\ell(\delta_{\mathsf{SIG}} + 2\delta_{\mathsf{KEM}})/2$. We thus have

$$|\Pr[\mathsf{S}_0] - \Pr[\mathsf{S}_1]| \leq \frac{\mu\ell}{2} \cdot (\delta_{\mathsf{SIG}} + 2\delta_{\mathsf{KEM}}).$$

In the following games we assume no decryption error or signature verification error occurs.

We now divide the game sequence depending on the strategy taken by the adversary $\mathcal{A}$. Regardless of $\mathcal{A}$'s strategy, we prove that ϵ_1 is negligible, which in particular implies that ϵ is also negligible. Formally, this is shown in Lemmata 3.3.4 to 3.3.7 provided in their respective subsections below. We first complete the proof of the theorem. Specifically, by combining all the lemmata together and folding adversaries $\mathcal{B}_{3,0}$ and $\mathcal{B}_{3,1}$ into one adversary $\mathcal{B}_3$, we obtain the following desired bound:

$$\begin{aligned}\mathsf{Adv}^{\mathsf{AKE\text{-}FS}}_{\Pi_{\mathsf{SC\text{-}AKE}},\mathcal{A}}(\kappa) \leq \max \left\{ \begin{array}{l} \mu^2\ell^2 \cdot (2\mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{wKEM},\mathcal{B}_1}(\kappa) + \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F},\mathcal{D}_1}(\kappa) + \varepsilon_{\mathsf{Ext}}), \\ \mu^2\ell \cdot (2\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{KEM},\mathcal{B}_2}(\kappa) + \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F},\mathcal{D}_2}(\kappa) + \varepsilon_{\mathsf{Ext}}) + \mu\ell^2 \cdot \left(\frac{1}{2^{2\chi_{\mathsf{KEM}}}} + \frac{1}{2^{\gamma_{\mathsf{KEM}}}}\right), \\ \mu \cdot \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{SIG},\mathcal{B}_3}(\kappa), \end{array} \right\} \\ + \frac{\mu\ell}{2} \cdot (\delta_{\mathsf{SIG}} + 2\delta_{\mathsf{KEM}})\end{aligned}$$

Here, the running time of the algorithms $\mathcal{B}_1$, $\mathcal{B}_2$, $\mathcal{B}_3$, $\mathcal{D}_1$ and $\mathcal{D}_2$ consist essentially the time required to simulate the security game for $\mathcal{A}$ once, plus a minor number of additional operations. □

It remains to prove Lemmata 3.3.4 to 3.3.7.

Proof of Lemma 3.3.4: Against Type-1 or Type-2 Adversary.

Lemma 3.3.4. *For any* QPT *adversaries $\mathcal{A}$ using the Type-1 or Type-2 strategy, there exist* QPT *algorithms $\mathcal{B}_1$ breaking the* IND-CPA *security of* wKEM *and $\mathcal{D}_1$ breaking the security of PRF* F *such that*

$$\epsilon_1 \leq \mu^2\ell^2 \cdot \left(2\mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{wKEM},\mathcal{B}_1}(\kappa) + \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F},\mathcal{D}_1}(\kappa) + \varepsilon_{\mathsf{Ext}}\right).$$

Proof of Lemma 3.3.4. We present the rest of the sequence of games from Game 1.

Game 2. In this game, at the beginning of the game, $\mathcal{C}$ chooses an initiator oracle $\pi_i^{\hat{s}}$ and a responder oracle $\pi_j^{\hat{t}}$ uniformly at random from the $\mu\ell$ oracles. Let $\mathsf{E}_{\mathsf{testO}}$ be the event that the tested oracle is neither $\pi_i^{\hat{s}}$ nor $\pi_j^{\hat{t}}$, or $\pi_i^{\hat{s}}$ and $\pi_j^{\hat{t}}$ are not partner. Since $\mathsf{E}_{\mathsf{testO}}$ is an efficiently checkable event, $\mathcal{C}$ aborts as soon as it detects that event $\mathsf{E}_{\mathsf{testO}}$ occurs.[16] $\mathcal{C}$ guesses the choice made by $\mathcal{A}$ correctly with probability at least $1/\mu^2\ell^2$, so we have

$$\epsilon_2 \geq \frac{1}{\mu^2\ell^2}\epsilon_1.$$

Game 3. In this game, we modify the way the initiator oracle $\pi_i^{\hat{s}}$ responds on its second invocation. In particular, when $\pi_i^{\hat{s}}$ is invoked (on the second time) on input $(\mathsf{C}, \mathsf{C}_T, \mathsf{c})$, it proceeds as in the previous game except that it uses the key K_T that was generated by the responder oracle $\pi_j^{\hat{t}}$ rather than using the key obtained through decrypting C_T. Here, conditioned on $\mathsf{E}_{\mathsf{testO}}$ not occurring, we are guaranteed that the responder oracle $\pi_j^{\hat{t}}$ generated C_T by running $(\mathsf{K}_T, \mathsf{C}_T) \leftarrow \mathsf{wKEM.Encap}(\mathsf{ek}_T)$, where ek_T is the

[16] For example, $\mathcal{C}$ can efficiently notice if the two oracles $\pi_i^{\hat{s}}$ and $\pi_j^{\hat{t}}$ become non-partners even before $\mathcal{A}$ makes a `Test`-query by checking the input-output of each oracles.

encapsulation key that $\pi_i^{\hat{s}}$ outputs on the first invocation. This is because otherwise, the oracles $\pi_i^{\hat{s}}$ and $\pi_j^{\hat{t}}$ will not be partner oracles. Conditioning on event $\mathsf{E}_{\mathsf{corr}}$ (i.e., decryption failure) not occurring, the two games Game 2 and Game 3 are identical. Hence,

$$\epsilon_3 = \epsilon_2.$$

Game 4. In this game, we modify the way the responder oracle $\pi_j^{\hat{t}}$ responds. When the responder oracle $\pi_j^{\hat{t}}$ is invoked on input ek_T, the game samples a random key $\mathsf{K}_T \leftarrow\!\$\, \mathcal{KS}_{\mathsf{wKEM}}$ instead of computing $(\mathsf{K}_T, \mathsf{C}_T) \leftarrow \mathsf{wKEM.Encap}(\mathsf{ek}_T)$. Note that when the initiator oracle $\pi_i^{\hat{s}}$ is invoked (on the second time) on input $(\mathsf{C}, \mathsf{C}_T, \mathsf{c})$, it uses this random key K_T. We claim Game 3 and Game 4 are indistinguishable assuming the IND-CPA security of wKEM. To prove this, we construct an algorithm $\mathcal{B}_1$ breaking the IND-CPA security as follows.

$\mathcal{B}_1$ receives a public parameter $\mathsf{pp}_{\mathsf{wKEM}}$, a public key ek^*, and a challenge $(\mathsf{K}^*, \mathsf{C}^*)$ from its challenger. $\mathcal{B}_1$ sets up the public parameter of $\Pi_{\mathsf{SC\text{-}AKE}}$ using $\mathsf{pp}_{\mathsf{wKEM}}$ and computes $(\mathsf{lpk}_i, \mathsf{lsk}_i)$ for all $i \in [\mu]$ by running the protocol honestly, and samples $(\hat{\imath}, \hat{\jmath}, \hat{s}, \hat{t})$ uniformly random from $[\mu]^2 \times [\ell]^2$. It then invokes $\mathcal{A}$ on the public parameter of $\Pi_{\mathsf{SC\text{-}AKE}}$ and $\{\, \mathsf{lpk}_i \mid i \in [\mu] \,\}$ and answers queries made by $\mathcal{A}$ as follows:

- $\mathtt{Send}(i, s, \langle \mathtt{START} : \mathsf{role}, j \rangle)$: If $(i, s, j) = (\hat{\imath}, \hat{s}, \hat{\jmath})$, then $\mathcal{B}_1$ returns ek^* to $\mathcal{A}$ and implicitly sets $\mathsf{state}_i^s := \mathsf{dk}^*$. Otherwise, $\mathcal{B}_1$ responds as in Game 4.
- $\mathtt{Send}(j, t, m = (\mathsf{ek}_T, \sigma_i))$: Let $i := \mathsf{Pid}_j^t$. Depending on the values of (j, t, i), it performs the following:
  - If $(j, t) = (\hat{\jmath}, \hat{t})$ and $i \neq \hat{\imath}$, then $\pi_i^{\hat{s}}$ and $\pi_j^{\hat{t}}$ cannot be partner oracles. Therefore, since event $\mathsf{E}_{\mathsf{testO}}$ is triggered $\mathcal{B}_1$ aborts.
  - If $(j, t, i) = (\hat{\jmath}, \hat{t}, \hat{\imath})$, then $\mathcal{B}_1$ checks if $\mathsf{ek}_T = \mathsf{ek}^*$. If not, event $\mathsf{E}_{\mathsf{testO}}$ is triggered so it aborts. Otherwise, it proceeds as in Game 4 except that it sets $\mathsf{K}_T = \mathsf{K}^*$ and $\mathsf{C}_T = \mathsf{C}^*$ rather than sampling them on its own. It then returns the message $(\mathsf{C}, \mathsf{C}_T, \mathsf{c})$.
  - If $(j, t, i) \neq (\hat{\jmath}, \hat{t}, \hat{\imath})$, then $\mathcal{B}_1$ responds as in Game 4.
- $\mathtt{Send}(i, s, m = (\mathsf{C}, \mathsf{C}_T, \mathsf{c}))$: Let $j := \mathsf{Pid}_i^s$. Depending on the values of (i, s, j), it performs the following:
  - If $(i, s) = (\hat{\imath}, \hat{s})$ and $j \neq \hat{\jmath}$, then $\pi_i^{\hat{s}}$ and $\pi_j^{\hat{t}}$ cannot be partner oracles. Therefore, since event $\mathsf{E}_{\mathsf{testO}}$ is triggered $\mathcal{B}_1$ aborts.
  - If $(i, s, j) = (\hat{\imath}, \hat{s}, \hat{\jmath})$, then $\mathcal{B}_1$ checks if $\mathsf{C}_T = \mathsf{C}^*$. If not, event $\mathsf{E}_{\mathsf{testO}}$ is triggered so it aborts. Otherwise, it responds as in Game 4.
  - If $(i, s, j) \neq (\hat{\imath}, \hat{s}, \hat{\jmath})$, then $\mathcal{B}_1$ responds as in Game 4.
- $\mathtt{RevLTK}(i)$, $\mathtt{RegisterLTK}(i)$, $\mathtt{RevState}(i, s)$, $\mathtt{RevSessKey}(i, s)$: $\mathcal{B}_1$ proceeds as in the previous game. Here, note that since $\mathcal{A}$ follows the Type-1 or Type-2 strategy, $\mathcal{B}_1$ can answer all the $\mathtt{RevState}$-query. Namely, $\mathcal{A}$ never queries $\mathtt{RevState}(\hat{\imath}, \hat{s})$ (i.e., $\mathsf{state}_{\hat{\imath}}^{\hat{s}} := \mathsf{dk}^*$) conditioning on $\mathsf{E}_{\mathsf{testO}}$ not occurring, which is the only query that $\mathcal{B}_1$ cannot answer.
- $\mathtt{Test}(i, s)$: $\mathcal{B}_1$ responds as in Game 4. Here, in case $(i, s) \notin \{\, (\hat{\imath}, \hat{s}), (\hat{\jmath}, \hat{t}) \,\}$, then event $\mathsf{E}_{\mathsf{testO}}$ is triggered so it aborts.

Finally, if $\mathcal{A}$ outputs a guess b', $\mathcal{B}_1$ outputs 0 if $b' = b$ 1 otherwise. It can be checked that $\mathcal{B}_1$ perfectly simulates Game 3 (resp. Game 4) to $\mathcal{A}$ when the challenge K^* is the real key (resp. a random key). Thus we have

$$|\Pr[\mathsf{S}_3] - \Pr[\mathsf{S}_4]| \leq 2\mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{wKEM},\mathcal{B}_1}(\kappa).$$

Game 5. In this game, we modify how the PRF key K_2 is generated by the tested oracle and its partner oracle. Instead of computing $\mathsf{K}_2 \leftarrow \mathsf{Ext}_s(\mathsf{K}_T)$, both oracles use the same randomly sampled $\mathsf{K}_2 \leftarrow\!\!\$\, \mathcal{K}$. Due to the modification we made in the previous game, K_T is chosen uniformly at random from $\mathcal{KS}_{\mathsf{wKEM}}$ so K_T has $\log_2(|\mathcal{KS}_{\mathsf{wKEM}}|) \geq \gamma_{\mathsf{KEM}}$ min-entropy. Then, by the definition of the strong $(\gamma_{\mathsf{KEM}}, \varepsilon_{\mathsf{Ext}})$-extractor Ext, we have

$$|\Pr[\mathsf{S}_4] - \Pr[\mathsf{S}_5]| \leq \varepsilon_{\mathsf{Ext}}.$$

Game 6. In this game, we modify how the session key k is generated by the tested oracle. Instead of computing $\mathsf{k}\|\tilde{\mathsf{k}} \leftarrow \mathsf{F}_{\mathsf{K}_1}(\mathsf{sid}) \oplus \mathsf{F}_{\mathsf{K}_2}(\mathsf{sid})$, the tested oracle (which is either π_i^s or π_j^t conditioned on event $\mathsf{E}_{\mathsf{testO}}$ not occurring) computes the session key as $\mathsf{k}\|\tilde{\mathsf{k}} \leftarrow \mathsf{F}_{\mathsf{K}_1}(\mathsf{sid}) \oplus x$, where x is chosen uniformly at random from $\{0,1\}^{\kappa+d}$. Since K_2 is chosen uniformly and hidden from the views of the adversary $\mathcal{A}$, games Game 5 and Game 6 are indistinguishable by the security of the PRF.[17] In particular, we can construct a PRF adversary $\mathcal{D}_1$ that uses $\mathcal{A}$ as a subroutine such that

$$|\Pr[\mathsf{S}_5] - \Pr[\mathsf{S}_6]| \leq \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F},\mathcal{D}_1}(\kappa).$$

In Game 6, the session key in the tested oracle is chosen uniformly at random. Thus, even an unbounded adversary $\mathcal{A}$ cannot have distinguishing advantages. Therefore, $\Pr[\mathsf{S}_6] = 1/2$. Combining everything together, we have

$$\epsilon_1 \leq \mu^2\ell^2 \cdot \left(2\mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{wKEM},\mathcal{B}_1}(\kappa) + \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F},\mathcal{D}_1}(\kappa) + \varepsilon_{\mathsf{Ext}}\right).$$

□

Proof of Lemma 3.3.5: Against Type-3 or Type-4 Adversary.

Lemma 3.3.5. *For any* QPT *adversary $\mathcal{A}$ using the Type-3 or Type-4 strategy, there exist* QPT *algorithms $\mathcal{B}_2$ breaking the* IND-CCA *security of* KEM *and $\mathcal{D}_2$ breaking the security of PRF* F *such that*

$$\epsilon_1 \leq \mu^2\ell \cdot \left(2\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{KEM},\mathcal{B}_2}(\kappa) + \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F},\mathcal{D}_2}(\kappa) + \varepsilon_{\mathsf{Ext}}\right) + \mu\ell^2 \cdot \left(\frac{1}{2^{2\chi_{\mathsf{KEM}}}} + \frac{1}{2^{\nu_{\mathsf{KEM}}}}\right).$$

Proof of Lemma 3.3.5. We present the rest of the sequence of games from Game 1.

Game 2. This game is identical to Game 1, except that we add another abort condition. Let $\mathsf{E}_{\mathsf{uniq}}$ be the event that there exists an oracle that has more than one partner oracle. If $\mathsf{E}_{\mathsf{uniq}}$ occurs, then $\mathcal{C}$ aborts. Since Game 1 and Game 2 proceed identically unless $\mathsf{E}_{\mathsf{uniq}}$ occurs, we have

$$|\epsilon_1 - \epsilon_2| \leq \Pr[\mathsf{E}_{\mathsf{uniq}}].$$

We claim

$$\Pr[\mathsf{E}_{\mathsf{uniq}}] \leq \mu\ell^2 \cdot \left(\frac{1}{2^{2\chi_{\mathsf{KEM}}}} + \frac{1}{2^{\nu_{\mathsf{KEM}}}}\right).$$

Fix $j \in [\mu]$ and consider the set of oracles $S_j = \{\, \pi_i^s \mid \mathsf{Pid}_i^s = j \,\}$. For any $\pi_i^s \in S_j$, if there exist two oracles π_j^t and $\pi_j^{t'}$ with $t \neq t' \in [\ell]$ that are partners of π_i^s, then $\mathsf{sid}_i^s = \mathsf{sid}_j^t = \mathsf{sid}_j^{t'}$ holds. We distinguish between the following cases.

[17] We note that for Lemma 3.3.4 we do not require the full power of the PRF; a pseudorandom *generator* (PRG) would have sufficed since the key K_2 is used nowhere else in the game.

Case 1. We first consider the case π_i^s is an initiator and π_j^t and $\pi_j^{t'}$ are responders. Let ek_T be the ephemeral encapsulation key generated by π_i^s. In this case, $\mathsf{E}_{\mathsf{uniq}}$ occurs if the responder oracles π_j^t and $\pi_j^{t'}$ generate the same ciphertext with respect to ek_i and ek_T. Since ek_i and ek_T are independently and honestly generated by the game and each responder oracle is assigned uniform randomness, the probability of a ciphertext collision is upper bounded by $\ell^2/2^{2\chi_{\mathsf{KEM}}}$, where recall χ_{KEM} is the ciphertext min-entropy of wKEM and KEM. Taking the union bound over all $j \in [\mu]$, we conclude that Case 1 occurs with probability at most $\mu\ell^2/2^{2\chi_{\mathsf{KEM}}}$.

Case 2. We next consider the case π_i^s is a responder and π_j^t and $\pi_j^{t'}$ are initiators. In this case, $\mathsf{E}_{\mathsf{uniq}}$ occurs if the initiator oracles π_j^t and $\pi_j^{t'}$ generate the same ephemeral encapsulation key. Since each initiator oracle samples an encapsulation key independently, the probability of an encapsulation key collision is upper bounded by $\ell^2/2^{\nu_{\mathsf{KEM}}}$, where recall ν_{KEM} is the encapsulation key min-entropy of wKEM. Taking the union bound over all $j \in [\mu]$, we conclude that Case 2 occurs with probability at most $\mu\ell^2/2^{\nu_{\mathsf{KEM}}}$.

The claim can be shown by combining the two probabilities from Case 1 and Case 2. In the following games, we assume every oracle has a unique partner oracle if it exists.

Game 3. In this game, at the beginning of the game, $\mathcal{C}$ chooses a random party $P_{\hat{\imath}}$ from the μ parties and a random responder oracle $\pi_{\hat{\jmath}}^{\hat{t}}$ from the $\mu\ell$ oracles. Let $\mathsf{E}_{\mathsf{testO}}$ be the event where $\neg\mathsf{E}_{\mathsf{testO}}$ denotes the event that either the tested oracle is $\pi_{\hat{\imath}}^{\hat{s}}$ for some $s \in [\ell]$ and its partner oracle is $\pi_{\hat{\jmath}}^{\hat{t}}$, or the tested oracle is $\pi_{\hat{\jmath}}^{\hat{t}}$ and its peer is $P_{\hat{\imath}}$. Since $\mathsf{E}_{\mathsf{testO}}$ is an efficiently checkable event, $\mathcal{C}$ aborts as soon as it detects that event $\mathsf{E}_{\mathsf{testO}}$ occurs. $\mathcal{C}$ guesses the choice made by $\mathcal{A}$ correctly with probability $1/\mu^2\ell$, so we have

$$\epsilon_3 = \frac{1}{\mu^2\ell}\epsilon_2.$$

Game 4. In this game, we modify the way the initiator oracle $\pi_{\hat{\imath}}^s$ for any $s \in [\ell]$ responds on its second invocation. Let (K, C) be the KEM key-ciphertext pair generated by oracle $\pi_{\hat{\jmath}}^{\hat{t}}$. Then, when $\pi_{\hat{\imath}}^s$ is invoked (on the second time) on input $(\mathsf{C}', \mathsf{C}_T, \mathsf{c})$, it first checks if $\mathsf{C}' = \mathsf{C}$. If so, it proceeds as in the previous game except that it uses the key K that was generated by $\pi_{\hat{\jmath}}^{\hat{t}}$ rather than using the key obtained through decrypting C'. Otherwise, if $\mathsf{C}' \neq \mathsf{C}$, then it proceeds exactly as in the previous game. Conditioning on event $\mathsf{E}_{\mathsf{corr}}$ (i.e., decryption failure) not occurring, the two games Game 3 and Game 4 are identical. Hence,

$$\epsilon_4 = \epsilon_3.$$

Game 5. In this game, we modify the way the responder oracle $\pi_{\hat{\jmath}}^{\hat{t}}$ responds. When the responder oracle $\pi_{\hat{\jmath}}^{\hat{t}}$ is invoked on input ek_T, it samples a random key $\mathsf{K} \leftarrow\!\$\, \mathcal{KS}_{\mathsf{KEM}}$ instead of computing $(\mathsf{K}, \mathsf{C}) \leftarrow \mathsf{KEM.Encap}(\mathsf{ek}_{\hat{\imath}})$. Note that due to the modification we made in the previous game, when the initiator oracle $\pi_{\hat{\imath}}^s$ for any $s \in [\ell]$ is invoked (on the second time) on input $(\mathsf{C}', \mathsf{C}_T, \mathsf{c})$ for $\mathsf{C}' = \mathsf{C}$, it uses the random key K generated by oracle $\pi_{\hat{\jmath}}^{\hat{t}}$. We claim Game 4 and Game 5 are indistinguishable assuming the IND-CCA security of KEM. To prove this, we construct an algorithm $\mathcal{B}_2$ breaking the IND-CCA security as follows.

$\mathcal{B}_2$ receives a public parameter $\mathsf{pp}_{\mathsf{KEM}}$, a public key ek^*, and a challenge $(\mathsf{K}^*, \mathsf{C}^*)$ from its challenger. $\mathcal{B}_2$ then samples a random $(\hat{\imath}, \hat{\jmath}, \hat{t}) \leftarrow\!\$\, [\mu]^2 \times [\ell]$, sets up the public parameter of $\Pi_{\mathsf{SC\text{-}AKE}}$ using $\mathsf{pp}_{\mathsf{KEM}}$, and generates the long-term key pairs as follows. For party $P_{\hat{\imath}}$, $\mathcal{B}_2$ runs $(\mathsf{vk}_{\hat{\imath}}, \mathsf{sk}_{\hat{\imath}}) \leftarrow \mathsf{SIG.KeyGen}(\mathsf{pp}_{\mathsf{SIG}})$ and sets the long-term public key as $\mathsf{lpk}_{\hat{\imath}} := (\mathsf{ek}^*, \mathsf{vk}_{\hat{\imath}})$ and implicitly sets the long-term secret key as $\mathsf{lsk}_{\hat{\imath}} := (\mathsf{dk}^*, \mathsf{sk}_{\hat{\imath}})$, where note that $\mathcal{B}_2$ does not know dk^*. For all the other parties $i \in [\mu \backslash \hat{\imath}]$, $\mathcal{B}_2$ computes the long-term

key pairs $(\mathsf{lpk}_i, \mathsf{lsk}_i)$ as in Game 5. Finally, $\mathcal{B}_2$ invokes $\mathcal{A}$ on input the public parameter of $\Pi_{\mathsf{SC\text{-}AKE}}$ and $\{\, \mathsf{lpk}_i \mid i \in [\mu] \,\}$ and answers the queries made by $\mathcal{A}$ as follows:

- $\mathtt{Send}(i, s, \langle \mathtt{START} : \mathsf{role}, j \rangle)$: $\mathcal{B}_2$ responds as in Game 5.
- $\mathtt{Send}(j, t, m = (\mathsf{ek}_T, \sigma_i))$: Let $i := \mathsf{Pid}_j^t$. Depending on the values of (j, t, i), it performs the following:
  - If $(j, t, i) = (\hat{\jmath}, \hat{t}, \hat{\imath})$, then $\mathcal{B}_2$ responds as in Game 5 except that it sets $(\mathsf{K}, \mathsf{C}) := (\mathsf{K}^*, \mathsf{C}^*)$ rather than generating them on its own. It then returns the message $(\mathsf{C}^*, \mathsf{C}_T, \mathsf{c})$.
  - If $(j, t, i) \neq (\hat{\jmath}, \hat{t}, \hat{\imath})$, then $\mathcal{B}_2$ responds as in Game 5.
- $\mathtt{Send}(i, s, m = (\mathsf{C}, \mathsf{C}_T, \mathsf{c}))$: Depending on the value of i, it performs the following:
  - If $i = \hat{\imath}$, then $\mathcal{B}_2$ checks if $\mathsf{C} = \mathsf{C}^*$. If so, it responds as in Game 5 except that it sets $\mathsf{K} := \mathsf{K}^*$. Otherwise, if $\mathsf{C} \neq \mathsf{C}^*$, then it queries the decapsulation oracle on C and receives back K'. $\mathcal{B}_2$ then responds as in Game 5 except that it sets $\mathsf{K} := \mathsf{K}'$.
  - If $i \neq \hat{\imath}$, then $\mathcal{B}_2$ responds as in Game 5.
- $\mathtt{RevLTK}(i)$, $\mathtt{RegisterLTK}(i)$, $\mathtt{RevState}(i, s)$, $\mathtt{RevSessKey}(i, s)$: $\mathcal{B}_2$ responds as in Game 5. Here, note that since $\mathcal{A}$ follows the Type-3 or Type-4 strategy, $\mathcal{B}_2$ can answer all the $\mathtt{RevLTK}$-query. Namely, $\mathcal{A}$ never queries $\mathtt{RevLTK}(\hat{\imath})$ (i.e., $\mathsf{lsk}_{\hat{\imath}} := (\mathsf{dk}^*, \mathsf{sk}_{\hat{\imath}})$) conditioning on $\mathsf{E}_{\mathsf{testO}}$ not occurring, which is the only query that $\mathcal{B}_2$ cannot answer.
- $\mathtt{Test}(i, s)$: $\mathcal{B}_2$ responds to the query as the definition. Here, in case $i \neq \hat{\imath}$ or $(i, s) \neq (\hat{\jmath}, \hat{t})$, then event $\mathsf{E}_{\mathsf{testO}}$ is triggered so it aborts.

If $\mathcal{A}$ outputs a guess b', $\mathcal{B}_2$ outputs 0 if $b' = b$ 1 otherwise. It can be checked that $\mathcal{B}_2$ perfectly simulates Game 4 (resp. Game 5) to $\mathcal{A}$ when the challenge K^* is the real key (resp. a random key). Thus we have

$$|\Pr[\mathsf{S}_4] - \Pr[\mathsf{S}_5]| \leq 2\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{KEM},\mathcal{B}_2}(\kappa).$$

Game 6. In this game, whenever we need to derive $\mathsf{K}_1^* \leftarrow \mathsf{Ext}_s(\mathsf{K}^*)$, we instead use a uniformly and randomly chosen PRF key $\mathsf{K}_1^* \leftarrow\!\!\$\, \mathcal{K}$ (fixed once and for all), where K^* is the KEM key chosen by oracle $\pi_{\hat{\jmath}}^{\hat{\imath}}$. Due to the modification we made in the previous game, K^* is chosen uniformly at random from $\mathcal{KS}_{\mathsf{KEM}}$ so K has $\log_2(|\mathcal{KS}_{\mathsf{KEM}}|) \geq \gamma_{\mathsf{KEM}}$ min-entropy. Then, by the definition of the strong $(\gamma_{\mathsf{KEM}}, \varepsilon_{\mathsf{Ext}})$-extractor Ext, we have

$$|\Pr[\mathsf{S}_5] - \Pr[\mathsf{S}_6]| \leq \varepsilon_{\mathsf{Ext}}.$$

Game 7. In this game, we sample a random function RF and whenever we need to compute $\mathsf{F}_{\mathsf{K}_1^*}(\mathsf{sid})$ for any sid, we instead compute $\mathsf{RF}(\mathsf{K}_1^*, \mathsf{sid})$. Due to the modification made in the previous game, K_1^* is sampled uniformly from $\mathcal{K}$. Therefore, the two games can be easily shown to be indistinguishable assuming the pseudo-randomness of the PRF. In particular, we can construct a PRF adversary $\mathcal{D}_2$ such that

$$|\Pr[\mathsf{S}_6] - \Pr[\mathsf{S}_7]| \leq \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F},\mathcal{D}_2}(\kappa).$$

It remains to show that the session key output by the tested oracle in the *game*7 is uniformly random regardless of the challenge bit $b \in \{0, 1\}$ chosen by the game. We consider the case where $b = 0$ and prove that the honestly generated session key by the tested oracle is distributed uniformly random. First conditioning on event $\mathsf{E}_{\mathsf{testO}}$ not occurring, it must be the case that the tested oracle (and its partner oracle)

prepares the session key as $\mathsf{k}^*\|\tilde{\mathsf{k}} \leftarrow \mathsf{RF}(\mathsf{K}_1^*, \mathsf{sid}^*) \oplus \mathsf{F}_{\mathsf{K}_2}(\mathsf{sid}^*)$ for some sid^*. That is, K_1^* sampled by the responder oracle $\pi_{\hat{\jmath}}^{\hat{t}}$ is used to compute the session key. Next, conditioning on event $\mathsf{E}_{\mathsf{uniq}}$ not occurring, the only oracles that share the same sid^* must be the tested oracle and its partner oracle since otherwise it would break the uniqueness of partner oracles. Therefore, we conclude that $\mathsf{RF}(\mathsf{K}_1^*, \mathsf{sid}^*)$ is only used to compute the session key of the tested oracle and its partner oracle. Since the output of RF is distributed uniformly random for different inputs, we conclude that $\Pr[\mathsf{S}_7] = 1/2$. Combining all the arguments together, we obtain

$$\epsilon_1 \leq \mu^2\ell \cdot \left(2\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{KEM},\mathcal{B}_2}(\kappa) + \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F},\mathcal{D}_2}(\kappa) + \varepsilon_{\mathsf{Ext}}\right) + \mu\ell^2 \cdot \left(\frac{1}{2^{2\chi_{\mathsf{KEM}}}} + \frac{1}{2^{\nu_{\mathsf{KEM}}}}\right).$$

□

Proof of Lemma 3.3.6: Against Type-5 or Type-6 Adversary.

Lemma 3.3.6. *For any* QPT *adversary* $\mathcal{A}$ *using the Type-5 or Type-6 strategy, there exists a* QPT *algorithm* $\mathcal{B}_{3,0}$ *breaking the* EUF-CMA *of* SIG *such that*

$$\epsilon_1 \leq \mu \cdot \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{SIG},\mathcal{B}_{3,0}}(\kappa).$$

Proof of Lemma 3.3.6. We present the rest of the sequence of games from Game 1.

Game 2. In this game, at the beginning of the game, $\mathcal{C}$ chooses a party $P_{\hat{\jmath}}$ uniformly at random from the μ parties. Let $\mathsf{E}_{\mathsf{testO}}$ be the event that the peer of the tested oracle is not $P_{\hat{\jmath}}$. If event $\mathsf{E}_{\mathsf{testO}}$ occurs, $\mathcal{C}$ aborts. Since $\mathcal{C}$ guesses the choice made by $\mathcal{A}$ correctly with probability $1/\mu$, we have

$$\epsilon_2 = \frac{1}{\mu}\epsilon_1.$$

Game 3. This game is identical to Game 2, except that we add an abort condition. Let S be a list of message-signature pairs that $P_{\hat{\jmath}}$ generated as being a responder oracle. That is, every time $\pi_{\hat{\jmath}}^t$ for some $t \in [\ell]$ is invoked as a responder, it updates the list S by appending the message-signature pair $(\mathsf{sid}_{\hat{\jmath}}^t, \sigma_{\hat{\jmath}}^t)$ that it generates. Then, when an initiator oracle π_i^s for any $(i,s) \in [\mu] \times [\ell]$ is invoked on input $(\mathsf{C}, \mathsf{C}_T, \mathsf{c})$ from party $P_{\hat{\jmath}}$ (i.e., $\mathsf{Pid}_i^s = \hat{\jmath}$), it first computes sid_i^s and σ as in the previous game, and it checks whether $(\mathsf{sid}_i^s, \sigma) \in S$ when $\mathsf{SIG.Verify}(\mathsf{vk}_{\hat{\jmath}}, \mathsf{sid}_i^s, \sigma) = 1$ and $P_{\hat{\jmath}}$ is not corrupted. If this condition does not hold, the game aborts. Otherwise, it proceeds as in the previous game. We call the event the abort occurs as $\mathsf{E}_{\mathsf{sig}}$. Since the two games are identical until abort, we have

$$|\Pr[\mathsf{S}_2] - \Pr[\mathsf{S}_3]| \leq \Pr[\mathsf{E}_{\mathsf{sig}}].$$

Before, bounding $\Pr[\mathsf{E}_{\mathsf{sig}}]$, we finish the proof of the lemma. We show that no adversary $\mathcal{A}$ following the Type-5 or Type-6 strategy has winning advantage in Game 3, i.e., $\Pr[\mathsf{S}_3] = 1/2$. To see this, let us assume $\mathcal{A}$ issued $\mathtt{Test}(i^*, s^*)$ and received a key that is not a $\perp$. That is $\pi_{i^*}^{s^*}$ is in the `accept` state. Due to the modification we made in *game*2 and by the definition of the Type-5 or Type-6 strategy, $\pi_{i^*}^{s^*}$ has no partner oracle $\pi_{\hat{\jmath}}^t$ for any $t \in [\ell]$ and the peer $P_{\hat{\jmath}}$ was not corrupted before $\pi_{i^*}^{s^*}$ completes the protocol execution conditioning on $\mathsf{E}_{\mathsf{testO}}$ not occurring. On the other hand, if $\pi_{i^*}^{s^*}$ is in the `accept` state, then event $\mathsf{E}_{\mathsf{sig}}$ must have not triggered. Consequently, there exists some oracle $\pi_{\hat{\jmath}}^t$ that output $(\mathsf{sid}_{i^*}^{s^*}, \sigma^*)$. Parsing $\mathsf{sid}_{i^*}^{s^*}$ as $P_{i^*}\|P_{\hat{\jmath}}\|\mathsf{lpk}_{i^*}\|\mathsf{lpk}_{\hat{\jmath}}\|\mathsf{ek}_T^*\|\mathsf{C}^*\|\mathsf{C}_T^*$, this implies that $\pi_{\hat{\jmath}}^t$ and $\pi_{i^*}^{s^*}$ are partner oracles. Since this forms

a contradiction, $\mathcal{A}$ can only receive $\perp$ when it issues $\texttt{Test}(i^*, s^*)$. Hence, since the challenge bit b is statistically hidden from $\mathcal{A}$, we have $\Pr[\mathsf{S}_3] = 1/2$.

It remains to bound $\Pr[\mathsf{E}_{\mathsf{sig}}]$. We do this by constructing an algorithm $\mathcal{B}_{3,0}$ against the EUF-CMA security of SIG. The description of $\mathcal{B}_{3,0}$ follows: $\mathcal{B}_{3,0}$ receives the public parameter $\mathsf{pp}_{\mathsf{SIG}}$ and the challenge verification key vk^*. $\mathcal{B}_{3,0}$ sets up the public parameter of $\Pi_{\mathsf{SC\text{-}AKE}}$ as in Game 2 using $\mathsf{pp}_{\mathsf{SIG}}$. $\mathcal{B}_{3,0}$ then samples $\hat{\jmath}$ randomly from $[\mu]$, runs $(\mathsf{dk}_{\hat{\jmath}}, \mathsf{ek}_{\hat{\jmath}}) \leftarrow \mathsf{KEM.KeyGen}(\mathsf{pp}_{\mathsf{KEM}})$, and sets the long-term public key of party $P_{\hat{\jmath}}$ as $\mathsf{lpk}_{\hat{\jmath}} := (\mathsf{ek}_{\hat{\jmath}}, \mathsf{vk}^*)$. The long-term secret key is implicitly set as $\mathsf{lsk}_{\hat{\jmath}} := (\mathsf{dk}_{\hat{\jmath}}, \mathsf{sk}^*)$, where sk^* is unknown to $\mathcal{B}_{3,0}$. For the rest of the parties P_i for $i \in [\mu \backslash \hat{\jmath}]$, $\mathcal{B}_{3,0}$ generates $(\mathsf{lpk}_i, \mathsf{lsk}_i)$ as in Game 2. Finally, $\mathcal{B}_{3,0}$ invokes $\mathcal{A}$ on input the public parameter of $\Pi_{\mathsf{SC\text{-}AKE}}$ and $\{\ \mathsf{lpk}_i \mid i \in [\mu]\ \}$ and answers the queries by $\mathcal{A}$ as follows:

- $\texttt{Send}(i, s, \langle \texttt{START} : \mathsf{role}, j \rangle)$: $\mathcal{B}_{3,0}$ responds as in Game 2.
- $\texttt{Send}(j, t, m = (\mathsf{ek}_T, \sigma_i))$: Depending on the value of j, it performs the following:
  - If $j = \hat{\jmath}$, then $\mathcal{B}_{3,0}$ prepares $\mathsf{sid}_{\hat{\jmath}}^t$ as in Game 2, and then sends $\mathsf{sid}_{\hat{\jmath}}^t$ to its signing oracle and receives back a signature σ' for message $\mathsf{sid}_{\hat{\jmath}}^t$ under sk^*. $\mathcal{B}_{3,0}$ then responds as in Game 2 except that it sets $\sigma := \sigma'$.
  - If $j \neq \hat{\jmath}$, then $\mathcal{B}_{3,0}$ responds as in Game 2.
- $\texttt{Send}(i, s, m = (\mathsf{C}, \mathsf{C}_T, \mathsf{c}))$: $\mathcal{B}_{3,0}$ responds as in Game 2.
- $\texttt{RevLTK}(i)$, $\texttt{RegisterLTK}(i)$, $\texttt{RevState}(i, s)$, $\texttt{RevSessKey}(i, s)$: $\mathcal{B}_{3,0}$ responds as in Game 2. Here, note that since $\mathcal{A}$ follows the Type-5 or Type-6 strategy, $\mathcal{B}_{3,0}$ can answer all the $\texttt{RevLTK}$-query. Namely, $\mathcal{A}$ never queries $\texttt{RevLTK}(\hat{\jmath})$ (i.e., $\mathsf{lsk}_{\hat{\jmath}} := (\mathsf{dk}_{\hat{\jmath}}, \mathsf{sk}^*)$) conditioning on $\mathsf{E}_{\mathsf{testO}}$ not occurring, which is the only query that $\mathcal{B}_{3,0}$ cannot answer.
- $\texttt{Test}(i, s)$: $\mathcal{B}_{3,0}$ responds as in Game 2. Here, in case $\mathsf{Pid}_i^s \neq \hat{\jmath}$, then event $\mathsf{E}_{\mathsf{testO}}$ is triggered so it aborts.

It is clear that $\mathcal{B}_{3,0}$ perfectly simulates the view of Game 2 to $\mathcal{A}$. Below, we analyze the probability that $\mathcal{B}_{3,0}$ breaks the EUF-CMA security of SIG and relate it to $\Pr[\mathsf{E}_{\mathsf{sig}}]$.

We assume $\mathcal{A}$ issues $\texttt{Test}(i^*, s^*)$. Let the message sent by the initiator oracle $\pi_{i^*}^{s^*}$ be $(\mathsf{ek}_T^*, \sigma_{i^*})$ and the message received by $\pi_{i^*}^{s^*}$ be $(\mathsf{C}^*, \mathsf{C}_T^*, \mathsf{c}^*)$. Let σ^* be the signature recovered from c^*. Then, by the definition of the Type-5 or Type-6 strategy and conditioned on $\mathsf{E}_{\mathsf{testO}}$ not occurring, the tested oracle $\pi_{i^*}^{s^*}$ satisfies the following conditions:

- $\mathsf{role}_{i^*}^{s^*} = \texttt{init}$ and $\mathsf{Pid}_{i^*}^{s^*} = \hat{\jmath}$,
- $\pi_{i^*}^{s^*}$ is in the $\texttt{accept}$ state. This implies $\mathsf{SIG.Verify}(\mathsf{vk}^*, P_{i^*} \| P_{\hat{\jmath}} \| \mathsf{lpk}_{i^*} \| \mathsf{lpk}_{\hat{\jmath}} \| \mathsf{ek}_T^* \| \mathsf{C}^* \| \mathsf{C}_T^*, \sigma^*) = 1$ holds,
- $P_{\hat{\jmath}}$ was not corrupted before $\pi_{i^*}^{s^*}$ completes the protocol execution,
- $\pi_{i^*}^{s^*}$ has no partner oracles.

Since $\pi_{i^*}^{s^*}$ has no partner oracles, there exists no responder oracle $\pi_{\hat{\jmath}}^t$ that has received ek_T^* from P_{i^*} and output $(\mathsf{C}^*, \mathsf{C}_T^*)$. In other words, there is no oracle $\pi_{\hat{\jmath}}^t$ that has signed on the message $P_{i^*} \| P_{\hat{\jmath}} \| \mathsf{lpk}_{i^*} \| \mathsf{lpk}_{\hat{\jmath}} \| \mathsf{ek}_T^* \| \mathsf{C}^* \| \mathsf{C}_T^*$. Notice that this is exactly the event $\mathsf{E}_{\mathsf{sig}}$; an initiator oracle $\pi_{i^*}^{s^*}$ receives a signature that was not signed by an oracle $\pi_{\hat{\jmath}}^t$ for any $t \in [\ell]$, and $P_{\hat{\jmath}}$ was not corrupted when $\pi_{i^*}^{s^*}$ receives the signature. Therefore,

$\mathcal{B}_{3,0}$ obtains a valid forgery $(P_{i^*}\|P_{\hat{\jmath}}\|\mathsf{lpk}_{i^*}\|\mathsf{lpk}_{\hat{\jmath}}\|\mathsf{ek}_T^*\|\mathsf{C}^*\|\mathsf{C}_T^*,\sigma^*)$, and we have $\Pr[\mathsf{E}_{\mathsf{sig}}] = \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{SIG}}(\mathcal{B}_{3,0})$. Combining everything together, we conclude

$$\epsilon_1 \leq \mu \cdot \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{SIG},\mathcal{B}_{3,0}}(\kappa).$$

□

Proof of Lemma 3.3.7: Against Type-7 or Type-8 Adversary.

Lemma 3.3.7. *For any* QPT *adversary* $\mathcal{A}$ *using the Type-7 or Type-8 strategy, there exists a* QPT *algorithm* $\mathcal{B}_{3,1}$ *breaking the* EUF-CMA *of* SIG *such that*

$$\epsilon_1 \leq \mu \cdot \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{SIG},\mathcal{B}_{3,1}}(\kappa).$$

Proof of Lemma 3.3.7. We present the rest of the sequence of games from Game 1.

Game 2**.** In this game, at the beginning of the game, $\mathcal{C}$ chooses a party $P_{\hat{\imath}}$ uniformly at random from the μ parties. Let $\mathsf{E}_{\mathsf{testO}}$ be the event that the peer of the tested oracle is not $P_{\hat{\imath}}$. If event $\mathsf{E}_{\mathsf{testO}}$ occurs, $\mathcal{C}$ aborts. Since $\mathcal{C}$ guesses the choice made by $\mathcal{A}$ correctly with probability $1/\mu$, we have

$$\epsilon_2 = \frac{1}{\mu}\epsilon_1.$$

Game 3**.** This game is identical to Game 2, except that we add an abort condition. Let S be a list of message-signature pairs that $P_{\hat{\imath}}$ generated as being an initiator oracle. That is, every time $\pi_{\hat{\imath}}^s$ for some $s \in [\ell]$ is invoked as an initiator, it updates the list S by appending the message-signature pair $(\mathsf{ek}_{\hat{\imath}}^s, \sigma_{\hat{\imath}}^s)$ that it generates. Then, when a responder oracle π_j^t for any $(j,t) \in [\mu] \times [\ell]$ is invoked on input (ek_T, σ) from party $P_{\hat{\imath}}$ (i.e., $\mathsf{Pid}_j^t = \hat{\imath}$), it checks if $\mathsf{SIG.Verify}(\mathsf{vk}_{\hat{\imath}}, \mathsf{ek}_T, \sigma) = 1$ and $P_{\hat{\imath}}$ is not corrupted, then $(\mathsf{ek}_T, \sigma) \in S$. If not, the game aborts. Otherwise, it proceeds as in the previous game. We call the event this abort occurs as $\mathsf{E}_{\mathsf{sig}}$. Since the two games are identical until abort, we have

$$|\Pr[\mathsf{S}_2] - \Pr[\mathsf{S}_3]| \leq \Pr[\mathsf{E}_{\mathsf{sig}}].$$

Before, bounding $\Pr[\mathsf{E}_{\mathsf{sig}}]$, we finish the proof of the lemma. We show that no adversary $\mathcal{A}$ following the Type-7 or Type-8 strategy has winning advantage in Game 3, i.e., $\Pr[\mathsf{S}_3] = 1/2$. To see this, let us assume $\mathcal{A}$ issued $\mathtt{Test}(j^*, t^*)$ and received a key that is not a $\perp$. That is $\pi_{j^*}^{t^*}$ is in the `accept` state. Due to the modification we made in *game*2 and by the definition of the Type-7 or Type-8 strategy, $\pi_{j^*}^{t^*}$ has no partner oracle $\pi_{\hat{\imath}}^s$ for any $s \in [\ell]$ and the peer $P_{\hat{\imath}}$ was not corrupted before $\pi_{j^*}^{t^*}$ completes the protocol execution conditioning on $\mathsf{E}_{\mathsf{testO}}$ not occurring. On the other hand, if $\pi_{j^*}^{t^*}$ is in the `accept` state, then event $\mathsf{E}_{\mathsf{sig}}$ must have not been triggered. Consequently, there exists some oracle $\pi_{\hat{\imath}}^s$ that output $(\mathsf{ek}_{\hat{\imath}}^s, \sigma_{\hat{\imath}}^s)$ and $\pi_{j^*}^{t^*}$ receives it. This implies that $\pi_{\hat{\imath}}^s$ and $\pi_{j^*}^{t^*}$ are partner oracles. Since this forms a contradiction, $\mathcal{A}$ can only receive $\perp$ when it issues $\mathtt{Test}(j^*, t^*)$. Hence, since the challenge bit b is statistically hidden from $\mathcal{A}$, we have $\Pr[\mathsf{S}_3] = 1/2$.

It remains to bound $\Pr[\mathsf{E}_{\mathsf{sig}}]$. We do this by constructing an algorithm $\mathcal{B}_{3,1}$ against the EUF-CMA security of SIG. The description of $\mathcal{B}_{3,1}$ follows: $\mathcal{B}_{3,1}$ receives the public parameter $\mathsf{pp}_{\mathsf{SIG}}$ and the challenge verification key vk^*. $\mathcal{B}_{3,1}$ sets up the public parameter of $\Pi_{\mathsf{SC\text{-}AKE}}$ as in Game 2 using $\mathsf{pp}_{\mathsf{SIG}}$. $\mathcal{B}_{3,1}$ then samples $\hat{\imath}$ randomly from $[\mu]$, runs $(\mathsf{dk}_{\hat{\imath}}, \mathsf{ek}_{\hat{\imath}}) \leftarrow \mathsf{KEM.KeyGen}(\mathsf{pp}_{\mathsf{KEM}})$, and sets the long-term public key

of party $P_{\hat{\imath}}$ as $\mathsf{lpk}_{\hat{\imath}} := (\mathsf{ek}_{\hat{\imath}}, \mathsf{vk}^*)$. The long-term secret key is implicitly set as $\mathsf{lsk}_{\hat{\imath}} := (\mathsf{dk}_{\hat{\imath}}, \mathsf{sk}^*)$, where sk^* is unknown to $\mathcal{B}_{3,1}$. For the rest of the parties P_i for $i \in [\mu \backslash \hat{\imath}]$, $\mathcal{B}_{3,1}$ generates $(\mathsf{lpk}_i, \mathsf{lsk}_i)$ as in Game 2. Finally, $\mathcal{B}_{3,1}$ invokes $\mathcal{A}$ on input the public parameter of $\Pi_{\mathsf{SC\text{-}AKE}}$ and $\{ \mathsf{lpk}_i \mid i \in [\mu] \}$ and answers the queries by $\mathcal{A}$ as follows:

- $\mathtt{Send}(i, s, \langle \mathtt{START} : \mathsf{role}, j \rangle)$: Depending on the value of i, it performs the following:
  - If $i = \hat{\imath}$, then $\mathcal{B}_{3,1}$ prepares ek_T as in Game 2, and then sends ek_T to its signing oracle and receives back a signature σ' for message ek_T under sk^*. $\mathcal{B}_{3,1}$ then responds as in Game 2 except that it sets $\sigma_i := \sigma'$.
  - If $i \neq \hat{\imath}$, then $\mathcal{B}_{3,1}$ responds as in Game 2.
- $\mathtt{Send}(j, t, m = (\mathsf{ek}_T, \sigma_i))$: $\mathcal{B}_{3,1}$ responds as in Game 2.
- $\mathtt{Send}(i, s, m = (\mathsf{C}, \mathsf{C}_T, \mathsf{c}))$: $\mathcal{B}_{3,1}$ responds as in Game 2.
- $\mathtt{RevLTK}(i)$, $\mathtt{RegisterLTK}(i)$, $\mathtt{RevState}(i, s)$, $\mathtt{RevSessKey}(i, s)$: $\mathcal{B}_{3,1}$ responds as in Game 2. Here, note that since $\mathcal{A}$ follows the Type-7 or Type-8 strategy, $\mathcal{B}_{3,1}$ can answer all the $\mathtt{RevLTK}$-query. Namely, $\mathcal{A}$ never queries $\mathtt{RevLTK}(\hat{\imath})$ (i.e., $\mathsf{lsk}_{\hat{\imath}} := (\mathsf{dk}_{\hat{\imath}}, \mathsf{sk}^*)$) conditioning on $\mathsf{E}_{\mathsf{testO}}$ not occurring, which is the only query that $\mathcal{B}_{3,1}$ cannot answer.
- $\mathtt{Test}(i, s)$: $\mathcal{B}_{3,1}$ responds as in Game 2. Here, in case $\mathsf{Pid}_i^s \neq \hat{\imath}$, then event $\mathsf{E}_{\mathsf{testO}}$ is triggered, so it aborts.

It is clear that $\mathcal{B}_{3,1}$ perfectly simulates the view of Game 2 to $\mathcal{A}$. Below, we analyze the probability that $\mathcal{B}_{3,1}$ breaks the EUF-CMA security of SIG and relate it to $\Pr[\mathsf{E}_{\mathsf{sig}}]$.

We assume $\mathcal{A}$ issues $\mathtt{Test}(j^*, t^*)$. Let the message received by the responder oracle $\pi_{j^*}^{t^*}$ be $(\mathsf{ek}_T^*, \sigma^*)$. Then, by the definition of the Type-7 or Type-8 strategy and conditioned on $\mathsf{E}_{\mathsf{testO}}$ not occurring, the oracle $\pi_{j^*}^{t^*}$ satisfies the following conditions:

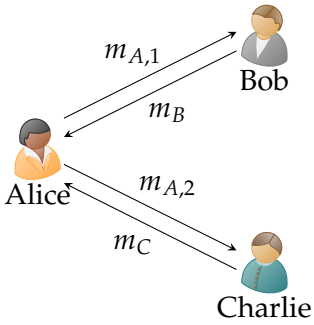
- $\mathsf{role}_{j^*}^{t^*} = \mathtt{resp}$ and $\mathsf{Pid}_{j^*}^{t^*} = \hat{\imath}$,
- $\pi_{j^*}^{t^*}$ is in the $\mathtt{accept}$ state. This implies $\mathsf{SIG.Verify}(\mathsf{vk}^*, \mathsf{ek}_T^*, \sigma^*) = 1$ holds,
- $P_{\hat{\imath}}$ was not corrupted before $\pi_{j^*}^{t^*}$ completes the protocol execution,
- $\pi_{i^*}^{s^*}$ has no partner oracles.

Since $\pi_{j^*}^{t^*}$ has no partner oracles, there exists no initiator oracle $\pi_{\hat{\imath}}^{s}$ that has output $(\mathsf{ek}_T^*, \sigma^*)$. In other words, there is no oracle $\pi_{\hat{\imath}}^{s}$ that has signed the message ek_T^*. Notice that this is exactly the event $\mathsf{E}_{\mathsf{sig}}$; a responder oracle $\pi_{j^*}^{t^*}$ receives a signature that was not signed by an oracle $\pi_{\hat{\imath}}^{s}$ for any $s \in [\ell]$, and $P_{\hat{\imath}}$ was not corrupted when $\pi_{j^*}^{t^*}$ receives the signature. Therefore, $\mathcal{B}_{3,1}$ obtains a valid forgery $(\mathsf{ek}_T^*, \sigma^*)$, and we have $\Pr[\mathsf{E}_{\mathsf{sig}}] = \mathsf{Adv}_{\mathsf{SIG}}^{\mathsf{EUF\text{-}CMA}}(\mathcal{B}_{3,1})$

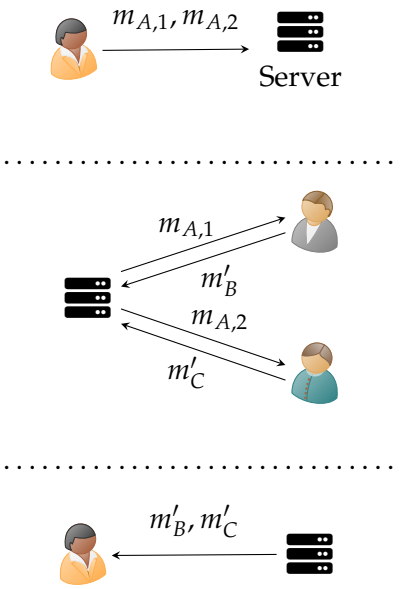
Combining everything together, we conclude

$$\epsilon_1 \leq \mu \cdot \mathsf{Adv}_{\mathsf{SIG},\mathcal{B}_{3,1}}^{\mathsf{EUF\text{-}CMA}}(\kappa).$$

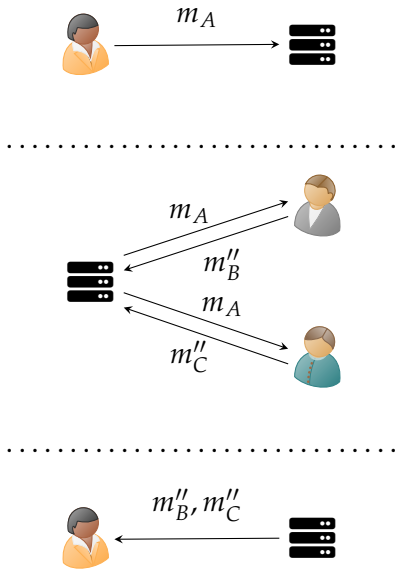
□



(A) AKE protocol



(B) Signal handshake as secure as AKE protocol but with large overhead (not used by the Signal app)



(C) Signal handshake slightly weaker compared to AKE protocol but with small overhead (used by the Signal app)

FIGURE 3.3: Comparison of the message flow of an AKE protocol (left) and that of a Signal handshake (center and right). The center protocol is more secure than the right protocol, while the right protocol is efficient 9n terms of storage and bandwidth compared to the center. The implemented Signal protocol uses the right protocol.

3.4 Post-Quantum Signal Handshake

In this section, we provide a concrete discussion on how to turn our Signal-conforming AKE protocol $\Pi_{\mathsf{SC\text{-}AKE}}$ into a post-quantum Signal handshake protocol. The proposed protocol can be used as a simple drop-in for the current X3DH protocol as a post-quantum secure replacement.

3.4.1 Signal Handshake From Signal-conforming AKE protocol

We first explain how to obtain a Signal handshake (i.e., Signal's initial key agreement protocol) from a Signal-conforming AKE protocol. Figure 3.3a depicts the message flow in AKE protocols. If Alice wants to share session keys with Bob and Charlie respectively, she sends the first message $m_{A,1}$ (resp. $m_{A,2}$) to Bob (resp. Charlie). On receiving the message, Bob and Charlie return the second message m_B and m_C to Alice respectively, and she derives the session keys. In AKE protocols, each party communicates synchronously and the communicating parties are online at the same time.

In contrast, in secure messaging, parties are not always online, and even more, the communicating partner may be unknown at the time of registration. Therefore, we need a mechanism that allows parties to start communication even when the partners are offline and undefined. Signal handshake realizes an asynchronous and anonymous communication by using a possibly untrusted server (see Figure 3.3b). In its most secure version (which is not used by the Signal app), if Alice anticipates communicating with at most two parties, she first uploads two first messages $m_{A,1}$ and $m_{A,2}$ of the AKE protocol to the server and

goes offline. She will replenish the first messages on the server periodically so that the server does not use up all the first messages. Since Alice does not know who will use the first messages in this case, only Signal-conforming AKE protocols (i.e., two-round and the first message can be generated independently of the responder) can be used for the Signal handshake. Next, if Bob wants to share a session key with Alice, he accesses the server and receives the unused $m_{A,1}$. Then, Bob computes the session key and uploads the second message m'_B. If Charlie also exchanges a session key with Alice, he executes the protocol in the same way as Bob, using a different unused first message $m_{A,2}$. Finally, when Alice comes online, she downloads the second messages m'_B and m'_C from the server and derives the session keys between Bob and Charlie. In this way, parties can exchange session keys asynchronously. Moreover, the Signal handshake in Figure 3.3b can be shown to be as secure as the underlying AKE protocol. This is because we can view the server as a person-in-the-middle adversary in AKE protocols, and AKE protocols are defined to be secure against such adversaries.

Although secure, the downside of this approach is that Alice must upload as many first messages as the number of parties she anticipates communicating with. To reduce storage overhead, the Signal protocol reuses the first message for multiple key exchange sessions (see Figure 3.3c). Alice uploads one first message m_A to the server, and periodically updates it, e.g., once a week or once a month. Bob and Charlie exchange session keys with Alice using the same first message m_A. Effectively, both the party's and server's storage overheads are reduced. The downside of reusing the first message is that it may make the protocol less secure compared to the underlying AKE protocol with the cost of better storage size. For example, if an adversary obtains the randomness used to generate m_A, depending on the underlying AKE protocol, it may expose both session keys exchanged between Bob and Charlie. In contrast, if the first message was not reused, then the security of the AKE protocol guarantees that an adversary can recover only the session key that used m_A. A more detailed discussion on the side effect of reusing the first message in our Signal handshake is provided next.

3.4.2 Details of Our Post-Quantum Signal Handshake

We provide the details of our post-quantum Signal handshake based on our Signal-conforming AKE protocol $\Pi_{\mathsf{SC\text{-}AKE}}$. Unlike the X3DH protocol, the proposed protocol can be made post-quantum by choosing appropriate post-quantum building blocks. The protocol description is presented in Figure 3.4.

As explained in the previous section, parties communicate with each other with the help of the server and reuse the first messages of the AKE protocol for a certain interval (see Figure 3.3c). First, Alice and Bob generate their long-term keys $\mathsf{lpk_A}$ and $\mathsf{lpk_B}$, respectively, and register them to the server (see the top of Figure 3.4). Note that parties upload their long-term keys only once. As in the Signal *app*, we assume the validity of the long-term keys are checked between the parties by some "out-of-bound" authentication mechanism (see [MP16b, Section 4.1]). Next, Alice uploads a first message of the AKE protocol: an encapsulation key ek_T (called *signed pre-key* in the Signal white paper [MP16b]) along with a signature σ_{A} (called *pre-key signature* in the Signal white paper [MP16b]) to the server (see the center of Figure 3.4)[18]. The signed pre-key is reused for multiple key exchange sessions and is updated at some interval (e.g. once a week or once a month). Finally, when Bob wants to communicate with Alice, he accesses the server and downloads Alice's long-term key and signed pre-key $(\mathsf{lpk_A}, \mathsf{ek}_T, \sigma_{\mathsf{A}})$. Then, he runs the AKE protocol of the responder's part and uploads the response message $(\mathsf{C}, \mathsf{C}_T, \mathsf{c})$ to the server. Finally, when Alice comes online, she downloads the long-term key and the response message from Bob and derives the shared initial secret (see the bottom of Figure 3.4). It is clear that if the signed pre-keys are not reused, then the protocol is secure as the underlying Signal-conforming AKE protocol.

[18]Unlike in the figure, the signed pre-key and pre-key signature are uploaded by all the parties and not only by Alice.

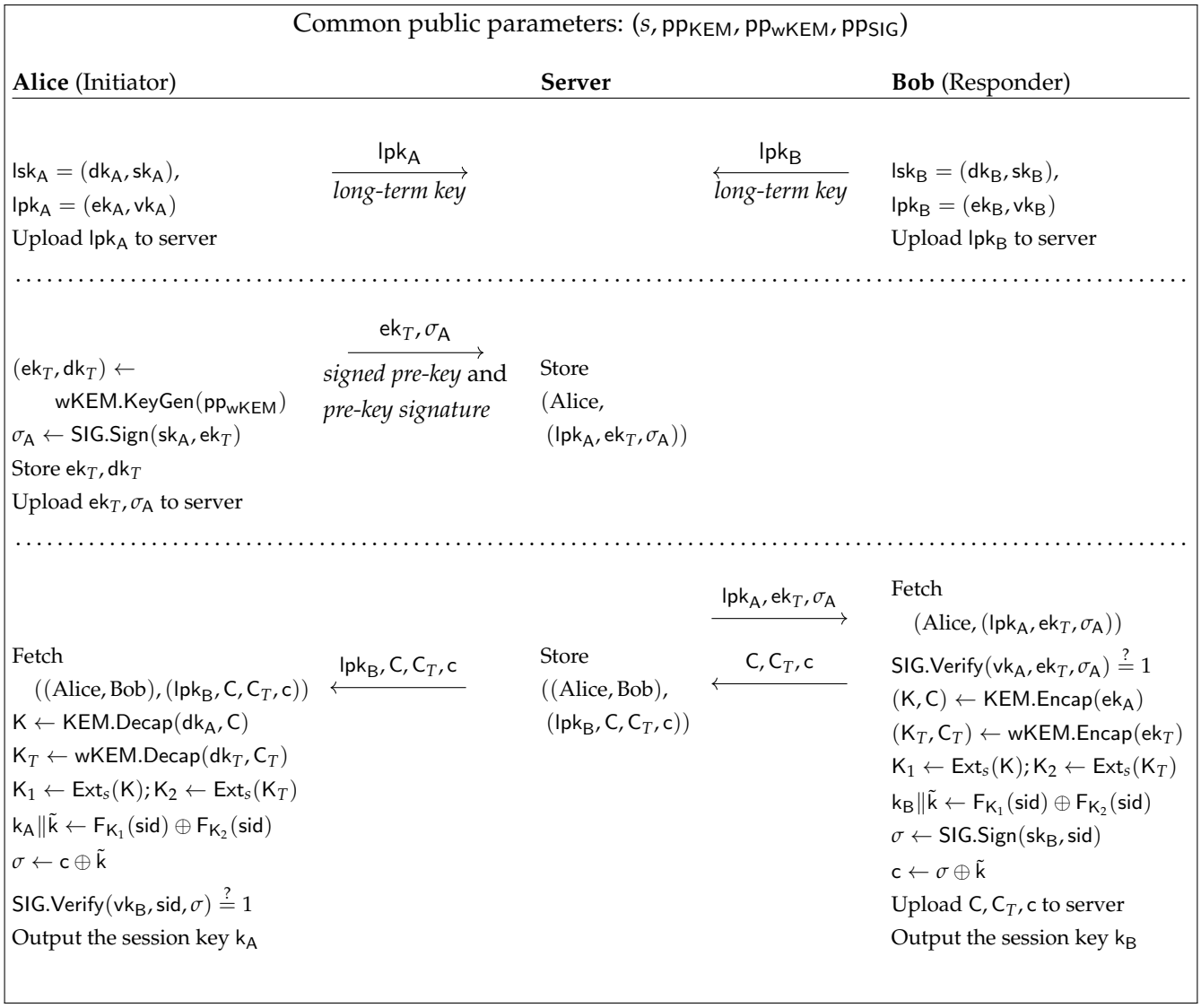


FIGURE 3.4: Post-quantum Signal handshake protocol based on our Signal-conforming AKE protocol $\Pi_{\mathsf{SC\text{-}AKE}}$ that reuses the same first message (i.e., signed pre-key and pre-key signature). The session identifier is defined as $\mathsf{sid} := \mathsf{A}\|\mathsf{B}\|\mathsf{lpk}_A\|\mathsf{lpk}_B\|\mathsf{ek}_T\|\mathsf{C}\|\mathsf{C}_T$. Alice and Bob only need to upload their long-term (public) keys to the server once, and the signed pre-key and pre-key signature are reused by multiple responders throughout some interval.

Let us discuss the security implication of reusing signed pre-keys. It is clear that it *seems* insecure to reuse the signed pre-keys compared to not reusing them. The question is, to what extent are they insecure? In our post-quantum Signal handshake, multiple session keys are exchanged using the same pair of long-term keys and signed pre-key. We first argue that, if either the long-term key or the signed pre-key is not leaked, then the exchanged keys remain secure. Our security model (defined in Section 3.2) guarantees that the session key of oracle π is secure even if an adversary forwards the first message, i.e., a signed pre-key and pre-key signature, generated by oracle π, to multiple responder oracles and obtains at most one of the two KEM keys include in the long-term key or the signed pre-key used by π. This attack captures the scenario where a party sends the same first message to multiple responders, i.e., reuses its signed pre-key. Therefore, our post-quantum Signal handshake is as secure as a variant that never reuses a signed pre-key (i.e., our Signal-conforming AKE protocol) as long as one of the long-term keys or the signed pre-key is not leaked. The distinction between our post-quantum Signal handshake and our Signal-conforming AKE protocol becomes clear when both the long-term key and the signed pre-key are leaked. Observe that once both keys are leaked, the adversary can compute *all* session keys that were exchanged using them. In other words, the number of session keys that are compromised is the same as the number of times the signed pre-key was reused. Thus, the Signal handshake is less secure than the underlying AKE protocol since the number of session keys that are exposed is larger when both the long-term key and the signed pre-key are leaked. To mitigate the number of exposed session keys, those parties looking for better security can use signed pre-keys only once or add a so-called *one-time pre-key* in the first message (i.e., an additional non-signed one-time KEM key). This is exactly what the Signal app does. In this case, even if all the KEM keys used in a specific session are leaked, an adversary can not compute the session keys of the other sessions since a different KEM key is used for each session. Thus, this mitigation can be used to enhance security in a scenario where the long-term key and the signed pre-key are both exposed but the one-time pre-key is not.

3.5 Instantiating Post-Quantum Signal Handshake

In this section, we present the implementation details of our post-quantum Signal handshake protocol presented in Figure 3.4. We take existing implementations of post-quantum KEMs and signature schemes submitted for the NIST post-quantum cryptography (PQC) standardization. To instantiate our Signal handshake, we pair variants of KEMs and signature schemes corresponding to the same security level. We consider security levels 1, 3, and 5 as defined by NIST for the PQC standardization. With more than 20 variants of KEM and 14 variants of signature schemes, we can create at least 93 different instantiations[19] of post-quantum Signal handshake protocols. The provided implementation simulates post-quantum, weakly deniable authenticated key exchange between two parties. We study the efficiency of our instantiations through two metrics — the total amount of data exchanged between parties and run-time performance. Our implementation [Kwi20] is available in the form of open-source software.

3.5.1 Instantiation details

Our implementation is instantiated with the following building blocks:

- s: (pseudo)-randomly generated 32 bytes of data calculated at session initialization phase,

[19] In Table 3.2, pairing KEMs and signatures schemes with the same NIST security level yields $7 \times 5 + 7 \times 4 + 6 \times 5 = 93$ distinct combinations (some schemes offer multiple instantiations at a given NIST level).

TABLE 3.2: Considered KEM and signature schemes under NIST security levels 1, 3, and 5. †: Use two SPHINCS instantiations with different hash functions. ‡: Use two NTRU instantiations with different parameter sets.

NIST security level	KEM	Signature
1	SABER, CLASSIC-MCELIECE, KYBER, NTRU HQC, SIKE, FRODOKEM	RAINBOW, FALCON, DILITHIUM SPHINCS†
3	SABER,NTRU‡, CLASSIC-MCELIECE, KYBER, HQC, FRODOKEM	DILITHIUM, RAINBOW SPHINCS†
5	SABER, CLASSIC-MCELIECE, NTRU, KYBER FRODOKEM, HQC	FALCON, RAINBOW,DILITHIUM SPHINCS†

- Ext_s: uses HMAC-SHA256 as a strong randomness extractor. As an input message, we use a key K/K_T prepended with byte 0x02 which works as a domain separator (since we also use HMAC-SHA256 as a PRF). Security of using HMAC as a strong randomness extractor is studied in [FPZ08],
- PRF: uses HMAC-SHA256 as a PRF. The session-specific sid is used as an input message to HMAC, prepended with byte 0x01. An output from Ext_s is used as a key. Security of using HMAC as a PRF is studied in [Bel06; Bel15],
- b: equals the security level of the underlying post-quantum KEM scheme, where $b \in \{128, 192, 256\}$,
- d: equals the byte length of the signature generated by the post-quantum signature scheme SIG,
- KEM, wKEM, SIG: the implementation uses pairs of KEM and signature schemes. The list of the schemes used can be found in Table 3.2. We always use the same KEM scheme for KEM and wKEM.

At a high level, the implementation is split into 4 main parts. A setup phase, where both parties perform long-term key generation and initialization of required during memory benchmarking. The session establishment phase implements an initiator's signed pre-key generation (the `offer` function), the responder's session key generation (`accept` function), and initiator's session key generation (`finalize` function), which finalizes session establishment resulting in session key. To evaluate the cost of our post-quantum Signal handshake, we instantiate the protocol with KEM and signature schemes from Table 3.2.

The concrete implementation of post-quantum schemes is provided by *PQ Crypto Catalog* library[20], which is a collection of implementations submitted to the NIST PQC standardization process. We also use *LibTomCrypt* library[21] which provides an implementation of the building blocks HMAC, HKDF and SHA-256. We note that we use portable `C` code implementations of schemes, which do not include platform-specific optimizations. There are two reasons for such a choice. First, our goal was to show the expected results on a broad number of platforms. Second, the *PQ Crypto Catalog* library does not provide hardware-assisted optimizations for all schemes, hence enabling those optimizations only for some algorithms would result in unfair comparison.

[20]https://github.com/kriskwiatkowski/pqc
[21]https://github.com/libtom/libtomcrypt

3.5.2 Efficiency Analysis

In this subsection, we provide an assessment of the costs related to running the concrete instantiation of the post-quantum Signal handshake.

To properly assess the cost, we modeled a scenario according to Figure 3.4. Namely, two parties try to establish a session key. Alice (the initiator) and Bob (the responder) generate and make their long-term public keys (lpk_A, lpk_B) available to others. Alice then generates her *signed pre-key* ek_T and creates her *pre-key signature* σ_A by signing it. The pair $(\mathsf{ek}_T, \sigma_\mathsf{A})$ is also uploaded to the publicly accessible server. Bob retrieves the pre-key bundle $(\mathsf{lpk}_\mathsf{A}, \mathsf{ek}_T, \sigma_\mathsf{A})$ and uses it to perform his part of the session establishment. Namely, Bob generates the triple $(\mathsf{C}, \mathsf{C}_T, \mathsf{c})$ and makes it available for Alice to download from the server. Once Alice comes online, she downloads the session initialization bundle from Bob together with his long-term key, lpk_B. She then finalizes the process by computing the session key on her side. Note that in the Signal protocol, long-term public keys lpk are fetched from the server. Parties do not store the keys lpk corresponding to those that they have not communicated with before.[22]

We provide three metrics:

- Data transfer cost: the amount of data exchanged when two parties establish a session key.
- Storage cost: the amount of data that needs to be stored on the server to allow a session establishment between parties.
- Computational cost: the number of CPU cycles spent in computation during session establishment by both parties.

Cost analysis for each metric is provided separately.

Data Transfer Cost. Table 3.3 provides the selected results for Round 3 candidates of the NIST PQC standardization process.[23] The lpk column contains the byte size of a long-term key. The following four columns contain the byte size of the data exchanged by the initiator, the server, and the responder during a session key establishment (as per Figure 3.4). Finally, the **Total** column contains the total size of data exchanged between Alice and Bob.

From Table 3.3, we can conclude that the transfer cost for Falcon512 paired with SIKEp434 is the order of magnitude lower than in the case of the other two pairs. Also, the small size of the Falcon public key and signature size makes it an attractive choice for the signature scheme in the case of that particular application.

Note that the long-term public keys (lpk) are uploaded to the server only once by each party (initiator and responder), hence the cost of uploading them is probably negligible for most applications. To further minimize the transfer cost, some implementations may decide to use caching mechanisms, meaning long-term keys are downloaded only once and cached locally. In this case, the validity of the key may be checked by hashing the lpk_A at both sides and comparing the hash values. In this case, Bob sends a hash of cached lpk_A when requesting the pre-key bundle, the server compares hashes and depending on the result of such comparison sends a response either with long-term keys or without.

Remark 3.5.1 (Note on Low-Quality Network Links). We anticipate the Signal handshake to be used with handheld devices and areas with a poor-quality network connection. In such cases, larger key, ciphertext,

[22] The X3DH protocol assumes the parties authenticate the long-term public keys through some authenticated channel [MP16b, Section 4.1].

[23] The results for all 93 instantiations can be found in the repository containing the implementation [Kwi20].

TABLE 3.3: Data transfer cost in bytes of Figure 3.4 instantiated with various post-quantum schemes. We use the following abbreviations: I = Initiator, S = Server, R = Responder. Note that:
(a) (S→R) – (I→S) = (S→I) – (R→S) = |pk|.
(b) Total := (I→S) + (S→R) + (R→S) + (S→I).

Scheme	|pk|	I→S	S→R	R→S	S→I	Total
NIST security level 1						
Falcon512/Saber Light	1569	1362	2931	2162	3731	10186
Falcon512/SIKEp434	1227	1020	2247	1382	2609	7258
Dilithium2/NTRU hps2048509	2011	3119	5130	3818	5829	17896
SPHINCS-SHAKE256-128f-s/Saber Light	704	17760	18464	18560	19264	74048
NIST security level 3						
Dilithium3/NTRU hps2048677	2882	4223	7105	5153	8035	24516
Dilithium3/Saber	2944	4285	7229	5469	8413	25396
Rainbow III/McEliece460896	1406240	524324	1930564	540	1406780	3862208
SPHINCS-SHAKE256-192f-s/Kyber768	1232	36848	38080	37840	39072	151840
NIST security level 5						
Falcon1024/NTRU hps4096821	3023	2560	5583	3790	6813	18746
Falcon1024/Saber Fire	3105	2642	5747	4274	7379	20042
SPHINCS-SHAKE256-256f-s/Saber Fire	1376	51168	52544	52800	54176	210688

and signature sizes generated may negatively impact the quality of the connection. Network packet loss is an additional factor that should be considered when choosing schemes for concrete instantiation.

Data on the network is exchanged in packets. The maximum transmission unit (MTU) defines the maximal size of a single packet, usually set to 1500 bytes. Ideally, the size of data sent between participants in a single pass is less than MTU. Network quality is characterized by a packet loss rate. When a packet is lost, the TCP protocol ensures that it is retransmitted, where each retransmission causes a delay. A typical data loss on a high-quality network link is below 1%, while data loss on a mobile network depends on the strength of the network signal.

Depending on the scheme used, an increased packet loss may negatively impact session establishment time (see [PST20]). For example, a scheme instantiated with Falcon512 paired with Saber Light requires an exchange of $npacks = 7$ packets over the network, where instantiation with SPHINCS-SHAKE256-128f-simple paired with Saber Light requires 27 packets. Assuming increased packet rate loss of 2%, the probability of losing a packet in the former case is $1 - (1 - rate)^{npacks} = 13\%$, whereas in the latter it is 42%. In the latter case, at the median, every third session key establishment will experience packet retransmission and hence a delay.

Storage Cost. The Signal handshake protocol assumes the usage of an intermediate server during session key establishment. This allows parties to be offline during the establishment. The server stores long-term keys of each party uploaded during registration, signed pre-keys and pre-key signatures needed to initiate session establishment, as well as data generated during session establishment. Hence, it is important to correctly assess the amount of storage required.

The cost can be split into two parts. One part contains storage of the long-term key lpk and *signed pre-key* pair $(\mathsf{ek}_T, \sigma_\mathsf{A})$. The latter is updated on regular basis, but the server always stores one *signed pre-key*,

TABLE 3.4: Data storage cost in bytes of Figure 3.4 instantiated with various post-quantum schemes.

Scheme	Data per user	Data per session
NIST security level 1		
Falcon512/Saber Light	2931	2162
Falcon512/SIKEp434	2247	1382
Dilithium2/NTRU hps2048509	2985	2088
SPHINCS-SHAKE256-128f-s/Saber Light	18464	18560
NIST security level 3		
Dilithium3/NTRU hps2048677	7105	5153
Dilithium3/Saber	7229	5469
Rainbow III/McEliece460896	1930564	540
SPHINCS-SHAKE256-192f-s/Kyber768	38080	37840
NIST security level 5		
Falcon1024/NTRU hps4096821	5583	3790
Falcon1024/Saber Fire	5747	4274
Dilithium5/Kyber1024	10323	7731
SPHINCS-SHAKE256-256f-s/Saber Fire	52544	52800

hence that cost is constant and depends on the number of parties registered. The second part of the cost is data produced during session establishment, namely triple $(\mathsf{C}, \mathsf{C}_T, \mathsf{c})$ uploaded by the receiver. It is a variable cost, as data can be deleted from the server as soon as the initiator downloads it to finalize the session establishment.

Table 3.4 shows the split between both costs for a selected number of post-quantum schemes. We can see that to reduce the storage cost, it is beneficial to pair Falcon with SIKE for security levels 1 and 5 and Dilithium with NTRU for security level 3. In some applications, it could be interesting to reduce only the variable cost. In that case, instantiation can use Rainbow and McEliece pair of algorithms at a higher cost of long-term key storage.

Computational Cost. The computational cost of the protocol depends on the performance of the cryptographic primitives used. More precisely, the most expensive operations are those done by the post-quantum schemes. Our post-quantum Signal handshake performs 9 such operations during a session agreement: the initiator runs a KEM key generation, two KEM decapsulations, one signature generation, and one signature verification, and the responder performs two KEM encapsulations, one signature generation, and one signature verification.

It is important that our post-quantum Signal handshake protocol runs efficiently. But as it is an offline protocol, the performance is less critical when compared to online protocols (e.g., TLS). The most important difference is that, in the case of Signal handshake, the server does not perform any CPU-heavy operations, the session establishment happens less often and parties perform session establishment asynchronously when they are online.

The most performance-critical part of the protocol is the final part of session establishment done by the initiator. At that stage, it may happen that multiple parties request to establish a session with the initiator. In that case, the initiator downloads multiple (and unknown) triples of $(\mathsf{C}, \mathsf{C}_T, \mathsf{c})$, which then need to be efficiently processed. Hence, when optimizing for speed the performance of KEM decapsulation and

TABLE 3.5: Computational cost in CPU cycles of Figure 3.4 instantiated with various post-quantum schemes. Benchmarking run on the Intel Xeon E3-1220v3 @3.1GhZ with Turbo Boost disabled.

Scheme	3-way handshake	Finish
NIST security level 1		
Falcon512/Saber Light	3596396	638610
Falcon512/SIKEp434	1803371672	810254556
Dilithium2/NTRU hps2048509	6159630	748797
SPHINCS-SHAKE256-128f-s/Saber Light	256821041	11827077
NIST security level 3		
Dilithium3/NTRU hps2048677	11747527	1493656
Dilithium3/Saber	7597588	1476533
Rainbow III/McEliece460896	2798014516	513810164
SPHINCS-SHAKE256-192f-s/Kyber768	710632430	17476256
NIST security level 5		
Falcon1024/NTRU hps4096821	12718741	1328866
Falcon1024/Saber Fire	7611706	1417632
Dilithium5/Kyber1024	10576515	1672158
SPHINCS-SHAKE256-256f-s/Saber Fire	1343959982	18091448

signature verification is most important.

Table 3.5 contains the number of CPU cycles spent during session establishment for selected post-quantum schemes. The **3-way handshake** column shows the cost of the whole session establishment, and the **Finish** column contains the final part of the handshake, performed by the initiator. The presented results exclude the cost of long-term key (lpk) generation.

The best performance comes from instantiations that use Saber KEM and either Falcon or Dilithium signature scheme. We see instantiation with SIKE, SPHINCS, Rainbow, or McEliece schemes negatively impacts performance, resulting in orders of magnitude slower execution.

We note that the computational cost is far less absolute as it depends on the concrete implementation of the post-quantum schemes.

In conclusion. Instantiations of our post-quantum Signal handshake protocol that use Saber as a KEM and either Falcon or Dilithium as signature scheme seem to be the most promising choice for minimizing transfer and storage cost and maximizing performance.

Lastly, our implementation is based on open-source libraries. In total, we created 93 instantiations with different post-quantum schemes. We store the results in the repository containing the implementation [Kwi20]. We note that a variety of fine-tuning can be done, leading to different results. For example, one could imagine a scenario crafted for IoT devices, in which devices are pre-configured to communicate only with selected parties. In such a case, the exchange of long-term keys can be done ahead of time.

3.6 Adding Deniability to Basic Signal-Conforming AKE

In this section, we discuss to what extent our Signal-conforming AKE protocol $\Pi_{\mathsf{SC\text{-}AKE}}$ satisfies deniability and show how to modify the protocol to satisfy a progressively stronger notion of deniability. We first

motivate what deniability is and then provide an overview of this section.

Difference Between Deniability of an AKE Protocol and The Signal Handshake. Due to the subtle difference in the model of the standard AKE protocol and the Signal handshake, there is also a subtle difference in what it means to be deniable. In an AKE protocol, roughly, deniability states that the exchanged transcript does not leave any trace of the two parties that supposedly communicated with each other. Namely, both the initiator *and* responder in Figure 3.2 should be able to deny the fact that they engaged in a key exchange protocol. In contrast, in the Signal handshake, we mainly care about the deniability of the responder (i.e, Bob in Figure 3.4). This is because the initiator (i.e., Alice in Figure 3.4) only uploads materials that are independent of the responder. Specifically, an adversary can at most prove to a third party that the initiator was using the Signal app by showing the pre-key signature of the initiator, and nothing more. Therefore, in the Signal handshake, the main focus is to prevent an adversary from later proving that a certain responder tried to exchange a key with some (possibly malicious) initiator. In summary, the deniability required by an AKE protocol is arguably stronger than what is required by the Signal handshake since it also considers the deniability of the initiator.

That being said, in this section, we mainly focus on the deniability of our AKE protocol rather than the Signal handshake for three reasons. First, we believe our AKE protocol is interesting even outside the context of Signal, so it is worth investigating what kind of deniability it offers. Second, it is easy to argue deniability of the Signal handshake once the deniability of the AKE protocol is established since it only consists of ignoring the deniability of the initiator. Finally, if our AKE protocol (or a variant of it) can be shown to be deniable, then when viewed as the Signal handshake, we can further show that the initiator can deny the fact of using the Signal app. Specifically, since the uploaded content of the initiator would also be deniable, it cannot be used as evidence that the initiator was using the Signal app.

Overview of This Section. We first informally show that our AKE protocol $\Pi_{\mathsf{SC\text{-}AKE}}$ already has a very weak form of deniability that may be acceptable in some applications. We then show that we can slightly modify $\Pi_{\mathsf{SC\text{-}AKE}}$ by replacing a standard signature with a *ring signature* to satisfy a stronger notion of deniability. Although this satisfies a much stronger notion of deniability compared to our vanilla $\Pi_{\mathsf{SC\text{-}AKE}}$, it still assumes the parties follow the protocol description (i.e., honest-but-curious). We discuss in Remark 3.6.15 why this notion of deniability can still be insufficient in practice. Finally, we show how to make the protocol even more secure against malicious adversaries that can deviate arbitrarily from the protocol by additionally relying on NIZKs. As it is common with all deniable AKE protocols[24] secure against key-compromise attacks [DGK06; YZ10; Vat+20], we rely on strong knowledge-type assumptions, including a variant of the *plaintext-awareness* (PA) for the KEM scheme [BR95; Bel+98; BP04].

We note that the proposed protocol is deniable against *quantum* adversaries when they are limited to be honest-but-curious. However, we were not able to formally prove if the proposed protocol satisfies deniability against quantum adversaries taking arbitrary strategies. We believe this is an artifact of the current definition or proof strategy of deniability and leave it as an interesting open problem to formalize and prove the quantum deniability of the proposed protocol (see Remark 3.6.4 for more discussion).

Weak Deniability of $\Pi_{\mathsf{SC\text{-}AKE}}$. Our Signal-conforming AKE protocol $\Pi_{\mathsf{SC\text{-}AKE}}$ already satisfies a very weak notion of deniability, where the communication transcript does not leave a trace of the *responder* if the two parties honestly execute the AKE protocol. Note that it clearly leaves a trace of the initiator since a signature is included. Concretely, an adversary (e.g., the server) that is passively collecting the communication transcript cannot convince a third party that some responder tried to communicate with some initiator. Informally, this can be checked by observing that the message sent from the responder can be simulated

[24]We only consider schemes that are proven secure in the (possibly slight variant of the) deniability framework proposed by the seminal work of Di Raimondo et al. [DGK06].

by the adversary on its own. This notion of weak deniability may suffice for some particular settings: only the deniability of the responder is required; the two engaging parties fully trust each other for the correct execution of the protocol; and if they can tolerate the assumption that corruption will not occur. For instance, this includes the Signal handshake setting where the server is trying to provide proof to a judge that some responder tried to engage in a conversation with some initiator *without* the help of either of the parties. The proposed protocol will guarantee deniability in such a scenario.

However, in other cases, we may want to guarantee deniability even in the case the communicating peer may be compromised, or even worse, acting maliciously. In the above example, if the server is colluding with the initiator of the protocol, then they can provide proof that the responder wanted to start a conversation with the initiator by using knowledge of the session key. This is clear from the fact that in our $\Pi_{\mathsf{SC\text{-}AKE}}$ protocol, the responder generates a signature that nobody else can. Furthermore, in the context of an AKE protocol, it is also desirable for the initiator to be able to deny the fact that it was trying to engage in a key exchange protocol with some responder.

We now discuss how to make the proposed protocol satisfy a stronger notion of deniability where both the initiator and responder can deny even when the communicating peer may be compromised. To this end, we first define deniability for AKE protocols.

3.6.1 Definition of Deniability and Tool Preparation

We follow a simplified definition of deniability for AKE protocols introduced in the seminal work by Di Raimondo et al. [DGK06]. Discussion on the simplification is provided in Remark 3.6.3. At a high level, if there exists a simulator $\mathsf{SIM}_{\mathcal{M}}$ that uses only public information that can produce the same view to an adversary $\mathcal{M}$ that engages in a real AKE protocol with honest parties, then the protocol is deniable. The intuition is that if such a $\mathsf{SIM}_{\mathcal{M}}$ exists, then when $\mathcal{M}$ presents a protocol transcript as "proof" that some party was trying to communicate with it, the party can deny the fact by claiming that the transcript could have been generated by $\mathcal{M}$ running $\mathsf{SIM}_{\mathcal{M}}$ (that only uses public information).

Let Π_{AKE} be an AKE protocol and KeyGen be the key generation algorithm. That is, for any integer $\mu = \mathsf{poly}(\kappa)$ representing the number of parties in the system, define $\mathsf{KeyGen}(1^\kappa, \mu) \to (\mathsf{pp}, \mathsf{LPK}, \mathsf{LSK})$, where pp is the public parameter used by the system and $\mathsf{LPK} := \{ \mathsf{lpk}_i \mid i \in [\mu] \}$ and $\mathsf{LSK} := \{ \mathsf{lsk}_i \mid i \in [\mu] \}$ are the corresponding long-term public and secret keys of the μ parties, respectively.

Let $\mathcal{M}$ denote an adversary that engages in an AKE protocol with μ-honest parties in the system with long-term public keys LPK, acting as either an initiator or a responder. $\mathcal{M}$ may run individual sessions against an honest party in a concurrent manner and may deviate from the AKE protocol in an arbitrary fashion. The goal of $\mathcal{M}$ is not to impersonate someone to an honest party P but to collect (cryptographic) evidence that an honest party P interacted with $\mathcal{M}$. Therefore, when $\mathcal{M}$ interacts with P, it can use a (*possibly maliciously generated*) long-term public key $\mathsf{lpk}_{\mathcal{M}}$ that can be either associated to or not to $\mathcal{M}$'s identity. We then define the *view* of the adversary $\mathcal{M}$ as the entire sets of inputs and outputs of $\mathcal{M}$ and the *session keys* computed in all the protocols in which $\mathcal{M}$ participated with an honest party. Here, we assume in case the session is not completed by $\mathcal{M}$, the session key is defined as $\perp$. We denote this view as $\mathsf{View}_{\mathcal{M}}(\mathsf{pp}, \mathsf{LPK}, \mathsf{LSK})$. Note that if the session key is deniable, then the subsequent communications (that do not use any long-term keys) are deemed deniable as well. In other words, if the establishment of the session key is deniable, then any communications only using that session key between the two parties are deniable as well.

In order to define deniability, we consider a simulator SIM that simulates the view of honest parties (both initiator and responder) to the adversary $\mathcal{M}$ *without* knowledge of the corresponding long-term secret keys LSK of the honest parties. Specifically, SIM takes as input all the input given to the adversary $\mathcal{M}$ (along

with the description of $\mathcal{M}$) and simulates the view of $\mathcal{M}$ with the real AKE protocol Π_{AKE}. We denote this simulated view as $\mathsf{SIM}_{\mathcal{M}}(\mathsf{pp}, \mathsf{LPK})$. Roughly, if the view simulated by $\mathsf{SIM}_{\mathcal{M}}$ is indistinguishable from those generated by $\mathsf{View}_{\mathcal{M}}$, then we say the AKE protocol is deniable since $\mathcal{M}$ could have run $\mathsf{SIM}_{\mathcal{M}}$ (which does not take any secret information as input) to generate its view in the real protocol. Here, unlike the zero-knowledge simulator for NIZKs, the simulator for an AKE protocol must be executable in the real world [Pas03]. More formally, we have the following.

Definition 3.6.1 (Deniability). *We say an AKE protocol Π_{AKE} with key generation algorithm KeyGen is* deniable, *if for any integer $\mu = \mathsf{poly}(\kappa)$ and PPT adversary $\mathcal{M}$, there exist a PPT simulator $\mathsf{SIM}_{\mathcal{M}}$ such that the following two distributions are (computationally) indistinguishable for any PPT distinguisher $\mathcal{D}$:*

$$\mathsf{D}_{\mathsf{Real}} := \{\mathsf{pp}, \mathsf{LPK}, \mathsf{View}_{\mathcal{M}}(\mathsf{pp}, \mathsf{LPK}, \mathsf{LSK}) : (\mathsf{pp}, \mathsf{LPK}, \mathsf{LSK}) \leftarrow \mathsf{KeyGen}(1^{\kappa}, \mu)\},$$
$$\mathsf{D}_{\mathsf{Sim}} := \{\mathsf{pp}, \mathsf{LPK}, \mathsf{SIM}_{\mathcal{M}}(\mathsf{pp}, \mathsf{LPK}) : (\mathsf{pp}, \mathsf{LPK}, \mathsf{LSK}) \leftarrow \mathsf{KeyGen}(1^{\kappa}, \mu)\}.$$

When $\mathcal{M}$ is semi-honest (i.e., it follows the prescribed protocol), we say Π_{AKE} is deniable against semi-honest adversaries. *When $\mathcal{M}$ is malicious (i.e., it takes any efficient strategy), we say Π_{AKE} is* deniable against malicious adversaries.

In the above definition, a semi-honest adversary $\mathcal{M}$ is equivalent to an adversary that engages in the protocol honestly but it may try to learn as much as possible from the messages they receive from other parties. Semi-honest adversaries are also termed *passive* since they are only allowed to break security by observing a view of honest protocol execution.

Remark 3.6.2 (Including Public Information and Session Keys). It is crucial that the two distributions $\mathsf{D}_{\mathsf{Real}}$ and $\mathsf{D}_{\mathsf{Sim}}$ include the public information $(\mathsf{pp}, \mathsf{LPK})$. Otherwise, $\mathsf{SIM}_{\mathcal{M}}$ can simply create its own set of $(\mathsf{pp}', \mathsf{LPK}', \mathsf{LSK}')$ and simulate the view to $\mathcal{M}$. However, this does not correctly capture deniability in the real world since $\mathcal{M}$ would not be able to convince anybody with such a view using public information that it cooked up on its own. In addition, it is essential that the value of the session key is part of the output of $\mathsf{SIM}_{\mathcal{M}}$. This guarantees that the exchanged contents of the sessions authenticated by the session key can also be denied.

Remark 3.6.3 (Comparison Between Prior Definition). Our definition is weaker than the deniability notion originally proposed by Di Raimondo et al. [DGK06]. In their definition, an adversary $\mathcal{M}$ (and therefore the simulator $\mathsf{SIM}_{\mathcal{M}}$) is also provided as input some auxiliary information aux that can depend non-trivially on $(\mathsf{pp}, \mathsf{LPK}, \mathsf{LSK})$. For instance, this allows capturing information that $\mathcal{M}$ may have obtained by eavesdropping on conversations between honest parties (which is not modeled by $\mathsf{View}_{\mathcal{M}}$). Since our goal is to provide a preliminary result on the deniability of the proposed protocol, we only focus on the weaker definition where $\mathcal{M}$ does not obtain such auxiliary information. We leave it as future work to prove the proposed protocol deniable in the sense of Di Raimondo et al. [DGK06].[25] We also note that stronger forms of deniability are known and formalized in the universally composable (UC) model [Dod+09; UG15; UG18], however, AKE protocols satisfying such a strong deniability notion are known to achieve weaker security guarantees. For instance, as noted in [UG18], an AKE protocol cannot be online deniable while also being secure against KCI attacks.

[25] We observe that although in [DGK06, Definition 2], aux is defined as fixed information that $\mathcal{M}$ cannot adaptively choose, their proof implicitly assumes that aux is sampled adaptively from some distribution dependent on $(\mathsf{pp}, \mathsf{LPK}, \mathsf{LSK})$. Such adaptivity of aux is necessary to invoke PA-2 security of the underlying encryption scheme in their security proof. We consider that enhancing the deniability definition of [DGK06] to capture this adaptivity is an important future work.

Remark 3.6.4 (Extending to Malicious Quantum Adversaries). We only consider *classical* deniability in this work, where the adversary $\mathcal{M}$ is restricted to be classical. To be precise, although we are able to easily show deniability against semi-honest quantum adversaries, we are not able to do so against malicious quantum adversaries. This is mainly due to the fact that to prove deniability against malicious classical adversaries, we require a strong knowledge-type assumption (i.e., plaintext-awareness for KEM) that assumes the existence of an extractor that can invoke an adversary multiple times on the *same* randomness. The notion of fixing randomness is not well-defined in the quantum setting and rewinding an adversary without disturbing the adversary's quantum state is a non-trivial task. We leave it as an interesting problem to formally define a set of tools that allow showing deniability even against malicious quantum adversaries.

Remark 3.6.5 (A Note on the Deniability Definition of [Bre+22]). After the proceedings version of our paper [Has+21a] appeared, Brendel et al. [Bre+22] introduced a new definition of deniability for AKE protocols. Unlike prior definitions for AKE deniability [DGK06; Dod+09; YZ10; UG15; UG18; Vat+20], Brendel et al. considers an indistinguishability-based definition rather than a simulation-based definition. They consider a scenario where all the users honestly generate their keys and the adversary $\mathcal{M}$ is given the secret keys to all of the users. Informally, $\mathcal{M}$ can receive a transcript by querying the challenge oracle on a pair (I, R), representing the Initiator and Receiver. In one mode, $\mathcal{M}$ is given an honest transcript of a real AKE protocol between I and R. In another mode, $\mathcal{M}$ is given a transcript simulated by a simulator SIM who is only given the secret key of R (i.e., the responder) as input. An AKE protocol is then said to be deniable if no efficient adversary $\mathcal{M}$ can tell apart the two modes.

Other than the fact that their definition is not simulation-based, there are three main differences between the definition of Brendel et al. and Definition 3.6.1: (1) the users are assumed to generate their keys honestly; (2) the adversary $\mathcal{M}$ is assumed to remain passive during the execution of the AKE protocol between I and R; (3) $\mathcal{M}$ is given all the secret keys of the users. Regardless of $\mathcal{M}$ being malicious or semi-honest in Definition 3.6.1, the definition of Brendel et al. is stronger regarding (3) since the secret keys of the users are not provided to $\mathcal{M}$ in Definition 3.6.1. On the other hand, Definition 3.6.1 is always stronger than Brendel et al. regarding (2) since $\mathcal{M}$ is allowed to deviate from the AKE protocol. Finally, when $\mathcal{M}$ can act maliciously in Definition 3.6.1, it is stronger than Brendel et al. regarding (1) since $\mathcal{M}$ can inject malicious keys to the system. In general, the two definitions are incomparable.

It is not immediately clear what the real-world impact is of whether or not (1), (2), and (3) are satisfied. In Remark 3.6.15, we show that if $\mathcal{M}$ can register malicious keys, then it can break deniability, thus showing that deniability under (1) can be insufficient in some practical applications. Put differently, considering only a semi-honest adversary $\mathcal{M}$ in Definition 3.6.1 or the definition of Brendel et al. may be insufficient. We leave the investigation of the impact of (2) and (3) as interesting future work.

Required Tools. To argue deniability in the following sections we rely on the following tools: ring signature, plaintext-aware (PA-1) secure KEM scheme, and a non-interactive zero-knowledge (NIZK) argument. We use standard notions of ring signatures and NIZK arguments as provided in Sections 2.7 and 2.8. On the other hand, we use a slightly stronger variant of PA-1 secure KEM schemes than those originally defined in [BR95; Bel+98; BP04]. Informally, a KEM scheme is PA-1 secure if for any adversary $\mathcal{M}$ that outputs a valid ciphertext C, there is an extractor $\mathsf{Ext}_{\mathcal{M}}$ that outputs the associating plaintext K. In our work, we require PA-1 security to hold even when $\mathcal{M}$ is given multiple public keys rather than a single public key [MSs12]. We note that although Di Raimondo et al. [DGK06] considered the standard notion of PA-1 security in their seminal work on deniability of AKE protocols, we observe that their proof only works in the case where multiple public keys are considered. Finally, we further require the extractor $\mathsf{Ext}_{\mathcal{M}}$ to be efficiently computable given $\mathcal{M}$ (which is another subtle restriction omitted in the definition used in [DGK06]). The formal definition is provided in Section 2.2.

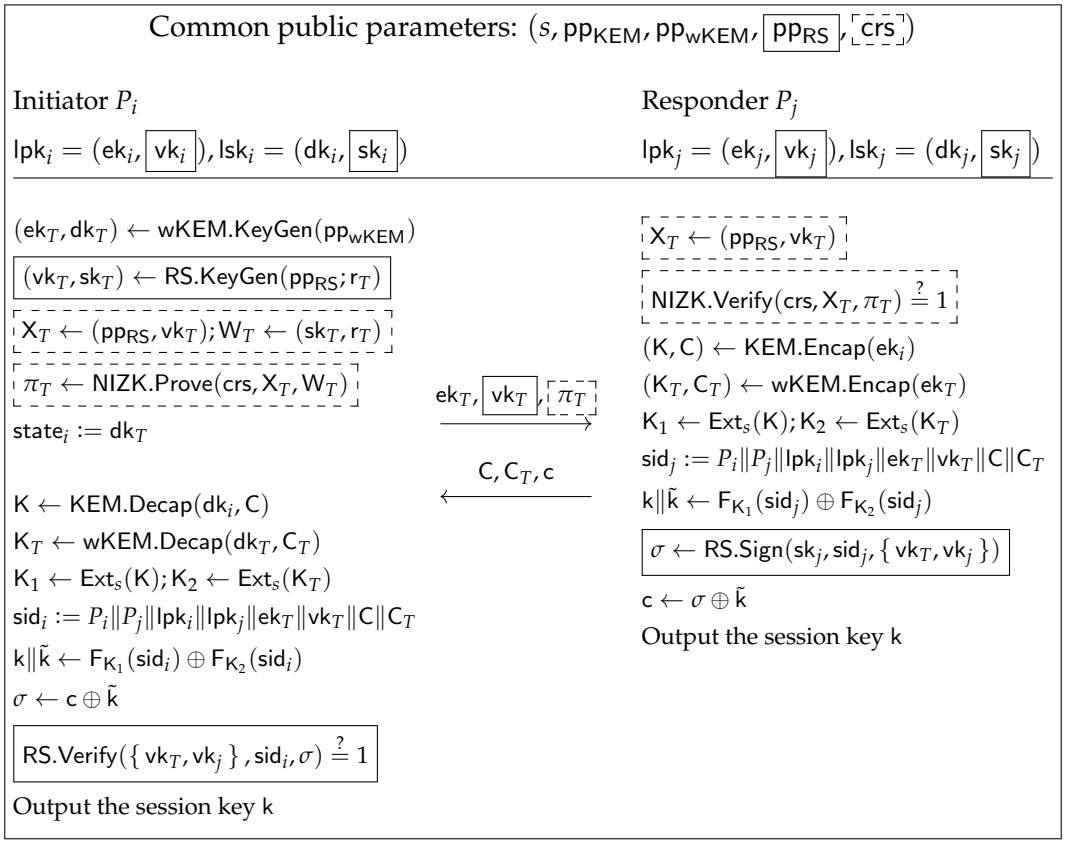


FIGURE 3.5: Deniable Signal-conforming AKE protocol $\Pi_{\mathsf{SC\text{-}DAKE}}$ and $\Pi'_{\mathsf{SC\text{-}DAKE}}$. The initiator no longer signs the first message and the other components that differ from the non-deniable protocol $\Pi_{\mathsf{SC\text{-}AKE}}$ are indicated by (dotted) boxes. The protocol with (resp. without) the dotted-box components satisfies deniability against malicious (resp. semi-honest) adversaries.

3.6.2 Deniable Signal-Conforming AKE $\Pi_{\mathsf{SC\text{-}DAKE}}$ against Semi-Honest Adversaries

We provide a Signal-conforming AKE protocol $\Pi_{\mathsf{SC\text{-}DAKE}}$ that is deniable against semi-honest adversaries. The construction of $\Pi_{\mathsf{SC\text{-}DAKE}}$ is a simple modification of $\Pi_{\mathsf{SC\text{-}AKE}}$ where the initiator no longer signs the first message and a standard signature is replaced by a ring signature. In the context of the Signal handshake, this means the initiator no longer uploads a pre-key signature. We show that this modification provides a secure AKE protocol that has weak forward secrecy as in Definition 3.2.3. In Remark 3.6.16, we provide some discussion on what happens if the initiator signs the first message as in $\Pi_{\mathsf{SC\text{-}AKE}}$, while the responder uses a ring signature.

In Section 3.6.3, we show how to further modify $\Pi_{\mathsf{SC\text{-}DAKE}}$ to a protocol that is deniable even against malicious adversaries by relying on other tools. The high-level idea presented in this section naturally extends to the malicious setting.

An overview of $\Pi_{\mathsf{SC\text{-}DAKE}}$ and $\Pi'_{\mathsf{SC\text{-}DAKE}}$ is provided in Figure 3.5, where the dotted-box components are only used to obtain deniability against malicious adversaries.

Building Blocks. Our deniable Signal-conforming AKE protocol $\Pi_{\mathsf{SC\text{-}DAKE}}$ against semi-honest adversaries consists of the following building blocks.

- $\mathsf{KEM} = (\mathsf{KEM.Setup}, \mathsf{KEM.KeyGen}, \mathsf{KEM.Encap}, \mathsf{KEM.Decap})$ is a KEM scheme that is IND-CCA secure and assume we have $(1-\delta_{\mathsf{KEM}})$-correctness, ν_{KEM}-high encapsulation key min-entropy and χ_{KEM}-high ciphertext min-entropy.
- $\mathsf{wKEM} = (\mathsf{wKEM.Setup}, \mathsf{wKEM.KeyGen}, \mathsf{wKEM.Encap}, \mathsf{wKEM.Decap})$ is a KEM schemes that is IND-CPA secure (and not IND-CCA secure) and assume we have $(1-\delta_{\mathsf{wKEM}})$-correctness, ν_{wKEM}-high encapsulation key min-entropy, and χ_{wKEM}-high ciphertext min-entropy. In the following, for simplicity of presentation and without loss of generality, we assume $\delta_{\mathsf{wKEM}} = \delta_{\mathsf{KEM}}$, $\nu_{\mathsf{wKEM}} = \nu_{\mathsf{KEM}}$, $\chi_{\mathsf{wKEM}} = \chi_{\mathsf{KEM}}$.
- $\mathsf{RS} = (\mathsf{RS.Setup}, \mathsf{RS.KeyGen}, \mathsf{RS.Sign}, \mathsf{RS.Verify})$ is a ring signature scheme that is anonymous and unforgeable and assume we have $(1-\delta_{\mathsf{RS}})$-correctness. We denote d as the bit length of the signature generated by RS.Sign.
- $\mathsf{F} : \mathcal{K} \times \{0,1\}^* \to \{0,1\}^{\kappa+d}$ is a pseudo-random function family with key space $\mathcal{K}$.
- $\mathsf{Ext} : \mathcal{S} \times \mathcal{KS} \to \mathcal{K}$ is a strong $(\gamma_{\mathsf{KEM}}, \varepsilon_{\mathsf{Ext}})$-extractor.

Public Parameters. All the parties in the system are provided the following public parameters as input: $(s, \mathsf{pp}_{\mathsf{KEM}}, \mathsf{pp}_{\mathsf{wKEM}}, \mathsf{pp}_{\mathsf{RS}})$. Here, s is a random seed chosen uniformly from $\mathcal{S}$, and pp_{X} for $\mathsf{X} \in \{\mathsf{KEM}, \mathsf{wKEM}, \mathsf{RS}\}$ are public parameters generated by X.Setup.

Long-Term Public and Secret Keys. Each party P_i runs $(\mathsf{ek}_i, \mathsf{dk}_i) \leftarrow \mathsf{KEM.KeyGen}(\mathsf{pp}_{\mathsf{KEM}})$ and $(\mathsf{vk}_i, \mathsf{sk}_i) \leftarrow \mathsf{RS.KeyGen}(\mathsf{pp}_{\mathsf{RS}})$. Party P_i's long-term public key and secret key are set as $\mathsf{lpk}_i = (\mathsf{ek}_i, \mathsf{vk}_i)$ and $\mathsf{lsk}_i = (\mathsf{dk}_i, \mathsf{sk}_i)$, respectively.

Construction. A key exchange between an initiator P_i in the s-th session (i.e., π_i^s) and responder P_j in the t-th session (i.e., π_j^t) is executed as in Figure 3.2. More formally, we have the following.

1. Party P_i sets $\mathsf{Pid}_i^s := j$ and $\mathsf{role}_i^s := \texttt{init}$. P_i computes $(\mathsf{dk}_T, \mathsf{ek}_T) \leftarrow \mathsf{wKEM.KeyGen}(\mathsf{pp}_{\mathsf{wKEM}})$ and $(\mathsf{vk}_T, \mathsf{sk}_T) \leftarrow \mathsf{RS.KeyGen}(\mathsf{pp}_{\mathsf{RS}})$, and sends $(\mathsf{ek}_T, \mathsf{vk}_T)$ to party P_j. P_i erases the signing key sk_T and stores the ephemeral decapsulation key dk_T as the session-state i.e., $\mathsf{state}_i^s := \mathsf{dk}_T$.[26]
2. Party P_j sets $\mathsf{Pid}_j^t := i$ and $\mathsf{role}_j^t := \texttt{resp}$. Upon receiving $(\mathsf{ek}_T, \mathsf{vk}_T)$, P_j first computes $(\mathsf{K}, \mathsf{C}) \leftarrow \mathsf{KEM.Encap}(\mathsf{ek}_i)$ and $(\mathsf{K}_T, \mathsf{C}_T) \leftarrow \mathsf{wKEM.Encap}(\mathsf{ek}_T)$ and derives two PRF keys $\mathsf{K}_1 \leftarrow \mathsf{Ext}_s(\mathsf{K})$, $\mathsf{K}_2 \leftarrow \mathsf{Ext}_s(\mathsf{K}_T)$. It then defines the session identifier as $\mathsf{sid}_j^t := P_i \| P_j \| \mathsf{lpk}_i \| \mathsf{lpk}_j \| \mathsf{ek}_T \| \mathsf{vk}_T \| \mathsf{C} \| \mathsf{C}_T$ and computes $\mathsf{k} \| \tilde{\mathsf{k}} \leftarrow \mathsf{F}_{\mathsf{K}_1}(\mathsf{sid}_j) \oplus \mathsf{F}_{\mathsf{K}_2}(\mathsf{sid}_j)$, where $\mathsf{k} \in \{0,1\}^\kappa$ and $\tilde{\mathsf{k}} \in \{0,1\}^d$. P_j sets the session key as $\mathsf{k}_j^t := \mathsf{k}$. P_j then signs $\sigma \leftarrow \mathsf{RS.Sign}(\mathsf{sk}_j, \mathsf{sid}_j^t, \{\mathsf{vk}_T, \mathsf{vk}_j\})$ and encrypts it as $\mathsf{c} \leftarrow \sigma \oplus \tilde{\mathsf{k}}$. Finally, it sends $(\mathsf{C}, \mathsf{C}_T, \mathsf{c})$ to P_i and sets $\mathsf{result}_j^t := \texttt{accept}$. Here, note that P_j does not require to store any session-state, i.e., $\mathsf{state}_j^t = \perp$.
3. Upon receiving $(\mathsf{C}, \mathsf{C}_T, \mathsf{c})$, P_i first decrypts $\mathsf{K} \leftarrow \mathsf{KEM.Decap}(\mathsf{dk}_i, \mathsf{C})$ and $\mathsf{K}_T \leftarrow \mathsf{wKEM.Decap}(\mathsf{dk}_T, \mathsf{C}_T)$, and derives two PRF keys $\mathsf{K}_1 \leftarrow \mathsf{Ext}_s(\mathsf{K})$ and $\mathsf{K}_2 \leftarrow \mathsf{Ext}_s(\mathsf{K}_T)$. It then sets the session identifier as $\mathsf{sid}_i^s := P_i \| P_j \| \mathsf{lpk}_i \| \mathsf{lpk}_j \| \mathsf{ek}_T \| \mathsf{vk}_T \| \mathsf{C} \| \mathsf{C}_T$ and computes $\mathsf{k} \| \tilde{\mathsf{k}} \leftarrow \mathsf{F}_{\mathsf{K}_1}(\mathsf{sid}_i) \oplus \mathsf{F}_{\mathsf{K}_2}(\mathsf{sid}_i)$, where $\mathsf{k} \in \{0,1\}^\kappa$ and $\tilde{\mathsf{k}} \in \{0,1\}^d$. P_i then decrypts $\sigma \leftarrow \mathsf{c} \oplus \tilde{\mathsf{k}}$ and checks whether $\mathsf{RS.Verify}(\{\mathsf{vk}_T, \mathsf{vk}_j\}, \mathsf{sid}_i^s, \sigma) = 1$ holds. If not, P_i sets $(\mathsf{result}_i^s, \mathsf{k}_i^s, \mathsf{state}_i^s) := (\texttt{reject}, \perp, \perp)$ and stops. Otherwise, P_i sets $(\mathsf{result}_i^s, \mathsf{k}_i^s, \mathsf{state}_i^s) := (\texttt{accept}, \mathsf{k}, \perp)$. Here, note that P_i deletes the session-state $\mathsf{state}_i^s = \mathsf{dk}_T$ at the end of the key exchange.

[26] Notice the protocol is receiver oblivious since the first message is computed independently of the receiver.

Security. We first check that $\Pi_{\mathsf{SC\text{-}DAKE}}$ is correct and secure as a standard AKE protocol. The main difference from the security proof of $\Pi_{\mathsf{SC\text{-}AKE}}$ is that we have to make sure that removing the initiator's signature only affects forward secrecy, and using a ring signature instead of a standard signature does not allow the adversary to mount a key-compromise impersonation (KCI) attack (see Section 3.2.3 for the explanation on KCI attacks).

Theorem 3.6.6 (Correctness of $\Pi_{\mathsf{SC\text{-}DAKE}}$). *Assume* KEM *and* wKEM *are* $(1-\delta_{\mathsf{KEM}})$*-correct and* RS *is* $(1-\delta_{\mathsf{RS}})$*-correct. Then, the Signal-conforming AKE protocol* $\Pi_{\mathsf{SC\text{-}DAKE}}$ *is* $(1-\mu\ell(\delta_{\mathsf{RS}}+2\delta_{\mathsf{KEM}})/2)$*-correct.*

Proof of Theorem 3.6.6. This proof is similar to the proof of Theorem 3.3.2. It is clear that an initiator oracle and a responder oracle become partners when they execute the protocol faithfully. Moreover, if no correctness error occurs in the underlying KEM schemes and ring signature scheme, the partner oracles compute an identical session key. Since each oracle is assigned to uniform randomness, the probability that a correctness error occurs in one of the underlying schemes is bounded by $\delta_{\mathsf{RS}}+2\delta_{\mathsf{KEM}}$. Since there are at most $\mu\ell/2$ responder oracles, the AKE protocol is correct except with probability $\mu\ell\cdot(\delta_{\mathsf{RS}}+2\delta_{\mathsf{KEM}})/2$. □

Theorem 3.6.7 (Security of $\Pi_{\mathsf{SC\text{-}DAKE}}$). *For any* QPT *adversary* $\mathcal{A}$ *that plays the game* $\mathsf{Game}^{\mathsf{AKE\text{-}weakFS}}_{\Pi_{\mathsf{SC\text{-}DAKE}}}(\mu,\ell)$ *with* μ *parties that establishes at most* ℓ *sessions per party, there exist* QPT *algorithms* $\mathcal{B}_1$ *breaking the* IND-CPA *security of* wKEM*,* $\mathcal{B}_2$ *and* $\mathcal{B}_4$ *breaking the* IND-CCA *security of* KEM*,* $\mathcal{B}_3$ *breaking the unforgeability of* RS*, and* $\mathcal{D}_1,\ldots,\mathcal{D}_3$ *breaking the security of PRF* F *such that*

$$\mathsf{Adv}^{\mathsf{AKE\text{-}weakFS}}_{\Pi_{\mathsf{SC\text{-}DAKE}},\mathcal{A}}(\kappa) \le \max\left\{\begin{array}{l}\mu^2\ell^2\cdot(2\mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{wKEM},\mathcal{B}_1}(\kappa)+\mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F},\mathcal{D}_1}(\kappa)+\varepsilon_{\mathsf{Ext}}),\\ \mu^2\ell\cdot(2\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{KEM},\mathcal{B}_2}(\kappa)+\mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F},\mathcal{D}_2}(\kappa)+\varepsilon_{\mathsf{Ext}})+\mu\ell^2\cdot\left(\frac{1}{2^{2\chi_{\mathsf{KEM}}}}+\frac{1}{2^{v_{\mathsf{KEM}}}}\right),\\ \mathsf{Adv}^{\mathsf{RS\text{-}Unf}}_{\mathsf{RS},\mathcal{B}_3}(\kappa),\\ \mu^2\ell\cdot\left(2\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{KEM},\mathcal{B}_4}(\kappa)+\mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F},\mathcal{D}_3}(\kappa)+\varepsilon_{\mathsf{Ext}}\right)+\mu\ell^2\cdot\frac{1}{2^{\chi_{\mathsf{KEM}}}}\end{array}\right\} + \frac{\mu\ell}{2}\cdot(\delta_{\mathsf{RS}}+2\delta_{\mathsf{KEM}}),$$

where v_{KEM} *is the encapsulation key min-entropy of* wKEM *and* KEM*, and* χ_{KEM} *is the ciphertext min-entropy of* wKEM *and* KEM*. The running time of* $\mathcal{B}_1,\ldots,\mathcal{B}_4$ *and* $\mathcal{D}_1,\ldots,\mathcal{D}_3$ *are about that of* $\mathcal{A}$*.*

Proof of Theorem 3.6.7. Let $\mathcal{A}$ be an adversary that plays the security game $\mathsf{Game}^{\mathsf{AKE\text{-}weakFS}}_{\Pi_{\mathsf{SC\text{-}DAKE}}}(\mu,\ell)$ with the challenger $\mathcal{C}$ with advantage $\mathsf{Adv}^{\mathsf{AKE\text{-}weakFS}}_{\Pi_{\mathsf{SC\text{-}DAKE}},\mathcal{A}}(\kappa)=\epsilon$. The bulk of the proof is identical to the proof of Theorem 3.3.3 for the (non-deniable) protocol $\Pi_{\mathsf{SC\text{-}AKE}}$. Namely, we divide the strategy that can be taken by $\mathcal{A}$ (listed in Table 3.1) and we construct an algorithm that breaks one of the underlying assumptions by using such an $\mathcal{A}$ as a subroutine. Formally, we construct seven algorithms $\mathcal{B}_1,\ldots,\mathcal{B}_4$ and $\mathcal{D}_1,\ldots,\mathcal{D}_3$ satisfying the following:

1. If $\mathcal{A}$ uses the Type-1 (or Type-2) strategy, then $\mathcal{B}_1$ succeeds in breaking the IND-CPA security of wKEM with advantage $\approx\frac{1}{\mu^2\ell^2}\epsilon$ or $\mathcal{D}_1$ succeeds in breaking the security of PRF F with advantage $\approx\frac{1}{\mu^2\ell^2}\epsilon$.
2. If $\mathcal{A}$ uses the Type-3 (or Type-4) strategy, then $\mathcal{B}_2$ succeeds in breaking the IND-CCA security of KEM with advantage $\approx\frac{1}{\mu^2\ell}\epsilon$ or $\mathcal{D}_2$ succeeds in breaking the security of PRF F with advantage $\approx\frac{1}{\mu^2\ell}\epsilon$.
3. If $\mathcal{A}$ uses the Type-5 or Type-6 strategy, then $\mathcal{B}_3$ succeeds in breaking the unforgeability of RS with advantage $\approx\epsilon$.

4. If $\mathcal{A}$ uses the Type-7 (or Type-8) strategy, then $\mathcal{B}_4$ succeeds in breaking the IND-CCA security of KEM with advantage $\approx \frac{1}{\mu^2\ell}\epsilon$ or $\mathcal{D}_3$ succeeds in breaking the security of PRF F with advantage $\approx \frac{1}{\mu^2\ell}\epsilon$.

We present a security proof structured as a sequence of games. Without loss of generality, we assume that $\mathcal{A}$ always issues a Test-query. In the following, let S_j denote the event that $b = b'$ occurs in Game j and let $\epsilon_j := \left|\Pr\left[\mathsf{S}_j\right] - 1/2\right|$ denote the advantage of the adversary in Game j. Regardless of the strategy taken by $\mathcal{A}$, all proofs share the common game sequences Game 0 and Game 1 as described below. Although they are identical to those of Theorem 3.3.3, we provide them for completeness.

Game 0. This game is identical to the original security game. We thus have

$$\epsilon_0 = \epsilon.$$

Game 1. This game is identical to Game 0, except that we add an abort condition. Let $\mathsf{E}_{\mathsf{corr}}$ be the event that there exist two partner oracles π_i^s and π_j^t that do not agree on the same session key. If $\mathsf{E}_{\mathsf{corr}}$ occurs, then $\mathcal{C}$ aborts (i.e., sets $\mathcal{A}$'s output to be a random bit) at the end of the game.

There are at most $\mu\ell/2$ responder oracles and each oracle is assigned uniform randomness. From Theorem 3.6.6, the probability of error occurring during the security game is at most $\mu\ell(\delta_{\mathsf{RS}} + 2\delta_{\mathsf{KEM}})/2$. Therefore, $\mathsf{E}_{\mathsf{corr}}$ occurs with probability at most $\mu\ell(\delta_{\mathsf{RS}} + 2\delta_{\mathsf{KEM}})/2$. We thus have

$$\left|\Pr\left[\mathsf{S}_0\right] - \Pr\left[\mathsf{S}_1\right]\right| \leq \frac{\mu\ell}{2} \cdot (\delta_{\mathsf{RS}} + 2\delta_{\mathsf{KEM}}).$$

In the following games we assume no decryption error or signature verification error occurs.

We now divide the game sequence depending on the strategy taken by the adversary $\mathcal{A}$. Regardless of $\mathcal{A}$'s strategy, we prove that ϵ_1 is negligible, which in particular implies that ϵ is also negligible. Formally, this is shown in Lemmata 3.6.8 to 3.6.11 provided below. We first complete the proof of the theorem. Specifically, by combining all the lemmata together, we obtain the following desired bound:

$$\begin{aligned}\mathsf{Adv}^{\mathsf{AKE\text{-}weakFS}}_{\Pi_{\mathsf{SC\text{-}DAKE}},\mathcal{A}}(\kappa) \leq \max \left\{ \begin{array}{l} \mu^2\ell^2 \cdot (2\mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{wKEM},\mathcal{B}_1}(\kappa) + \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F},\mathcal{D}_1}(\kappa) + \varepsilon_{\mathsf{Ext}}), \\ \mu^2\ell \cdot (2\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{KEM},\mathcal{B}_2}(\kappa) + \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F},\mathcal{D}_2}(\kappa) + \varepsilon_{\mathsf{Ext}}) + \mu\ell^2 \cdot \left(\frac{1}{2^{2\chi_{\mathsf{KEM}}}} + \frac{1}{2^{\nu_{\mathsf{KEM}}}}\right), \\ \mathsf{Adv}^{\mathsf{RS\text{-}Unf}}_{\mathsf{RS},\mathcal{B}_3}(\kappa), \\ \mu^2\ell \cdot \left(2\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{KEM},\mathcal{B}_4}(\kappa) + \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F},\mathcal{D}_3}(\kappa) + \varepsilon_{\mathsf{Ext}}\right) + \mu\ell^2 \cdot \frac{1}{2^{\chi_{\mathsf{KEM}}}} \end{array} \right\} \\ + \frac{\mu\ell}{2} \cdot (\delta_{\mathsf{RS}} + 2\delta_{\mathsf{KEM}}),\end{aligned}$$

Here, the running time of the algorithms $\mathcal{B}_1, \ldots, \mathcal{B}_4$ and $\mathcal{D}_1, \ldots, \mathcal{D}_3$ consist essentially the time required to simulate the security game for $\mathcal{A}$ once, plus a minor number of additional operations. □

It remains to prove Lemmata 3.6.8 to 3.6.11. Since the proof of Lemmata 3.6.10 and 3.6.11 is a direct consequence of the proof of the corresponding Lemmata 3.3.4 and 3.3.5 of Theorem 3.3.3,[27] we focus on proving Lemma 3.6.8 and Lemma 3.6.9 below.

Lemma 3.6.8. *For any* QPT *adversary* $\mathcal{A}$ *using the Type-5 or Type-6 strategy, there exists a* QPT *algorithm* $\mathcal{B}_3$ *breaking the unforgeability of* RS *such that*

$$\epsilon_1 \leq \mathsf{Adv}^{\mathsf{RS\text{-}Unf}}_{\mathsf{RS},\mathcal{B}_3}(\kappa).$$

[27] Note that Lemma 3.6.10 (resp. Lemma 3.6.11) corresponds to Lemma 3.3.4 (resp. Lemma 3.3.5).

Proof of Lemma 3.6.8. We present the rest of the sequence of games from Game 1.

Game 2. This game is identical to Game 1, except that we add an abort condition. Let S_j be a list of message-signature pairs that P_j generated as being a responder oracle. That is, every time π_j^t for some $t \in [\ell]$ is invoked as a responder, it updates the list S_j by appending the message-signature pair $(\mathsf{sid}_j^t, \sigma_j^t)$ that it generates. Then, when an initiator oracle π_i^s for any $(i,s) \in [\mu] \times [\ell]$ is invoked on input $(\mathsf{C}, \mathsf{C}_T, \mathsf{c})$ from party P_j (i.e., $\mathsf{Pid}_i^s = j$), it first computes sid_i^s and σ as in the previous game and checks if $\mathsf{RS.Verify}(\{\mathsf{vk}_T, \mathsf{vk}_j\}, \mathsf{sid}_i^s, \sigma) = 1$ and $(\mathsf{sid}_i^s, \sigma) \in S_j$. If not, the game aborts. Otherwise, it proceeds as in the previous game. We call the event this abort occurs as $\mathsf{E}_{\mathsf{sig}}$. Since the two games are identical until abort, we have

$$|\Pr[\mathsf{S}_2] - \Pr[\mathsf{S}_3]| \leq \Pr[\mathsf{E}_{\mathsf{sig}}].$$

Before, bounding $\Pr[\mathsf{E}_{\mathsf{sig}}]$, we finish the proof of the lemma. We show that no adversary $\mathcal{A}$ following the Type-5 or Type-6 strategy has winning advantage in Game 2, i.e., $\Pr[\mathsf{S}_2] = 1/2$. To see this, first let us assume $\mathcal{A}$ issued $\mathtt{Test}(i^*, s^*)$ and received a key that is not a $\perp$. In other words, $\pi_{i^*}^{s^*}$ is in the `accept` state. By the definition of the Type-5 or Type-6 strategy, $\pi_{i^*}^{s^*}$ has no partner oracle π_j^t for any $(j,t) \in [\mu] \times [\ell]$. On the other hand, if $\pi_{i^*}^{s^*}$ is in the `accept` state, then event $\mathsf{E}_{\mathsf{sig}}$ must have not triggered. Consequently, there exists some oracle π_j^t that output $(\mathsf{sid}_{i^*}^{s^*}, \sigma^*)$. Parsing $\mathsf{sid}_{i^*}^{s^*}$ as $P_{i^*} \| P_j \| \mathsf{lpk}_{i^*} \| \mathsf{lpk}_j \| \mathsf{ek}_T^* \| \mathsf{vk}_T^* \| \mathsf{C}^* \| \mathsf{C}_T^*$, this implies that π_j^t and $\pi_{i^*}^{s^*}$ are partner oracles. Since this forms a contradiction, $\mathcal{A}$ can only receive $\perp$ when it issues $\mathtt{Test}(i^*, s^*)$. Hence, since the challenge bit b is statistically hidden from $\mathcal{A}$, we have $\Pr[\mathsf{S}_2] = 1/2$.

It remains to bound $\Pr[\mathsf{E}_{\mathsf{sig}}]$. We do this by constructing an algorithm $\mathcal{B}_3$ against the unforgeability of RS. The description of $\mathcal{B}_3$ follows: $\mathcal{B}_3$ receives the public parameter $\mathsf{pp}_{\mathsf{RS}}$ and $\mu + \mu\ell$ verification keys $\mathsf{vk}_1, \ldots, \mathsf{vk}_\mu$ and $\mathsf{vk}_1^1, \ldots, \mathsf{vk}_\mu^\ell$. $\mathcal{B}_3$ sets up the public parameter of $\Pi_{\mathsf{SC\text{-}DAKE}}$ as in Game 2 using $\mathsf{pp}_{\mathsf{RS}}$. $\mathcal{B}_3$ then runs $(\mathsf{dk}_i, \mathsf{ek}_i) \leftarrow \mathsf{KEM.KeyGen}(\mathsf{pp}_{\mathsf{KEM}})$ and sets the long-term public key of party P_i as $\mathsf{lpk}_i := (\mathsf{ek}_i, \mathsf{vk}_i)$. The long-term secret key is implicitly set as $\mathsf{lsk}_i := (\mathsf{dk}_i, \mathsf{sk}_i)$, where sk_i is unknown to $\mathcal{B}_3$. Finally, $\mathcal{B}_3$ invokes $\mathcal{A}$ on input the public parameter of $\Pi_{\mathsf{SC\text{-}DAKE}}$ and $\{\mathsf{lpk}_i \mid i \in [\mu]\}$ and answers the queries by $\mathcal{A}$ as follows:

- $\mathtt{Send}(i, s, \langle \mathtt{START} : \mathsf{role}, j \rangle)$: $\mathcal{B}_3$ responds as in Game 1 except that it sets $\mathsf{vk}_T := \mathsf{vk}_i^s$.
- $\mathtt{Send}(j, t, m = (\mathsf{ek}_T, \mathsf{vk}_T))$: $\mathcal{B}_3$ responds as in Game 1 except that rather than constructing the signature σ on its own, it queries $\mathtt{Sign}(j, \mathsf{sid}_j^t, \{\mathsf{vk}_T, \mathsf{vk}_j\})$ to its signing oracle and uses the signature σ' that it receives.
- $\mathtt{Send}(i, s, m = (\mathsf{C}, \mathsf{C}_T, \mathsf{c}))$: $\mathcal{B}_3$ responds as in Game 1.
- $\mathtt{RevLTK}(i)$: $\mathcal{B}_3$ queries $\mathtt{Corr}(i)$ to its corruption oracle and receives back a signing key sk_i'. $\mathcal{B}_3$ then sets $\mathsf{sk}_i := \mathsf{sk}_i'$ and returns $\mathsf{lsk}_i = (\mathsf{dk}_i, \mathsf{sk}_i)$.
- $\mathtt{RevState}(i, s)$, $\mathtt{RevSessKey}(i, s)$: $\mathcal{B}_3$ responds as in Game 1.
- $\mathtt{Test}(i, s)$: $\mathcal{B}_3$ responds as in Game 1.

It is clear that $\mathcal{B}_3$ perfectly simulates the view of Game 2 to $\mathcal{A}$. Below, we analyze the probability that $\mathcal{B}_3$ breaks the unforgeability of RS and relate it to $\Pr[\mathsf{E}_{\mathsf{sig}}]$.

We assume $\mathcal{A}$ issues $\mathtt{Test}(i^*, s^*)$. Let the message sent by the initiator oracle $\pi_{i^*}^{s^*}$ be $(\mathsf{ek}_T^*, \mathsf{vk}_T^*)$ and the message received by $\pi_{i^*}^{s^*}$ be $(\mathsf{C}^*, \mathsf{C}_T^*, \mathsf{c}^*)$. Let σ^* be the signature recovered from c^*. Then, by the definition of the Type-5 or Type-6 strategy, the tested oracle $\pi_{i^*}^{s^*}$ satisfies the following conditions:

- $\mathsf{role}_{i^*}^{s^*} = \texttt{init}$,
- P_j is not corrupted where $\mathsf{Pid}_{i^*}^{s^*} = j$ and $j \in [\mu]$,
- $\pi_{i^*}^{s^*}$ is in the `accept` state. This implies $\mathsf{RS.Verify}(\{\mathsf{vk}_T^*, \mathsf{vk}_j\}, P_{i^*} \| P_j \| \mathsf{lpk}_{i^*} \| \mathsf{lpk}_j \| \mathsf{ek}_T^* \| \mathsf{vk}_T^* \| \mathsf{C}^* \| \mathsf{C}_T^*, \sigma^*) = 1$ holds,
- $\pi_{i^*}^{s^*}$ has no partner oracles.

Since P_j is not corrupted, $\mathcal{A}$ has never queried $\texttt{RevLTK}(j)$-query. Moreover, since an honest initiator discards sk_T^* on generation, $\mathcal{B}_3$ never uses them for simulation. These two facts imply that $\texttt{Corr}(j)$ and $\texttt{Corr}((i,T))$ has never been queried, where $\texttt{Corr}((i,T))$ is a query regarding the verification key $\mathsf{vk}_{i^*}^{s^*}$. In particular, the ring $\{\mathsf{vk}_T^*, \mathsf{vk}_j\}$ consists of non-corrupted verification keys. Moreover, since $\pi_{i^*}^{s^*}$ has no partner oracles, there exists no responder oracle π_j^t that has received $(\mathsf{ek}_T^*, \mathsf{vk}_T^*)$ from P_{i^*} and sent $(\mathsf{C}^*, \mathsf{C}_T^*)$. In other words, there is no oracle π_j^t that has signed on the message $P_{i^*} \| P_j \| \mathsf{lpk}_{i^*} \| \mathsf{lpk}_j \| \mathsf{ek}_T^* \| \mathsf{vk}_T^* \| \mathsf{C}^* \| \mathsf{C}_T^*$. Notice that this is exactly the event $\mathsf{E}_{\mathsf{sig}}$; an initiator oracle $\pi_{i^*}^{s^*}$ receives a signature that was not signed by an oracle π_j^t for any $t \in [\ell]$. Therefore, we have $\Pr[\mathsf{E}_{\mathsf{sig}}] = \mathsf{Adv}_{\mathsf{RS}}^{\mathsf{RS\text{-}Unf}}(\mathcal{B}_3)$.

Combining everything together, we conclude

$$\epsilon_1 \leq \mathsf{Adv}_{\mathsf{RS},\mathcal{B}_3}^{\mathsf{RS\text{-}Unf}}(\kappa).$$

□

Lemma 3.6.9. *For any* QPT *adversary* $\mathcal{A}$ *using the Type-7 or Type-8 strategy, there exist* QPT *algorithms* $\mathcal{B}_4$ *breaking the* IND-CCA *security of* KEM *and* $\mathcal{D}_3$ *breaking the security of PRF* F *such that*

$$\epsilon_1 \leq \mu^2 \ell \cdot \left(2\mathsf{Adv}_{\mathsf{KEM},\mathcal{B}_4}^{\mathsf{IND\text{-}CCA}}(\kappa) + \mathsf{Adv}_{\mathsf{F},\mathcal{D}_3}^{\mathsf{PRF}}(\kappa) + \varepsilon_{\mathsf{Ext}}\right) + \mu\ell^2 \cdot \frac{1}{2^{\chi_{\mathsf{KEM}}}}.$$

Proof of Lemma 3.6.9. We present the rest of the sequence of games from Game 1.

Game 2. This game is identical to Game 1, except that we add another abort condition. Let $\mathsf{E}_{\mathsf{coll}}$ be the event that there exists two responder oracles π_j^t and $\pi_j^{t'}$ for any $j \in [\mu]$ and $t \neq t' \in [\ell]$ such that they output the same KEM ciphertext. That is, there exists two oracles π_j^t and $\pi_j^{t'}$ that output $(\mathsf{C}, \mathsf{C}_T, \mathsf{c})$ and $(\mathsf{C}', \mathsf{C}_T', \mathsf{c}')$ such that $\mathsf{C} = \mathsf{C}'$. Here, we only consider the case where Pid_j^t and $\mathsf{Pid}_j^{t'}$ correspond to parties generated by the game (and not parties added by the adversary). If $\mathsf{E}_{\mathsf{coll}}$ occurs, then $\mathcal{C}$ aborts. Since Game 1 and Game 2 proceed identically unless $\mathsf{E}_{\mathsf{coll}}$ occurs, we have

$$|\epsilon_1 - \epsilon_2| \leq \Pr[\mathsf{E}_{\mathsf{coll}}].$$

We claim

$$\Pr[\mathsf{E}_{\mathsf{coll}}] \leq \mu\ell^2 \cdot \frac{1}{2^{\chi_{\mathsf{KEM}}}}.$$

Since each oracles π_j^t are initialized with uniform random and independent randomness and ek_i is honestly generated, where $i = \mathsf{Pid}_j^t$, each ciphertext C output by oracle π_j^t has χ_{KEM}-min entropy due to the χ_{KEM}-high ciphertext min-entropy of KEM. Fixing on one $j \in [\mu]$, the probability of a collision occurring is upper bounded by $\mu^2/2^{\chi_{\mathsf{KEM}}}$. Then, taking the union bound on all the parties, we obtain the claimed bound.

Game 3. In this game, before starting the game, $\mathcal{C}$ chooses a responder oracle $\pi_{\hat{j}}^{\hat{t}}$ and a party $P_{\hat{\imath}}$ uniformly at random from $\mu\ell$ oracles and μ parties, respectively. Let $\mathsf{E}_{\mathsf{testO}}$ be the event that the tested oracle is not $\pi_{\hat{j}}^{\hat{t}}$ or the peer of the tested oracle is not $P_{\hat{\imath}}$. Since $\mathsf{E}_{\mathsf{testO}}$ is an efficiently checkable event, $\mathcal{C}$ aborts as soon as it detects that event $\mathsf{E}_{\mathsf{testO}}$ occurs. $\mathcal{C}$ guesses the choice made by $\mathcal{A}$ correctly with probability $1/\mu^2\ell$, so we have

$$\epsilon_3 = \frac{1}{\mu^2\ell}\epsilon_2.$$

Game 4. In this game, we modify the way the initiator oracle π_i^s for any $s \in [\ell]$ responds on its second invocation. Let (K, C) be the KEM key-ciphertext pair generated by oracle $\pi_{\hat{\jmath}}^{\hat{t}}$. Then, when π_i^s is invoked (on the second time) on input $(\mathsf{C}', \mathsf{C}_T, \mathsf{c})$, it first checks if $\mathsf{C}' = \mathsf{C}$. If so, it proceeds as in the previous game except that it uses the key K that was generated by $\pi_{\hat{\jmath}}^{\hat{t}}$ rather than using the key obtained through decrypting C'. Otherwise, if $\mathsf{C}' \neq \mathsf{C}$, then it proceeds exactly as in the previous game. Conditioning on event $\mathsf{E}_{\mathsf{corr}}$ (i.e., decryption failure) not occurring, the two games Game 3 and Game 4 are identical. Hence,

$$\epsilon_4 = \epsilon_3.$$

Game 5. In this game, we modify the way the responder oracle $\pi_{\hat{\jmath}}^{\hat{t}}$ responds. When the responder oracle $\pi_{\hat{\jmath}}^{\hat{t}}$ is invoked on input ek_T, it samples a random key $\mathsf{K} \leftarrow\!\$\, \mathcal{KS}_{\mathsf{KEM}}$ instead of computing $(\mathsf{K}, \mathsf{C}) \leftarrow \mathsf{KEM.Encap}(\mathsf{ek}_{\hat{\imath}})$. Note that due to the modification we made in the previous game, when the initiator oracle π_i^s for any $s \in [\ell]$ is invoked (on the second time) on input $(\mathsf{C}', \mathsf{C}_T, \mathsf{c})$ for $\mathsf{C}' = \mathsf{C}$, it uses the random key K generated by oracle $\pi_{\hat{\jmath}}^{\hat{t}}$. We claim Game 4 and Game 5 are indistinguishable assuming the IND-CCA security of KEM. To prove this, we construct an algorithm $\mathcal{B}_4$ breaking the IND-CCA security as follows.

$\mathcal{B}_4$ receives a public parameter $\mathsf{pp}_{\mathsf{KEM}}$, a public key ek^*, and a challenge $(\mathsf{K}^*, \mathsf{C}^*)$ from its challenger. $\mathcal{B}_4$ then samples a random $(\hat{\imath}, \hat{\jmath}, \hat{t}) \leftarrow\!\$\, [\mu]^2 \times [\ell]$, sets up the public parameter of $\Pi_{\mathsf{SC\text{-}AKE}}$ using $\mathsf{pp}_{\mathsf{KEM}}$, and generates the long-term key pairs as follows. For party $P_{\hat{\imath}}$, $\mathcal{B}_4$ runs $(\mathsf{vk}_{\hat{\imath}}, \mathsf{sk}_{\hat{\imath}}) \leftarrow \mathsf{RS.KeyGen}(1^\kappa)$ and sets the long-term public key as $\mathsf{lpk}_{\hat{\imath}} := (\mathsf{ek}^*, \mathsf{vk}_{\hat{\imath}})$ and implicitly sets the long-term secret key as $\mathsf{lsk}_{\hat{\imath}} := (\mathsf{dk}^*, \mathsf{sk}_{\hat{\imath}})$, where note that $\mathcal{B}_{3,1}$ does not know dk^*. For all the other parties $i \in [\mu \backslash \hat{\imath}]$, $\mathcal{B}_4$ computes the long-term key pairs $(\mathsf{lpk}_i, \mathsf{lsk}_i)$ as in Game 5. Finally, $\mathcal{B}_4$ invokes $\mathcal{A}$ on input the public parameter of $\Pi_{\mathsf{SC\text{-}AKE}}$ and $\{\, \mathsf{lpk}_i \mid i \in [\mu] \,\}$ and answers the queries made by $\mathcal{A}$ as follows:

- $\mathtt{Send}(i, s, \langle \mathtt{START} : \mathsf{role}, j \rangle)$: $\mathcal{B}_4$ proceeds as in Game 5.
- $\mathtt{Send}(j, t, m = (\mathsf{ek}_T, \sigma_i))$: Let $i := \mathsf{Pid}_j^t$. Depending on the values of (j, t, i), it performs the following:
  - If $(j, t, i) = (\hat{\jmath}, \hat{t}, \hat{\imath})$, then $\mathcal{B}_4$ responds as in Game 5 except that it sets $(\mathsf{K}, \mathsf{C}) := (\mathsf{K}^*, \mathsf{C}^*)$ rather than generating them on its own. It then returns the message $(\mathsf{C}^*, \mathsf{C}_T, \mathsf{c})$.
  - If $(j, t, i) \neq (\hat{\jmath}, \hat{t}, \hat{\imath})$, then $\mathcal{B}_4$ responds as in Game 5.
- $\mathtt{Send}(i, s, m = (\mathsf{C}, \mathsf{C}_T, \mathsf{c}))$: Depending on the value of i, it performs the following:
  - If $i = \hat{\imath}$, then $\mathcal{B}_4$ checks if $\mathsf{C} = \mathsf{C}^*$. If so, it responds as in Game 5 except that it sets $\mathsf{K} := \mathsf{K}^*$. Otherwise, if $\mathsf{C} \neq \mathsf{C}^*$, then it queries the decapsulation oracle on C and receives back K'. $\mathcal{B}_{3,1}$ then responds as in Game 5 except that it sets $\mathsf{K} := \mathsf{K}'$.
  - If $i \neq \hat{\imath}$, then $\mathcal{B}_4$ responds as in Game 5.

- $\texttt{RevLTK}(i)$, $\texttt{RegisterLTK}(i)$, $\texttt{RevState}(i,s)$, $\texttt{RevSessKey}(i,s)$: $\mathcal{B}_4$ responds as in Game 5. Here, note that since $\mathcal{A}$ follows the Type-7 or Type-8 strategy, $\mathcal{B}_{3,1}$ can answer all the $\texttt{RevLTK}$-query. Namely, $\mathcal{A}$ never queries $\texttt{RevLTK}(\hat{\imath})$ (i.e., $\mathsf{lsk}_{\hat{\imath}} := (\mathsf{dk}^*, \mathsf{sk}_{\hat{\imath}})$) conditioning on $\mathsf{E}_{\mathsf{testO}}$ not occurring, which is the only query that $\mathcal{B}_{3,1}$ cannot answer.
- $\texttt{Test}(i,s)$: $\mathcal{B}_4$ responds to the query as in the definition. Here, in case $(i,s) \neq (\hat{\jmath}, \hat{t})$, then event $\mathsf{E}_{\mathsf{testO}}$ is triggered so it aborts.

If $\mathcal{A}$ outputs a guess b', $\mathcal{B}_4$ outputs 0 if $b' = b$ 1 otherwise. It can be checked that $\mathcal{B}_4$ perfectly simulates Game 4 (resp. Game 5) to $\mathcal{A}$ when the challenge K^* is the real key (resp. a random key). Thus we have

$$|\Pr[\mathsf{S}_4] - \Pr[\mathsf{S}_5]| \leq 2\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{KEM},\mathcal{B}_4}(\kappa).$$

Game 6. In this game, whenever we need to derive $\mathsf{K}_1^* \leftarrow \mathsf{Ext}_s(\mathsf{K}^*)$, we instead use a uniformly and randomly chosen PRF key $\mathsf{K}_1^* \leftarrow\!\!\$\, \mathcal{K}$ (fixed once and for all), where K^* is the KEM key chosen by oracle $\pi_{\hat{\jmath}}^{\hat{t}}$. Due to the modification we made in the previous game, K^* is chosen uniformly at random from $\mathcal{KS}_{\mathsf{KEM}}$ so K has $\log_2(|\mathcal{KS}_{\mathsf{KEM}}|) \geq \gamma_{\mathsf{KEM}}$ min-entropy. Then, by the definition of the strong $(\gamma_{\mathsf{KEM}}, \varepsilon_{\mathsf{Ext}})$-extractor Ext, we have

$$|\Pr[\mathsf{S}_5] - \Pr[\mathsf{S}_6]| \leq \varepsilon_{\mathsf{Ext}}.$$

Game 7. In this game, we sample a random function RF and whenever we need to compute $\mathsf{F}_{\mathsf{K}_1^*}(\mathsf{sid})$ for any sid, we instead compute $\mathsf{RF}(\mathsf{K}_1^*, \mathsf{sid})$. Due to the modification we made in the previous game, K_1^* is sampled uniformly from $\mathcal{K}$. Therefore, the two games can be easily shown to be indistinguishable assuming the pseudo-randomness of the PRF. In particular, we can construct a PRF adversary $\mathcal{D}_3$ such that

$$|\Pr[\mathsf{S}_6] - \Pr[\mathsf{S}_7]| \leq \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F},\mathcal{D}_3}(\kappa).$$

It remains to show that the session key outputted by the tested oracle in the Game 7 is uniformly random regardless of the challenge bit $b \in \{0,1\}$ chosen by the game. We consider the case where $b = 0$ and prove that the honestly generated session key by the tested oracle is distributed uniformly random. First conditioning on event $\mathsf{E}_{\mathsf{testO}}$ not occurring, it must be the case that the tested oracle $\pi_{\hat{\jmath}}^{\hat{t}}$ prepares the session key as $\mathsf{k}^* \| \tilde{\mathsf{k}} \leftarrow \mathsf{RF}(\mathsf{K}_1^*, \mathsf{sid}^*) \oplus \mathsf{F}_{\mathsf{K}_2}(\mathsf{sid}^*)$ for some sid^*. Here, recall K_1^* is the random PRF key sampled by the oracle $\pi_{\hat{\jmath}}^{\hat{t}}$ (see Game 6). Next, since the tested oracle has no partner oracle (by definition of the Type-7 and Type-8 strategy), there are no oracles π_i^s such that $i \neq i$ that runs $\mathsf{RF}(\mathsf{K}_1^*, \cdot)$ on input sid^*. Moreover, conditioning on event $\mathsf{E}_{\mathsf{coll}}$ not occurring, no oracles $\pi_{\hat{\imath}}^t$ for $t \neq \hat{t}$ run $\mathsf{RF}(\mathsf{K}_1^*, \cdot)$ on input sid^* as well since $(\mathsf{C}, \mathsf{C}_T)$ output by these oracles must be distinct from what $\pi_{\hat{\jmath}}^{\hat{t}}$ outputs. Therefore, we conclude that $\mathsf{RF}(\mathsf{K}_1^*, \mathsf{sid}^*)$ is only used to compute the session key of the tested oracle and used nowhere else. Since the output of RF is distributed uniformly random for different inputs, we conclude that $\Pr[\mathsf{S}_7] = 1/2$. Combining all the arguments together, we obtain

$$\epsilon_1 \leq \mu^2 \ell \cdot \left(2\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{KEM},\mathcal{B}_2}(\kappa) + \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F},\mathcal{D}_2}(\kappa) + \varepsilon_{\mathsf{Ext}}\right) + \mu\ell^2 \cdot \frac{1}{2^{\chi_{\mathsf{KEM}}}}.$$

□

For completeness, we state the remaining Lemmata 3.6.10 and 3.6.11 and provide a proof sketch.

Lemma 3.6.10. *For any* QPT *adversary* $\mathcal{A}$ *using the Type-1 or Type-2 strategy, there exist* QPT *algorithms* $\mathcal{B}_1$ *breaking the* IND-CPA *security of* wKEM *and* $\mathcal{D}_1$ *breaking the security of PRF* F *such that*

$$\epsilon_1 \leq \mu^2 \ell^2 \cdot \left(2\mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{wKEM},\mathcal{B}_1}(\kappa) + \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F},\mathcal{D}_1}(\kappa) + \varepsilon_{\mathsf{Ext}}\right).$$

Lemma 3.6.11. *For any* QPT *adversary* $\mathcal{A}$ *using the Type-3 or Type-4 strategy, there exist* QPT *algorithms* $\mathcal{B}_2$ *breaking the* IND-CCA *security of* KEM *and* $\mathcal{D}_2$ *breaking the security of PRF* F *such that*

$$\epsilon_1 \leq \mu^2 \ell \cdot \left(2\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{KEM},\mathcal{B}_2}(\kappa) + \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F},\mathcal{D}_2}(\kappa) + \varepsilon_{\mathsf{Ext}}\right) + \mu\ell^2 \cdot \left(\frac{1}{2^{2\chi_{\mathsf{KEM}}}} + \frac{1}{2^{\nu_{\mathsf{KEM}}}}\right).$$

Proof Sketch of Lemmata 3.6.10 and 3.6.11. The difference between $\Pi_{\mathsf{SC\text{-}DAKE}}$ and $\Pi_{\mathsf{SC\text{-}AKE}}$ is that the former uses a ring signature and the first message sent by the initiator includes the ephemeral verification key vk_T; and the initiator does not sign the first message. In addition, the former considers weak forward secrecy ($\mathcal{A}$ plays $\mathsf{Game}^{\mathsf{AKE\text{-}weakFS}}_{\Pi_{\mathsf{SC\text{-}DAKE}}}$), and the latter considers perfect forward secrecy ($\mathcal{A}$ plays $\mathsf{Game}^{\mathsf{AKE\text{-}FS}}_{\Pi_{\mathsf{SC\text{-}AKE}}}$). However, it can be easily verified that this modification brings no advantage to the adversary following the strategies in the statement. In particular, when $\mathcal{A}$ uses the Type-1, Type-2, Type-3 or Type-4 strategy (i.e., the tested oracle has a partner oracle), the winning condition (cf. freshness clauses Items 1 to 4) of the two security game is identical. Specifically, the proofs are identical to the proofs of Lemmata 3.3.4 and 3.3.5.

In slightly more detail, notice the session key derivation step in $\Pi_{\mathsf{SC\text{-}DAKE}}$ is exactly the same as those in $\Pi_{\mathsf{SC\text{-}AKE}}$. Namely, the value of the derived session key is independent of the signature conditioning on the signature being valid. Further, notice the proofs of Lemmata 3.3.4 and 3.3.5 only relies on the security properties of the KEM, PRF, and extractor. That is, the proof does not hinge on the security offered by the signature scheme and this holds even if the signature is removed from the first message and the signature scheme is replaced with a ring signature scheme. Here, we note that the validity of the ephemeral ring signature verification key never comes into play in the security proof. Therefore, the proofs of Lemmata 3.3.4 and 3.3.5 follow. □

Remark 3.6.12 (Why the Protocol $\Pi_{\mathsf{SC\text{-}DAKE}}$ Does Not Satisfy Full Forward Secrecy). For completeness, we show that $\Pi_{\mathsf{SC\text{-}DAKE}}$ does not satisfy full forward secrecy by constructing an adversary $\mathcal{A}$ that wins the game $\mathsf{Game}^{\mathsf{AKE\text{-}FS}}_{\Pi_{\mathsf{SC\text{-}DAKE}}}(\mu,\ell)$ with overwhelming probability. For simplicity, we consider the game with 2 parties P_1 and P_2 that establishes one session per party, i.e., $\mu = 2$ and $\ell = 1$. $\mathcal{A}$ performs the following attack:

1. $\mathcal{C}$ setups the oracles π^1_1 and π^1_2, and $\mathcal{A}$ obtains the public parameter and the long-term public keys lpk_1 and lpk_2 from the challenger $\mathcal{C}$.
2. $\mathcal{A}$ sends $\mathtt{Send}(2,1,m = \langle \mathtt{START} : \mathtt{resp}, 1\rangle)$ to $\mathcal{C}$, which initializes the oracle π^1_2 as the responder and sets its partner to P_1.
3. $\mathcal{A}$ generates the first message $(\mathsf{ek}_T, \mathsf{dk}_T)$ and $(\mathsf{vk}_T, \mathsf{sk}_T)$ according to the protocol description and sends $\mathtt{Send}(2,1,m = (\mathsf{ek}_T, \mathsf{vk}_T))$ to $\mathcal{C}$ and receives $(\mathsf{C}, \mathsf{C}_T, \mathsf{c})$ from $\mathcal{C}$. Note that π^1_2 terminates at this point.
4. $\mathcal{A}$ then corrupts P_1 by sending $\mathtt{RevLTK}(1)$ to $\mathcal{C}$ and receives P_1's long-term secret key $\mathsf{lsk}_1 = (\mathsf{dk}_1, \mathsf{sk}_1)$.
5. $\mathcal{A}$ decrypts C and C_T using dk_1 and dk_T, respectively. Then it computes the session key k and verifies the signature according to the protocol description.

6. $\mathcal{A}$ sends $\texttt{Test}(2,1)$ and receive k'. If $\mathsf{k} = \mathsf{k}'$, $\mathcal{A}$ outputs 0 as the guessed bit; otherwise outputs 1.

It is clear that $\mathcal{A}$ perfectly impersonates P_1 acting as an initiator. The session key k computed by $\mathcal{A}$ is the same session key computed by π_2^1, and thus, $\mathcal{A}$ can guess the challenge bit with overwhelming probability. Moreover, the tested oracle π_2^1 is fresh because $\mathcal{A}$ did not send `RevSessKey` and `RevState` queries, and it corrupted P_1 only after π_2^1 finished the protocol execution. Therefore, $\mathcal{A}$ is a valid adversary against the full forward secrecy game.

The following guarantees deniability of the proposed protocol $\Pi_{\mathsf{SC\text{-}DAKE}}$ against semi-honest adversaries.

Theorem 3.6.13 (Deniability of $\Pi_{\mathsf{SC\text{-}DAKE}}$ Against Semi-Honest Adversaries). *Assume* RS *is anonymous. Then, the Signal-conforming protocol* $\Pi_{\mathsf{SC\text{-}DAKE}}$ *is deniable against semi-honest adversaries.*[28]

Proof. Let $\mathcal{M}$ be any PPT semi-honest adversary. We explain the behavior of the simulator $\mathsf{SIM}_{\mathcal{M}}$ by considering three cases: (a) $\mathcal{M}$ initializes an initiator P_i, (b) $\mathcal{M}$ queries the initiator P_i on message $(\mathsf{C}, \mathsf{C}_T, \mathsf{c})$, and (c) $\mathcal{M}$ queries the responder P_j on message $(\mathsf{ek}_T, \mathsf{vk}_T)$. In case (a), $\mathsf{SIM}_{\mathcal{M}}$ runs the honest initiator algorithm and returns $(\mathsf{ek}_T, \mathsf{vk}_T)$ as specified by the protocol. In case (b), since $\mathcal{M}$ is semi-honest, we are guaranteed that it runs the honest responder algorithm to generate $(\mathsf{C}, \mathsf{C}_T, \mathsf{c})$. In particular, since $\mathcal{M}$ is run on randomness sampled by $\mathsf{SIM}_{\mathcal{M}}$, $\mathsf{SIM}_{\mathcal{M}}$ gets to learn the key K that was generated along with C. Therefore, $\mathsf{SIM}_{\mathcal{M}}$ runs the real initiator algorithm except that it uses K extracted from $\mathcal{M}$ rather than computing $\mathsf{K} \leftarrow \mathsf{KEM.Decap}(\mathsf{dk}_i, \mathsf{C})$. Here, note that $\mathsf{SIM}_{\mathcal{M}}$ cannot run the latter since it does not know the corresponding dk_i held by an honest initiator party P_i. In case (c), similarly to case (b), $\mathsf{SIM}_{\mathcal{M}}$ learns dk_T and sk_T used by $\mathcal{M}$ to generate ek_T and vk_T. Therefore, $\mathsf{SIM}_{\mathcal{M}}$ runs the honest responder algorithm except that it runs $\sigma \leftarrow \mathsf{RS.Sign}(\mathsf{sk}_T, \mathsf{sid}_j, \{\mathsf{vk}_T, \mathsf{vk}_j\})$ instead of running $\sigma \leftarrow \mathsf{RS.Sign}(\mathsf{sk}_j, \mathsf{sid}_j, \{\mathsf{vk}_T, \mathsf{vk}_j\})$ as in the real protocol. Here, note that $\mathsf{SIM}_{\mathcal{M}}$ cannot run the latter since it does not know the corresponding sk_j held by an honest responder party P_j.

Let us analyze $\mathsf{SIM}_{\mathcal{M}}$. First, for case (a), the output by $\mathsf{SIM}_{\mathcal{M}}$ is distributed exactly as in the real transcript. Next, for case (b), the only difference between the real distribution and $\mathsf{SIM}_{\mathcal{M}}$'s output distribution (which is the derived session key k) is that $\mathsf{SIM}_{\mathcal{M}}$ uses the KEM key K output by $\mathsf{KEM.Encap}$ to compute the session key rather than using the KEM key decrypted using $\mathsf{KEM.Decap}$ with the initiator party P_i's decryption key dk_i. However, by $(1 - \delta_{\mathsf{KEM}})$-correctness of KEM, these two KEM keys are identical with the probability at least $(1 - \delta_{\mathsf{KEM}})$. Hence, the output distribution of $\mathsf{SIM}_{\mathcal{M}}$ and the real view are indistinguishable. Finally, for case (c), the only difference between the real distribution and $\mathsf{SIM}_{\mathcal{M}}$'s output distribution (which is the derived session key and the message sent $(\mathsf{C}, \mathsf{C}_T, \mathsf{c})$) is how the ring signature is generated. While the real protocol uses the signing key sk_j of the responder party P_j, the simulator $\mathsf{SIM}_{\mathcal{M}}$ uses sk_T. However, the signatures outputted by these two distributions are computationally indistinguishable assuming the anonymity of RS. Hence, the output distribution of $\mathsf{SIM}_{\mathcal{M}}$ and the real view are indistinguishable.

Combining everything together, we conclude the proof. □

Remark 3.6.14 (Efficiency of $\Pi_{\mathsf{SC\text{-}DAKE}}$). We evaluate the message size of $\Pi_{\mathsf{SC\text{-}DAKE}}$. The first message of $\Pi_{\mathsf{SC\text{-}DAKE}}$ contains the additional ring signature verification key vk_T. Thus, the size of the first message increases by the amount of vk_T compared to $\Pi_{\mathsf{SC\text{-}AKE}}$.[29] The second message of $\Pi_{\mathsf{SC\text{-}DAKE}}$ contains the

[28] Although we only consider a classical adversary $\mathcal{M}$, it can be checked that the exact same proof holds even for a quantum adversary.

[29] To be fair, we compare $\Pi_{\mathsf{SC\text{-}DAKE}}$ with a variant of $\Pi_{\mathsf{SC\text{-}AKE}}$ who not sign the first message. Presented in [Has+21a], such variant is as secure as $\Pi_{\mathsf{SC\text{-}DAKE}}$ (modulo the difference between weak and perfect forward secrecy), and the main difference between the two schemes is deniability.

ring signature for a ring of two users instead of the standard signature. Examples of post-quantum ring signatures sizes for a ring of two users at the NIST security level 1 are 2.5 KiB (Raptor [LAZ19], based on NTRU), 4.4 KiB (DualRing [Yue+21], based on M-LWE/SIS), or 3.5 KiB (Calamari [BKP20], based on CSIDH). On the other hand, examples of standard signatures sizes at the same security level are 0.6 KiB (Falcon [Pre+20], based on NTRU), 2.3 KiB (Dilithium [Lyu+20], based on M-LWE/SIS) or 0.26 KiB (CSI-FiSh [BKV19], based on CSIDH). Therefore, when using ring signature schemes [LAZ19; BKP20; Yue+21], it is possible to get the size of the second message to be about 2-3 KiB larger than $\Pi_{\mathsf{SC\text{-}AKE}}$.

Remark 3.6.15 (Why Deniability Against a Semi-Malicious Adversary May Not Suffice). We provide a concrete attack on $\Pi_{\mathsf{SC\text{-}DAKE}}$ in case the adversary may act maliciously.[30] The scenario is as follows: Alice, the initiator in Figure 3.5, wants to prove that Bob, the responder in Figure 3.5, was engaging in communication with her. In the context of Signal, this means that Alice who uploads her (possibly maliciously generated) key package to the server wants to later prove that Bob was trying to communicate with her.

We consider a specific type of ring signature where for any public parameter $\mathsf{pp}_{\mathsf{RS}}$, the language of all possible verification key vk that can be output by $\mathsf{RS.KeyGen}(\mathsf{pp}_{\mathsf{RS}}, \cdot)$ is an $\mathsf{NP} \cap \mathsf{coNP}$ language. That is, if vk is in the image of $\mathsf{RS.KeyGen}(\mathsf{pp}_{\mathsf{RS}}, \cdot)$, then the randomness used to generate it will be the NP-witness, and if vk is not in the image, we assume there is a coNP-witness to prove it. We further assume the $\mathsf{NP} \cap \mathsf{coNP}$ language can be sampled efficiently along with an accompanying witness. Although it depends on the concrete set of parameters, many lattice-based ring signatures such as [BK10; LAZ19; Esg+19a; Esg+19b; Esg+19c; BKP20] where the verification key includes an LWE or NTRU instance could satisfy this property since the (approximated) gap closest vector GapCVP problem lies in $\mathsf{NP} \cap \mathsf{coNP}$ [AR04].

With such a ring signature, the attack is simple. Alice generates her ring signature verification key vk_A with an accompanying coNP-witness w_{no} to prove that vk_A does not have a corresponding secret key sk_A. This can informally be used as cryptographic evidence that justifies Alice's incapability of signing any message using vk_A. Now, if Bob generates a ring signature $\sigma \leftarrow \mathsf{RS.Sign}(\mathsf{sk}_B, \mathsf{sid}_B, \{\mathsf{vk}_A, \mathsf{vk}_B\})$ that includes Alice's verification key, then Alice can later claim to a third-party (e.g., a judge) by presenting $(\mathsf{vk}_A, w_{\mathsf{no}}, \sigma)$ that Bob engaged in a conversation with her. We note that to make this argument formal in a theoretical sense, Alice would also need to prove that any adversary that can output a valid σ can forge a signature using vk_B. That is, unless Alice knew Bob's verification key, she would not have been able to create σ. As a concrete example, we can consider the Raptor signature scheme [LAZ19]. Alice can set her verification key to be a zero-polynomial element. Then, the ring signature produced by Bob reduces to a standard signature created by Bob since the components that depend on Alice's verification key disappear.

Although informal, we believe there are several other simpler ways Alice may convince a real-world third-party her incapability of signing the ring signature provided by Bob. For instance, she may create the verification key using a cryptographically secure hash function, i.e., $\mathsf{vk}_A = \mathsf{H}(\mathsf{rand})$. Even if vk_A is not in $\mathsf{NP} \cap \mathsf{coNP}$, this may already be "good enough" evidence in practice to convince a third party that Alice could not have signed the message. This is because we can safely assume that computing the secret key from a random vk_A is infeasible.

Considering that the extent of cryptographic evidence that can be considered as real-world judicious evidence is unclear, Bob may want to be able to prove that Alice could have signed the ring signature in a cryptographically sound manner. We provide one possible way how to achieve this in the next section.

[30] The attack equally works for the subsequent protocol proposed by Brendel et al. [Bre+22]. We note that this does not contradict their security proof since the new definition of indistinguishability-based deniability they introduce does not capture malicious adversaries.

Remark 3.6.16 (Taking Advantage of the Asymmetry). As we explained at the beginning of this section, there is a subtle difference between the level of deniability we can target for a standard AKE protocol and the Signal handshake. Alice, the initiator in Figure 3.5, may not need to deny the fact that she was using the Signal app. In this case, Alice may be willing to sign her first message as in our original Signal-conforming AKE protocol $\Pi_{\mathsf{SC\text{-}AKE}}$. Such a signature allows us to prove perfect forward secrecy in Theorem 3.6.7 rather than weak forward secrecy while still providing deniability for the responder.

3.6.3 Deniable Signal-Conforming AKE $\Pi'_{\mathsf{SC\text{-}DAKE}}$ against Malicious Adversaries

We provide a Signal-conforming AKE protocol $\Pi'_{\mathsf{SC\text{-}DAKE}}$ that is secure even against malicious adversaries. The construction is provided in Figure 3.5. To achieve deniability against malicious adversaries, we modify our $\Pi_{\mathsf{SC\text{-}DAKE}}$ protocol so that the initiator party adds a NIZK proof attesting to the fact that it constructed the verification key of the ring signature vk_T honestly. Formally, we require the following additional building blocks.

Building Blocks. Our deniable Signal-conforming AKE protocol $\Pi'_{\mathsf{SC\text{-}DAKE}}$ against malicious adversaries requires the following primitives in addition to those required by $\Pi_{\mathsf{SC\text{-}DAKE}}$ in the previous section.

- $\mathsf{KEM} = (\mathsf{KEM.Setup}, \mathsf{KEM.KeyGen}, \mathsf{KEM.Encap}, \mathsf{KEM.Decap})$ is an IND-CCA secure KEM scheme as in the previous section that additionally satisfies PA--1 security with an efficiently constructible extractor, where μ is the number of parties in the system.
- $\mathsf{NIZK} = (\mathsf{NIZK.Setup}, \mathsf{NIZK.Prove}, \mathsf{NIZK.Verify})$ is a NIZK argument system for the relation $\mathcal{R}_{\mathsf{RS}}$ where $(\mathsf{X}, \mathsf{W}) \in \mathcal{R}_{\mathsf{RS}}$ if and only if the statement $\mathsf{X} = (\mathsf{pp}, \mathsf{vk})$ and witness $\mathsf{W} = (\mathsf{sk}, \mathsf{r})$ satisfy $(\mathsf{vk}, \mathsf{sk}) = \mathsf{RS.KeyGen}(\mathsf{pp}; \mathsf{r})$.

Additional Assumption. We require a knowledge-type assumption to prove deniability against malicious adversaries. Considering that all of the previous AKE protocols satisfying a strong form of security and deniability require such knowledge-type assumptions [DGK06; YZ10; Vat+20], this seems unavoidable. On the other hand, there are protocols achieving a strong form of deniability from standard assumptions [Dod+09; UG15; UG18], however, they make a significant compromise in the security such as being vulnerable to KCI attacks and state leakages.

The following knowledge assumption is defined similarly in spirit to those of Di Raimondo et al. [DGK06] that assumed that for any adversary $\mathcal{M}$ that outputs a valid MAC, then there exists an extractor algorithm Ext that extracts the corresponding MAC key. Despite it being a strong knowledge-type assumption in the standard model, we believe it holds in the random oracle model if we further assume the NIZK comes with an *online* knowledge extractor[31] like those provided by Fischlin's NIZK [Fis05]. We leave it to future works to investigate the credibility of the following assumption and those required to prove deniability of the X3DH protocol [Vat+20]. We also believe defining a more relaxed notion of deniability that still suffices in practice while also being satisfiable from standard assumptions to be of great importance.

Assumption 3.6.17 (Key-Awareness Assumption for $\Pi'_{\mathsf{SC\text{-}DAKE}}$). *We say that $\Pi'_{\mathsf{SC\text{-}DAKE}}$ has the* key-awareness property *if for all* PPT *adversaries $\mathcal{M}$ interacting with a real protocol execution in the deniability game as in Definition 3.6.1, there exists a* PPT *extractor $\mathsf{Ext}_{\mathcal{M}}$ such that for any choice of $(\mathsf{pp}, \mathsf{LPK}, \mathsf{LSK}) \in \mathsf{KeyGen}(1^\kappa, \mu)$, whenever $\mathcal{M}$ outputs a ring signature verification key vk and a* NIZK *proof π for the language $\mathcal{L}_{\mathsf{RS}}$, then $\mathsf{Ext}_{\mathcal{M}}$ taking input*

[31] This guarantees that the witness from a proof can be extracted without rewinding the adversary.

the same input as $\mathcal{M}$ (including its randomness) outputs a signing key sk *such that* $(\mathsf{vk}, \mathsf{sk}) \in \mathsf{RS.KeyGen}(\mathsf{pp}_{\mathsf{RS}})$ *for any* $\mathsf{pp}_{\mathsf{RS}} \in \mathsf{RS.Setup}(1^\kappa)$.

With the added building blocks along with the key-awareness assumption, we prove the following theorem. The high-level approach is similar to the previous proof against semi-honest adversaries, but the concrete proof required is rather involved. The main technicality is when invoking the PA--1 security: if we do the reduction naively, the extractor needs the randomness used to sample the ring signature key pairs of the honest party, but the simulator of the deniability game does not know such randomness. We circumvent this issue by hard-wiring the verification key of the ring signature of the adversary and considering PA--1 security against a non-uniform adversary.

Theorem 3.6.18 (Deniability of $\Pi'_{\mathsf{SC\text{-}DAKE}}$ against Malicious Adversaries). *Assume* KEM *is* PA--1 *secure with an efficiently constructible extractor,* RS *is anonymous,* NIZK *is sound* [32]*, and the key-awareness assumption in Assumption 3.6.17 holds. Then, the Signal-conforming protocol $\Pi'_{\mathsf{SC\text{-}DAKE}}$ with μ parties is deniable against malicious adversaries.*

Proof. The high-level idea of the proof is similar to those of Theorem 3.6.13. Below, we consider a sequence of simulators $\mathsf{SIM}_{\mathcal{M},i}$ where the first and last simulators $\mathsf{SIM}_{\mathcal{M},0}$ and $\mathsf{SIM}_{\mathcal{M},3}$ simulate the real and simulated protocols, respectively. That is, $\mathsf{SIM}_{\mathcal{M},3}$ is the desired simulator $\mathsf{SIM}_{\mathcal{M}}$. We define $\mathcal{F}_i$ to be the distribution of $(\mathsf{pp}, \mathsf{LPK})$ along with the output of $\mathsf{SIM}_{\mathcal{M},i}$. Our goal is to prove that $\mathcal{F}_0$ and $\mathcal{F}_3$ are indistinguishable.

- $\mathsf{SIM}_{\mathcal{M},0}$: It is given $(\mathsf{pp}, \mathsf{LPK}, \mathsf{LSK})$ as input and simulates the interaction with the adversary $\mathcal{M}$ following the protocol description of the real world. Here, note that $\mathcal{M}$ is invoked by $\mathsf{SIM}_{\mathcal{M},0}$ on input $(\mathsf{pp}, \mathsf{LPK})$ with uniform randomness. By definition $\mathsf{D}_{\mathsf{Real}} := \mathcal{F}_0$.

- $\mathsf{SIM}_{\mathcal{M},1}$: This is the same as $\mathsf{SIM}_{\mathcal{M},0}$ except that whenever $\mathcal{M}$ queries an honest responder party P_j on input $(\mathsf{ek}_T, \mathsf{vk}_T, \pi_T)$, $\mathsf{SIM}_{\mathcal{M},1}$ extracts the corresponding secret ring signature signing key sk_T. More formally, due to the key-awareness assumption of $\Pi'_{\mathsf{SC\text{-}DAKE}}$, for any PPT $\mathcal{M}$, there exists a PPT extractor $\mathsf{Ext}_{\mathcal{M}}$ such that whenever $\mathcal{M}$ outputs a ring signature verification key vk_T and a NIZK proof π_T for the language $\mathcal{L}_{\mathsf{RS}}$, then $\mathsf{Ext}_{\mathcal{M}}$ taking input the same input as $\mathcal{M}$ (including its randomness) outputs a signing key sk_T such that $(\mathsf{vk}_T, \mathsf{sk}_T) \in \mathsf{RS.KeyGen}(\mathsf{pp}_{\mathsf{RS}})$. Since $\mathsf{SIM}_{\mathcal{M},1}$ knows all the input and randomness fed to $\mathcal{M}$, it can run $\mathsf{Ext}_{\mathcal{M}}$. Namely, whenever $\mathcal{M}$ makes the above query, $\mathsf{SIM}_{\mathcal{M},1}$ invokes $\mathsf{Ext}_{\mathcal{M}}$ on input fed to $\mathcal{M}$ until that point along with its initial randomness and extracts sk_T. Since the output of $\mathsf{SIM}_{\mathcal{M},1}$ is unaltered, the distribution $\mathcal{F}_1$ is identical to the previous game. Below, for simplicity, we assume that $\mathcal{M}$ always outputs sk_T whenever it queries an honest responder party P_j on input $(\mathsf{ek}_T, \mathsf{vk}_T, \pi_T)$. This is without loss of generality since we can combine $\mathcal{M}$ and $\mathsf{Ext}_{\mathcal{M}}$ and view it as another adversary against the deniability game.

- $\mathsf{SIM}_{\mathcal{M},2}$: This is the same as $\mathsf{SIM}_{\mathcal{M},1}$ except that when $\mathcal{M}$ queries an honest responder party P_j on input $(\mathsf{ek}_T, \mathsf{vk}_T, \mathsf{sk}_T, \pi_T)$, $\mathsf{SIM}_{\mathcal{M},2}$ responds as in the real protocol except that it runs $\sigma \leftarrow \mathsf{RS.Sign}(\mathsf{sk}_T, \mathsf{sid}_j, \{\mathsf{vk}_T, \mathsf{vk}_j\})$ instead of running $\sigma \leftarrow \mathsf{RS.Sign}(\mathsf{sk}_j, \mathsf{sid}_j, \{\mathsf{vk}_T, \mathsf{vk}_j\})$. Due to the anonymity of the ring signature RS, the distributions $\mathcal{F}_1$ and $\mathcal{F}_2$ are indistinguishable.

Before explaining the next simulator, notice that we can view the combined algorithm $(\mathsf{SIM}_{\mathcal{M},2}, \mathcal{M})$ as a ciphertext creator $\mathcal{C}$ for the PA--1 security of the KEM scheme KEM. Formally, we decompose $\mathsf{SIM}_{\mathcal{M},2}$ into two algorithms: $\mathsf{SIM}'_{\mathcal{M},2}$ and $\mathsf{O}_{\mathsf{dec}}$, where $\mathsf{SIM}'_{\mathcal{M},2}$ is identical to $\mathsf{SIM}_{\mathcal{M},2}$ except that it outsources the decapsulation of ciphertexts corresponding to those of honest initiator parties to $\mathsf{O}_{\mathsf{dec}}$. That is, $\mathsf{SIM}'_{\mathcal{M},2}$ proceeds as

[32] We note that this is redundant since it is implicitly implied by the key-awareness assumption. We only include it for clarity.

$\mathsf{SIM}_{\mathcal{M},2}$ except that when $\mathcal{M}$ queries the honest initiator P_i on message $(\mathsf{C}, \mathsf{C}_T, \mathsf{c})$, it queries (i, C) to $\mathsf{O}_{\mathsf{dec}}$ to receive the corresponding KEM key K. Since $\mathsf{SIM}'_{\mathcal{M},2}$ no longer requires the secret KEM keys $\{ \mathsf{dk}_i \mid i \in [\mu] \}$ of the honest initiator parties, we can assume that $\mathsf{SIM}'_{\mathcal{M},2}$ only takes as input $(\mathsf{pp}, \{ \mathsf{ek}_i \mid i \in [\mu] \})$. Here, we also assume it has μ-ring signature verification keys $\{ \mathsf{vk}_i \mid i \in [\mu] \}$ hard-wired rather than $\mathsf{SIM}'_{\mathcal{M},2}$ generating it on its own. At this point, it is clear that the combined algorithm $(\mathsf{SIM}'_{\mathcal{M},2}, \mathcal{M})$ can be viewed as a valid ciphertext creator $\mathcal{C}$ that outputs the view of $\mathcal{M}$ as the string v, where $\mathsf{O}_{\mathsf{dec}}$ corresponds to the decapsulation oracle KEM.Decap run by the challenger in $\mathsf{Exp}^{\mathsf{dec}}_{\mathcal{C},\mathcal{D}}$. Then, by the PA--1 security, there must exist an extractor $\mathcal{E}_{\mathcal{C}}$ that simulates $\mathsf{O}_{\mathsf{dec}}$ that only takes as input $(\mathsf{pp}, (\mathsf{ek}_i)_{i\in[\mu]}, \mathsf{r}_{\mathcal{C}})$, where $\mathsf{r}_{\mathcal{C}}$ is the randomness used by $\mathcal{C}$ (i.e., by $(\mathsf{SIM}'_{\mathcal{M},2}, \mathcal{M})$). Moreover, such an extractor $\mathcal{E}_{\mathcal{C}}$ is efficiently constructible given the description of $\mathcal{C}$. Here, note that $\mathsf{r}_{\mathcal{C}}$ does *not* include the randomness used to generate the μ-ring signature verification keys since we hard-wire these to the description of $\mathsf{SIM}'_{\mathcal{M},2}$. In particular, $\mathcal{E}_{\mathcal{C}}$ does not require randomness used to generate LPK to be executed. We are now ready to define the next simulator.

$\mathsf{SIM}_{\mathcal{M},3} := \mathsf{SIM}_{\mathcal{M}}$: This is the same as $\mathsf{SIM}_{\mathcal{M},2}$ except that it constructs the extractor $\mathcal{E}_{\mathcal{C}}$ and when $\mathcal{M}$ queries the honest initiator P_i on message $(\mathsf{C}, \mathsf{C}_T, \mathsf{c})$ it runs $\mathcal{E}_{\mathcal{C}}(i, \mathsf{C})$ instead of $\mathsf{O}_{\mathsf{dec}}(\mathsf{dk}_i, \mathsf{C})$. Notice that $\mathsf{SIM}_{\mathcal{M},3}$ no longer requires any long-term secret key LSK to simulate $\mathcal{M}$. Due to the PA--1 security of the KEM scheme KEM, the two distributions $\mathcal{F}_2$ and $\mathcal{F}_3 := \mathsf{D}_{\mathsf{Sim}}$ are indistinguishable.

This completes the proof. □

Finally, it remains to show that the $\Pi'_{\mathsf{SC\text{-}DAKE}}$ is correct and secure as a standard Signal-conforming AKE protocol. Due to the correctness of Π_{NIZK}, the correctness of $\Pi'_{\mathsf{SC\text{-}DAKE}}$ follows from Theorem 3.6.6. Moreover, the security of $\Pi'_{\mathsf{SC\text{-}DAKE}}$ follows almost immediately from the proof of Theorem 3.6.7. The only difference is that in the proof of Lemma 3.6.8 (which is a sub-lemma used to prove Theorem 3.6.7), the reduction algorithm that does not know the corresponding signing key sk_T of the verification key vk_T invokes the zero-knowledge simulator to simulate the proof π_T. The rest of the proof is identical.

3.7 Equivalence Between Designated Verifier Signature and Ring Signature

In a subsequent work, Brendel et al. [Bre+22] showed a generic construction of a deniable Signal-conforming AKE protocol based on a designated verifier signature (DVS) and a KEM. They showed how to instantiate DVS from a ring signature (for a ring of two users) but left open the opposite implication and speculated the possibility of constructing DVS easier than a ring signature.

In this section, we solve this open problem. We show how to instantiate a ring signature (for a ring of two users) from DVS and show that DVS is a ring signature in disguise. As discussed in Footnote 8, the security notion of DVS and ring signatures may come in different flavors so it is not always the case that they are equivalent. We only focus on DVS and ring signatures that Brendel et al. [Bre+22] required to construct their AKE protocol. Namely, the definition of ring signature we provide in Section 2.7 is strictly stronger than those considered in [Bre+22]. We make this clear when we provide the security proof of our ring signature based on DVS.

The following syntax and security definition of DVS is taken almost verbatim from [Bre+22, Section 3]. One thing to keep in mind is that even though it is called *designated verifier*, the syntax of Brendel et al. allows the signature to be publicly verifiable. This will be essential when building a ring signature.

Definition 3.7.1. *A designated verifier signature (DVS) scheme* DVS *consists of a tuple of algorithms* $\mathsf{DVS} = (\mathsf{Setup}, \mathsf{SKGen}, \mathsf{VKGen}, \mathsf{Sign}, \mathsf{Vrfy}, \mathsf{Sim})$ *along with a message space* $\mathcal{M}$*.*

$\mathsf{Game}^{\text{RS-Unf}}_{\mathsf{DVS},\mathcal{A}}(\kappa)$

1 : $Q \leftarrow \emptyset$
2 : $L \leftarrow \emptyset$
3 : $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\kappa)$
4 : $(\mathsf{pk_S}, \mathsf{sk_S}) \leftarrow \mathsf{SKGen}(\mathsf{pp})$
5 : $(\mathsf{pk_D}, \mathsf{sk_D}) \leftarrow \mathsf{VKGen}(\mathsf{pp})$
6 : **for** $i \in [n]$ **do**
7 : $\quad(\mathsf{pk}_{\mathsf{D},i}, \mathsf{sk}_{\mathsf{D},i}) \leftarrow \mathsf{VKGen}(\mathsf{pp})$
8 : $\quad L \mathrel{+}\!\leftarrow (\mathsf{pk}_{\mathsf{D},i}, \mathsf{sk}_{\mathsf{D},i})$
9 : $(\mathsf{m}^*, \sigma^*) \leftarrow \mathcal{A}^{\mathtt{Sign}(\cdot,\cdot)}(\mathsf{pp}, \mathsf{pk_S}, \mathsf{pk_D}, L)$
10 : $d \leftarrow \mathsf{Vrfy}(\mathsf{pk_S}, \mathsf{pk_D}, \mathsf{m}^*, \sigma^*)$
11 : **return** $[\![d = 1 \wedge \mathsf{m}^* \notin Q]\!]$

Signing Oracle $\mathtt{Sign}(\mathsf{pk}, \mathsf{m})$

1 : **if** $(\mathsf{pk}, \mathsf{sk}) \notin L$ **then**
2 : $\quad$**return** $\perp$
3 : **if** $\mathsf{pk} = \mathsf{pk_D}$ **then**
4 : $\quad Q \mathrel{+}\!\leftarrow \mathsf{m}$
5 : $\sigma \leftarrow \mathsf{Sign}(\mathsf{sk_S}, \mathsf{pk}, \mathsf{m})$
6 : **return** σ

FIGURE 3.6: Security game for defining the unforgeability of DVS.

- $\mathsf{Setup}(1^\kappa) \to \mathsf{pp}$: *The setup algorithm takes a security parameter* 1^κ *as input and outputs a public parameter* pp. *In the following, we assume* pp *is provided to all the algorithms and may omit it for simplicity.*
- $\mathsf{SKGen}(\mathsf{pp}) \to (\mathsf{pk_S}, \mathsf{sk_S})$: *A probabilistic key generation algorithm that outputs a public-/secret-key pair for the* signer.
- $\mathsf{VKGen}(\mathsf{pp}) \to (\mathsf{pk_D}, \mathsf{sk_D})$: *A probabilistic key generation algorithm that outputs a public-/secret-key pair for the* verifier.
- $\mathsf{Sign}(\mathsf{sk_S}, \mathsf{pk_D}, \mathsf{m}) \to \sigma$: *A probabilistic signing algorithm that uses a signer's secret key* $\mathsf{sk_S}$ *to produce a signature* σ *for a message* $\mathsf{m} \in \mathcal{M}$ *for a designated verifier with public key* $\mathsf{pk_D}$.
- $\mathsf{Vrfy}(\mathsf{pk_S}, \mathsf{pk_D}, \mathsf{m}, \sigma) \to 1/0$: *A deterministic verification algorithm that checks a message* m *and a signature* σ *against a signer's public key* $\mathsf{pk_S}$ *and a verifier's public key* $\mathsf{pk_D}$.
- $\mathsf{Sim}(\mathsf{pk_S}, \mathsf{sk_D}, \mathsf{m}) \to \sigma$: *A probabilistic signature simulation algorithm that uses the verifier's secret key* $\mathsf{sk_D}$ *to produce a signature* σ *on a message* m *for signer's public key* $\mathsf{pk_S}$.

Definition 3.7.2 (Unforgeability). *We say that a DVS scheme* DVS *is* unforgeable *if for any efficient adversary* $\mathcal{A}$ *we have* $\Pr\left[\mathsf{Game}^{\text{RS-Unf}}_{\mathsf{DVS},\mathcal{A}}(\kappa) = 1\right] \leq \mathsf{negl}(\kappa)$, *where the game* $\mathsf{Game}^{\text{RS-Unf}}_{\mathsf{DVS},\mathcal{A}}$ *is defined in Figure 3.6.*

Definition 3.7.3 (Source Hiding). *We say that a DVS scheme* DVS *is* source hiding *if for any efficient adversary* $\mathcal{A}$ *we have* $\left|\Pr\left[\mathsf{Game}^{\text{RS-SrcHid}}_{\mathsf{DVS},\mathcal{A}}(\kappa) = 1\right] - 1/2\right| \leq \mathsf{negl}(\kappa)$, *where the game* $\mathsf{Game}^{\text{RS-SrcHid}}_{\mathsf{DVS},\mathcal{A}}$ *is defined in Figure 3.7.*

Using a standard hybrid argument, we can assume without loss of generality that $\mathcal{A}$ queries oracle `Chall` once.

Construction. We now provide a generic construction of a ring signature from any DVS schemes satisfying the above syntax and security definitions. We only consider a ring signature for a ring of two users as this is sufficient to construct an AKE protocol. Moreover, we assume without loss of generality that $\mathsf{pk_S}$ can be ordered lexicographically, e.g., $\mathsf{pk_S} < \mathsf{pk_S}'$.

$\mathsf{Game}^{\text{RS-SrcHid}}_{\mathsf{DVS},\mathcal{A}}(\kappa)$

1 : $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\kappa)$
2 : $(\mathsf{pk_S}, \mathsf{sk_S}) \leftarrow \mathsf{SKGen}(\mathsf{pp})$
3 : $(\mathsf{pk_D}, \mathsf{sk_D}) \leftarrow \mathsf{VKGen}(\mathsf{pp})$
4 : $b \leftarrow\!\$ \{0,1\}$
5 : $b' \leftarrow \mathcal{A}^{\mathtt{Chall}(\cdot)}(\mathsf{pp}, \mathsf{pk_S}, \mathsf{sk_S}, \mathsf{pk_D}, \mathsf{sk_D})$
6 : **return** $[\![b = b']\!]$

Challenge Oracle $\mathtt{Chall}(\mathsf{m})$

1 : **if** $b = 0$ **then**
2 : $\quad \sigma \leftarrow \mathsf{Sign}(\mathsf{sk_S}, \mathsf{pk_D}, \mathsf{m})$
3 : **else**
4 : $\quad \sigma \leftarrow \mathsf{Sim}(\mathsf{pk_S}, \mathsf{sk_D}, \mathsf{m})$
5 : **return** σ

FIGURE 3.7: Security game for defining the source hiding of DVS.

$\mathsf{RS.Setup}(1^\kappa)$: Run $\mathsf{pp_{DVS}} \leftarrow \mathsf{Setup}(1^\kappa)$ and output $\mathsf{pp_{RS}} := \mathsf{pp_{DVS}}$.

$\mathsf{RS.KeyGen}(\mathsf{pp_{RS}})$: Run $(\mathsf{pk_S}, \mathsf{sk_S}) \leftarrow \mathsf{SKGen}(\mathsf{pp_{DVS}})$ and $(\mathsf{pk_D}, \mathsf{sk_D}) \leftarrow \mathsf{VKGen}(\mathsf{pp_{DVS}})$, and output $(\mathsf{RS.vk} := (\mathsf{pk_S}, \mathsf{pk_D}), \mathsf{RS.sk} := (\mathsf{sk_S}, \mathsf{sk_D}))$.

$\mathsf{RS.Sign}(\mathsf{RS.sk}, \mathsf{m}, \mathsf{R} = \{\mathsf{RS.vk}, \mathsf{RS.vk}'\})$: Parse $(\mathsf{pk_S}, \mathsf{pk_D}) \leftarrow \mathsf{RS.vk}$ and $(\mathsf{pk_S}', \mathsf{pk_D}') \leftarrow \mathsf{RS.vk}'$. If $\mathsf{pk_S} < \mathsf{pk_S}'$, then output $\sigma \leftarrow \mathsf{Sign}(\mathsf{sk_S}, \mathsf{pk_D}', \mathsf{m})$. Otherwise, output $\sigma \leftarrow \mathsf{Sim}(\mathsf{pk_S}', \mathsf{sk_D}, \mathsf{m})$.

$\mathsf{RS.Verify}(\mathsf{R} = \{\mathsf{RS.vk}, \mathsf{RS.vk}'\}, \mathsf{m}, \sigma)$: Parse $(\mathsf{pk_S}, \mathsf{pk_D}) \leftarrow \mathsf{RS.vk}$ and $(\mathsf{pk_S}', \mathsf{pk_D}') \leftarrow \mathsf{RS.vk}'$. If $\mathsf{pk_S} < \mathsf{pk_S}'$, then output $\mathsf{Vrfy}(\mathsf{pk_S}, \mathsf{pk_D}', \mathsf{m}, \sigma)$. Otherwise, output $\mathsf{Vrfy}(\mathsf{pk_S}', \mathsf{pk_D}, \mathsf{m}, \sigma)$.

Security. We first prove the anonymity of the ring signature. The anonymity definition considered by Brendel et al. [Bre+22] is almost identical to those in Definition 2.7.3 except that they additionally consider the verification and signing keys to be generated honestly, rather than being generated by possibly malicious randomness. This suffices to prove their deniability since the AKE keys are assumed to be generated honestly.

Lemma 3.7.4. *If a DVS scheme* DVS *satisfies source hiding, then the ring signature scheme is anonymous (with respect to honestly generated verification and signing keys with rings of size two).*

Proof. Assume there exists an adversary $\mathcal{B}$ against the anonymity of the ring signature. We construct an adversary $\mathcal{A}$ against the source hiding of DVS as follows.

$\mathcal{A}$ is provided $(\mathsf{pp_{DVS}}, \mathsf{pk_S}, \mathsf{sk_S}, \mathsf{pk_D}, \mathsf{sk_D})$ from the DVS challenger. It queries m to oracle $\mathtt{Chall}$ and receives σ. It then generates $(\overline{\mathsf{pk_S}}, \overline{\mathsf{sk_S}}) \leftarrow \mathsf{SKGen}(\mathsf{pp_{DVS}})$ and $(\overline{\mathsf{pk_D}}, \overline{\mathsf{sk_D}}) \leftarrow \mathsf{VKGen}(\mathsf{pp_{DVS}})$ conditioned on $\mathsf{pk_S} < \overline{\mathsf{pk_S}}$. Note that this is without loss of generality since $\mathcal{A}$ can simply regenerate $\overline{\mathsf{pk_S}}$ until it succeeds (and possibly halt after it exceeds some number of trials to make $\mathcal{A}$ run in strict polynomial time). It then samples a random bit $d \leftarrow \{0,1\}$ and sets

$$(\mathsf{RS.vk}_d, \mathsf{RS.sk}_d) := ((\mathsf{pk_S}, \overline{\mathsf{pk_D}}), (\mathsf{sk_S}, \overline{\mathsf{sk_D}})) \text{ and } (\mathsf{RS.vk}_{1-d}, \mathsf{RS.sk}_{1-d}) := ((\overline{\mathsf{pk_S}}, \mathsf{pk_D}), (\overline{\mathsf{sk_S}}, \mathsf{sk_D})).$$

It finally provides $\mathcal{B}$ with $\{(\mathsf{RS.vk}_i, \mathsf{RS.sk}_i)\}_{i\in\{0,1\}}$ and σ. When $\mathcal{B}$ outputs d' as its guess, $\mathcal{A}$ outputs its guess as $b' := d \oplus d'$.

Let us analyze the advantage of $\mathcal{A}$. First of all, since d is information theoretically hidden from $\mathcal{B}$, the ring signature keys $\{(\mathsf{RS.vk}_i, \mathsf{RS.sk}_i)\}_{i\in\{0,1\}}$ are distributed identically to the anonymity game even conditioned on $\mathsf{pk_S} < \overline{\mathsf{pk_S}}$. Moreover, if oracle $\mathtt{Chall}$ was using $b = 0$, then $\sigma \leftarrow \mathsf{Sign}(\mathsf{sk_S}, \mathsf{pk_D}, \mathsf{m})$. Since $\mathsf{pk_S} < \overline{\mathsf{pk_S}}$, σ is distributed identical to $\mathsf{RS.Sign}(\mathsf{RS.sk}_d, \mathsf{m}, \mathsf{R} = \{\mathsf{RS.vk}_0, \mathsf{RS.vk}_1\})$. On the other

$\mathsf{Game}^{\mathsf{Mod\text{-}RS\text{-}Unf}}_{\mathsf{RS},\mathcal{A}}(\kappa, N)$

1 : $\mathsf{SL}, \mathsf{CL} \leftarrow \emptyset$
2 : $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\kappa)$
3 : **foreach** $i \in [N]$ **do**
4 : $\quad \mathsf{r}_i \leftarrow\!\$\, \mathcal{R}_{\mathsf{RS}}$
5 : $\quad (\mathsf{vk}_i, \mathsf{sk}_i) \leftarrow \mathsf{KeyGen}(\mathsf{pp}; \mathsf{r}_i)$
6 : $\mathsf{VK} := \{\, \mathsf{vk}_i \mid i \in [N] \,\}$
7 : $(i_0^*, i_1^*) \leftarrow\!\$\, [N] \times [N]$
8 : $(\mathsf{R}^*, \mathsf{m}^*, \sigma^*) \leftarrow \mathcal{A}^{\mathtt{Sign}(\cdot,\cdot,\cdot),\mathtt{Corr}(\cdot)}(\mathsf{pp}, \mathsf{VK})$
9 : $b_1 \leftarrow [\![\mathsf{R}^* = \{\, \mathsf{RS.vk}_{i_0^*}, \mathsf{RS.vk}_{i_1^*} \,\}]\!]$
10 : $b_2 \leftarrow [\![(*, \mathsf{m}^*, \mathsf{R}^*) \notin \mathsf{SL}]\!]$
11 : $b_3 \leftarrow [\![\mathsf{Verify}(\mathsf{R}^*, \mathsf{m}^*, \sigma^*) = 1]\!]$
12 : **return** $b_1 \wedge b_2 \wedge b_3$

Singing Oracle $\mathtt{Sign}(i, \mathsf{m}, \mathsf{R})$

1 : **if** $(\mathsf{vk}_i, \mathsf{vk}_j) \not\subseteq \mathsf{R}$ **then**
2 : $\quad$ **return** $\perp$
3 : **if** $i = i_0^*$ **then** $\sigma \leftarrow \mathsf{Sign}(\mathsf{sk}_j, \mathsf{m}, \mathsf{R})$
4 : **else** $\sigma \leftarrow \mathsf{Sign}(\mathsf{sk}_i, \mathsf{m}, \mathsf{R})$
5 : $\mathsf{SL} \mathrel{+}\!\leftarrow (i, \mathsf{m}, \mathsf{R})$
6 : **return** σ

Corruption Oracle $\mathtt{Corr}(i)$

1 : **req** $i \notin \{\, i_0^*, i_1^* \,\}$
2 : $\mathsf{CL} \mathrel{+}\!\leftarrow \mathsf{vk}_i$
3 : **return** r_i

FIGURE 3.8: Security game for defining unforgeability of ring signature schemes concerning honestly generated rings of size two. The modifications from the original Definition 2.7.4 are highlighted in gray. If the condition following **req** does not hold, the game terminates by returning 0.

hand, if oracle `Chall` was using $b = 1$, then $\sigma \leftarrow \mathsf{Sim}(\mathsf{pk}_\mathsf{S}, \mathsf{sk}_\mathsf{D}, \mathsf{m})$. Then, this is distributed identical to $\mathsf{RS.Sign}(\mathsf{RS.sk}_{1-d}, \mathsf{m}, \mathsf{R} = \{\, \mathsf{RS.vk}_0, \mathsf{RS.vk}_1 \,\})$. Hence, if $\mathcal{B}$ outputs a guess d' and $d = 0$, then $\mathcal{A}$ simply needs to output d' as its guess. Otherwise, $\mathcal{A}$ flips the guess d' in order not to compute the swap induced by $d = 1$. This completes the proof. □

The unforgeability definition considered by Brendel et al. [Bre+22] is similar to those in Definition 2.7.4 except that they restrict the adversary to only query the signing oracle on rings consisting of honestly generated verification keys. This weaker definition suffices for their application since they consider deniability only against honestly generated long-term keys.

Lemma 3.7.5. *If a DVS scheme* DVS *satisfies source hiding and unforgeability, then the ring signature is unforgeable (with respect to honestly generated rings of size two).*

Proof. Before providing the reduction, we first define the unforgeability game of the ring signature (concerning honestly generated rings of size two) as in Figure 3.8. The modification from the original Definition 2.7.4 is highlighted in gray. The difference is that the challenger samples two random distinct indices $(i_0^*, i_1^*) \in [N] \times [N]$ and hopes that the adversary outputs a forgery on the ring $\{\, \mathsf{RS.vk}_{i_0^*}, \mathsf{RS.vk}_{i_1^*} \,\}$. Moreover, whenever the adversary $\mathcal{A}$ queries the signing oracle, the challenger will never use $\mathsf{RS.sk}_{i_0^*}$ to sign the message.

It is straightforward to show that this modified unforgeability game is as hard as the original unforgeability game assuming that the ring signature is anonymous (which from Lemma 3.7.4 is an implication of the source hiding of DVS). Concretely, we first modify the original game to a game in which the challenger simply guesses the non-corrupted indices $(i_0^*, i_1^*) \in [N] \times [N]$ that the adversary will use for its forgery. Since these indices are information-theoretically hidden from the adversary, this is indistinguishable from

the original game (except for a loss of $1/N^2$ in the reduction). Next, assuming $\mathcal{A}$ makes at most Q-queries to the signing oracle, we can define Q-hybrids, where in the k-th hybrid, the challenger answers as in the original game up to the $(k-1)$-th signing query and as in the modified game from the k-th signing query. Each adjacent hybrid $(k-1)$ and k are indistinguishable assuming the anonymity of the ring signature; the reduction samples a random index $j^* \leftarrow [N]$ and embeds its two verification keys provided by the anonymity game in the two indices (i_0^*, j^*). It generates all other verification keys as in the unforgeability game. Note that the reduction knows the signing keys to all parties. It answers all k'-th signing query for $k' \neq k$ as in hybrids $(k-1)$ and k. If $i = i_0^*$ is used in the k-th query, it further checks if vk_{j^*} is used. If so, the reduction simulates the signing oracle by embedding its challenge. If vk_{j^*} is not used, then aborts the game. Otherwise, if $i \neq i_0^*$, then it answers the signing oracle as in the $(k-1)$th and k-th hybrids. This completes the reduction. In case the signature is created using $\mathsf{sk}_{i_0^*}$ (resp. sk_j), it perfectly simulates the $(k-1)$-th (resp. k-th) hybrid (condition on not aborting). Therefore, assuming the ring signature is anonymous, the two hybrids are indistinguishable.

Now, we are ready to show that this modified unforgeability for the ring signature is hard assuming the unforgeability of the DVS. Assume there exists an adversary $\mathcal{B}$ against the modified unforgeability of the ring signature. We construct an adversary $\mathcal{A}$ against the unforgeability of DVS as follows:

$\mathcal{A}$ is given $\mathsf{pp}_{\mathsf{DVS}}$, pk_{S}, pk_{D}, and $L = \{ (\mathsf{pk}_{\mathsf{D},i}, \mathsf{sk}_{\mathsf{D},i}) \}_{i\in[n]}$. It generates $(\overline{\mathsf{pk}_{\mathsf{D}}}, \overline{\mathsf{sk}_{\mathsf{D}}}) \leftarrow \mathsf{VKGen}(\mathsf{pp}_{\mathsf{DVS}})$, $(\overline{\mathsf{pk}_{\mathsf{S}}}, \overline{\mathsf{sk}_{\mathsf{S}}}) \leftarrow \mathsf{SKGen}(\mathsf{pp}_{\mathsf{DVS}})$, and $(\mathsf{pk}_{\mathsf{S},i}, \mathsf{sk}_{\mathsf{S},i}) \leftarrow \mathsf{SKGen}(\mathsf{pp}_{\mathsf{DVS}})$ for $i \in [n]$. It then creates $(n+2)$ pairs of verification key pair for the ring signature $(\mathsf{pk}_{\mathsf{S}}, \overline{\mathsf{pk}_{\mathsf{D}}})$, $(\overline{\mathsf{pk}_{\mathsf{S}}}, \mathsf{pk}_{\mathsf{D}})$, and $\{ (\mathsf{pk}_{\mathsf{S},i}, \mathsf{pk}_{\mathsf{D},i}) \}_{i\in[n]}$ and randomly permutes them and sets them as $(\mathsf{RS.vk}_i)_{i\in[n+2]}$. Let i_0^* be the index such that $\mathsf{RS.vk}_{i_0^*} := (\overline{\mathsf{pk}_{\mathsf{S}}}, \mathsf{pk}_{\mathsf{D}})$ and i_1^* be the index such that $\mathsf{RS.vk}_{i_1^*} := (\mathsf{pk}_{\mathsf{S}}, \overline{\mathsf{pk}_{\mathsf{D}}})$. $\mathcal{A}$ finally provides $\mathsf{VK} := \{ \mathsf{RS.vk}_i \mid i \in [n+2] \}$ to $\mathcal{B}$. Notice $\mathcal{A}$ knows the signing keys for indices in $[n+2] \setminus \{ i_0^*, i_1^* \}$, so it can simulate the signing query and corrupt query for any $i \notin \{ i_0^*, i_1^* \}$. Moreover, since $\mathcal{A}$ aborts when $i \in \{ i_0^*, i_1^* \}$ is queried to the corruption oracle, it remains to see how $\mathcal{A}$ simulates the signing queries when $i \in \{ i_0^*, i_1^* \}$. Due to the modification we made to the unforgeability game, $\mathcal{A}$ never needs to sign using the signing key corresponding to index i_0^*, so it suffices to check the case $i = i_1^*$. Now, when $i = i_1^*$ and $\mathsf{pk}_{\mathsf{S}} < \mathsf{pk}_{\mathsf{S},j}$, then $\mathcal{A}$ queries its signing oracle and obtains a signature using sk_{S}. Otherwise, it uses $\overline{\mathsf{sk}_{\mathsf{D}}}$ to generate $\sigma \leftarrow \mathsf{Sim}(\mathsf{pk}_{\mathsf{S}_j}, \overline{\mathsf{sk}_{\mathsf{D}}}, \mathsf{m})$. This completes the description of $\mathcal{A}$.

Notice the winning condition of the modified unforgeability of the ring signature and the unforgeability of the DVS is identical. Moreover, since $\mathcal{A}$ randomly permutes the indices in $[n+2]$, $\mathcal{A}$ simulates the distribution of the two indices (i_0^*, i_1^*) perfectly. Therefore, $\mathcal{A}$ has the same advantage as $\mathcal{B}$. This concludes the proof. □

Chapter 4

Continuous Group Key Agreement via Post-Quantum Multi-Recipient PKEs[1]

4.1 Introduction

4.1.1 Background

In a secure (group) conversation over e.g., Signal, the session may last years, there may be hundreds of users, and they may not be online simultaneously. This stands in stark contrast to a TLS session, which is bounded in time and deals with two online users (server and client). It also raises new security issues. For a crude example, consider a conversation involving N participants over a span of t units of time. If each participant has an independent probability ϵ of being compromised over a unit of time, this conversation will have its contents compromised with probability $1 - (1 - \epsilon)^{Nt}$, which becomes significant as soon as $Nt = \Omega(1/\epsilon)$. This issue can be resolved by having each participant refresh their key material at a regular pace, thus limiting the scope of a compromise. This practice, called *ratcheting*, provides post-compromise security (PCS) and forward secrecy (FS) [CCG16; Coh+17; ACD19]. It also forms the basis for more sophisticated techniques [BBR18; Alw+20a; Alw+20b] providing various levels of a stronger notion called post-compromise forward secrecy (PCFS) [Alw+20a; Alw+20b; AJM22].

Continuous Group Key Agreement. The notions of continuous (group) key agreement (CKA and CGKA) were put forward [ACD19; Alw+20a; Alw+20b; AJM22; Kle+21] to capture the particular setting that secure (group) messaging contends with, e.g., asynchrony and large groups, and achieve the security notions it requires, e.g., PCFS. In addition to representing a clean abstraction, CGKAs also include the complex cryptographic machinery of secure group messaging, and are therefore convenient objects to reason on.

The most widely academically discussed CGKA is TreeKEM [BBR18]. It underlies the IETF draft standard for secure messaging, MLS [Oma+21; Bar+20]. TreeKEM derives its name from its use of *ratchet trees* (bottom left of Figure 4.1, p. 81), and a significant amount of research and engineering effort has been undertaken to study the efficiency and security implications of this signature feature [Kle+21; Alw+20a; Wei19; AJM22; Alw+20b; BBN19a].

The most recent iterations of TreeKEM (i.e., after version 8 on MLS) follow a "*propose-and-commit*" flow, in which members of a group may propose to add new members, remove existing ones or update their own keys, by sending *proposal messages*. These proposals only take effect when a group member initiates a new epoch by transmitting a *commit message*, which simultaneously validates a list of indicated proposals.

[1]The contents of this chapter are based on the work presented at ACM CCS 2021 under the title "A Concrete Treatment of Efficient Continuous Group Key Agreement via Multi-Recipient PKEs" [Has+21b]. The full version is available at the IACR Cryptology ePrint Archive. [Has+21c].

Bandwidth and Commit Messages. In order to realize PCFS, commit messages in TreeKEM include $\lceil \log N \rceil$ encryption keys and at least as many ciphertexts[2] (see Figure 4.1), where $\log x$ denotes the logarithm in base 2 of x. As group members are arranged as the leaves of a binary tree, these encryption keys and ciphertexts allow all recipients to derive a fresh common group secret comSecret (*commit secret*), which is the root of the tree.

Let us discuss bandwidth consumption through three metrics: the cost of an upload and download, and the total cost. We focus on the bandwidth cost of the commit messages of TreeKEM, as they are the dominant term. Indeed, commit messages are the only cryptographically-heavy messages that need to be uploaded and downloaded at a regular rate, and each of them has a size of $\Omega(\log N)$. This therefore represents both the *upload* and *download* cost. If each member of a group sends a single commit message in a given time span, then they each must also download $(N-1)$ commit messages, for a *total* bandwidth cost of $\Omega(N \log N)$ per user.[3] For large groups, this can become significant. Ironically, large groups are also those that need the PCFS provided by commit messages the most, since their likelihood of compromise during a time span is higher.

This tension between security and bandwidth efficiency can be amplified by two factors: post-quantum cryptography, and the fact that secure messaging applications target mobile devices. In general, post-quantum cryptographic primitives consume more bandwidth than their classical counterparts by at least an order of magnitude, if not more: for example, all parameter sets of Classic McEliece entail encapsulation keys of at least 255 kibibytes (KiB). On the other hand, bandwidth can be a scarce resource over mobile devices, especially for users with limited mobile plans that charge an extra fee or block access to the network once a data cap has been reached.[4] To give a concrete example, instantiating TreeKEM with Classic McEliece in a group of $N = 256$ members will deplete a 1 GiB mobile plan once each user has sent *two* commit messages. This motivates the need for CGKA protocols and post-quantum primitives that remain efficient and secure for large groups. We note that mobile plan providers typically calculate data usage by treating uploaded and downloaded data as equal, and that being temporarily blocked from, or asked to pay more to continue to access, the mobile infrastructure is perhaps the most significant way in which bandwidth usage affects user experience. Hence, our bandwidth cost model: *downloading one-byte costs as much as uploading one byte*.

One could argue that assigning different weights to uploaded and downloaded data would be more appropriate, since uploading speed may be lower than downloading speed [Spe21]. We believe this speed-based distinction is not necessary, for two reasons. First, the bandwidth bottleneck of our CGKA resides in commit messages, which are uploaded and downloaded in a manner that is invisible to end users. Second, all our instantiations of the proposed protocol achieve uploaded commit messages of less than 50KiB for groups of at most 1024 users (see Figure 4.27), which, even in countries with low uploading speed (as of July 2021, the slowest is Afghanistan, with 2.90 Mbps [Spe21]), can be uploaded in less than 0.2 seconds. Both facts point to a minimal impact of uploading and downloading speeds on user experience.

[2] A documented property [Kle+21] of TreeKEM is that the number n_c of ciphertexts depends on the topology of the ratcheting tree, which might contain *blank* nodes. This number is $\lceil \log N \rceil$ in the best case but may degrade to $N-1$ for heavily blanked trees.

[3] Downloading and processing commit messages is important for security *and* functionality: a member refusing to download a commit message will be unable to decrypt subsequent messages.

[4] Surveys on mobile data pricing [Cab21] are interesting in that regard. The median cost of 1 GiB of mobile data is on average (across all countries) $4.07. Mobile plans that cost more than $20.00 / GiB are reported in 89 countries and, in expensive countries, "*People are often buying data packages of just a tens of megabytes at a time*" [Cab21]. This illustrates that mobile data can be a limited and expensive resource.

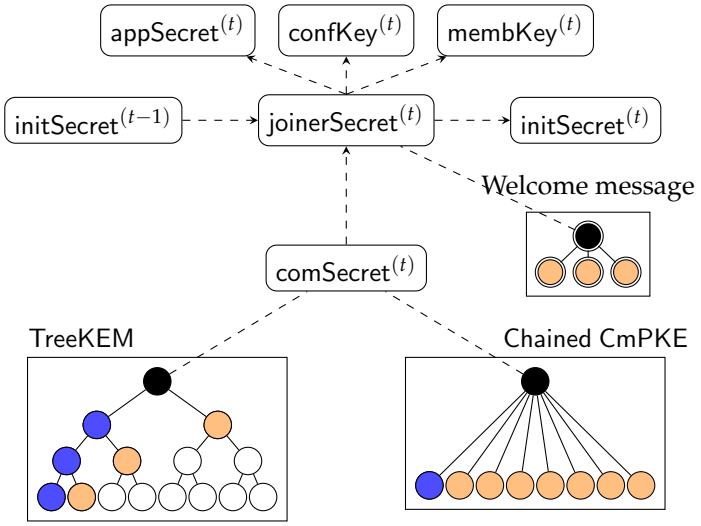


FIGURE 4.1: Initialization of a new epoch t, here with a group of $N = 8$ members. A dashed arrow from X to Y means that Y is computed by passing X (and possibly other values) to a HKDF, a dashed line means that $X = Y$. Here, the leftmost user in the TreeKEM (resp. Chained CmPKE) box initiates a new epoch by issuing a commit message, which contains one encryption key for each node, and one PKE (resp. CmPKE) ciphertext for each node. Each recipient in the current group is able to compute comSecret$^{(t)}$, which corresponds to the root. A commit message may include a welcome message, which contains ciphertexts encrypting joinerSecret$^{(t)}$ under the encryption key of each newly added member.

4.1.2 Contribution of This Work

We propose a new CGKA called Chained CmPKE along with a formal security proof (Section 4.4). The main technical tools we leverage are the existence of very efficient post-quantum multi-recipient PKEs (mPKE, Section 4.5), and the notion of a committing mPKE (CmPKE, Section 4.2). We believe these tools may be of independent interest.

The Chained CmPKE Protocol. At a very high level, the proposed protocol is inspired by the Chained mKEM protocol [BBN19a; Beu20]. One way of interpreting Chained mKEM is as TreeKEM with a tree of arity N and depth 1. This makes the size of *uploaded* commit messages scale as $O(N)$, see the bottom right of Figure 4.1. The main conceptual difference between our Chained CmPKE[5] and variations of TreeKEM (including Chained mKEM) is that we no longer consider the delivery service as a public bulletin board, and instead allow it to sanitize commit messages in a straightforward manner by delivering to each group member the strict amount of data they need while maintaining the same level of (dis)trust. In our case, this means that a member i may only receive the ciphertext ct_i that they can decrypt.

Our first line of research realizes this sanitizability by authenticating all ciphertexts with a single signature. To achieve this we rely on the notion of a CmPKE, essentially a multi-recipient PKE augmented with a commitment T. (Section 4.1.2). CmPKEs allow us to reduce the size of *downloaded* commit messages from $O(N)$ to $O(1)$. Effectively, this also reduces the *total bandwidth* cost of transmitting a commit message to $O(N)$, instead of $O(N^2)$ for Chained mKEM and $\Omega(N \log N)$ for TreeKEM. Alternatively, one could

[5]We consciously use the term Chained CmPKE rather than Chained CmKEM since we believe PKE better reflects the protocol description.

use a Merkle tree to authenticate all ciphertexts, as in Certificate Transparency [Lau+21]. However, each downloaded commit message would need to include a membership proof of size $O(\log N)$, in contrast to our $O(1)$ solution.

In the second line of research, we minimize the concrete cost of *uploaded* commit messages, which is $O(N)$ and larger than $\Omega(\log N)$ as for TreeKEM, by proposing and analyzing new efficient post-quantum mPKEs. (Section 4.1.2). As we show in Section 4.2, we can generically transform any mPKEs into CmPKEs with minimal overhead, thus we simply focus on mPKEs.

Compared to a naive instantiation of mPKEs using standard single-recipient PKEs, our mPKEs make the commit messages asymptotically smaller (in N) by factors of between 16 (Kyber512 vs. Ilum512) and 71 (Frodo640 vs. Bilbo640). In fact, while our *uploading* cost scales asymptotically as $O(N)$, it still compares favorably to the $\Omega(\log N)$ solution of TreeKEM in *concrete* efficiency, even for groups with hundreds of users.

Our bandwidth savings are summarized in an asymptotic manner in (Table 4.1, p. 111), and in a concrete manner in (Figure 4.27, p. 182) and (Figure 4.28, p. 184). While Figure 4.27 illustrates the upload and download cost, Figure 4.28 illustrates the total bandwidth cost. Compared to TreeKEM-based equivalents, our instantiations of Chained CmPKE have consistently better *upload* costs for groups of less than 200 users indicating that $O(N)$ solutions can be practically efficient. In addition, our *download* and *total* costs are better by factors of $\Omega(\log N)$ and performs well for any number of users.

Committing mPKEs. We introduce the notion of a *committing* mPKE, or CmPKE. First, a (decomposable) multi-recipient PKE (mPKE) [Kat+20] takes as input a message m and a list of N encryption keys, and outputs a multi-recipient ciphertext $(\mathsf{ct}_0, (\widehat{\mathsf{ct}}_i)_{i\in[N]})$. Each recipient $i \in [N]$ is able to recover m by decrypting $(\mathsf{ct}_0, \widehat{\mathsf{ct}}_i)$. The syntax of a CmPKE is mostly similar to that of an mPKE, however, it requires one additional component. The encryption procedure of a CmPKE outputs $(\mathsf{T}, (\mathsf{ct}_i)_{i\in[N]})$, where T is called a *commitment*. Decryption then works by taking the commitment-ciphertext pair $(\mathsf{T}, \mathsf{ct}_i)$. We require T to (a) have a size *independent* of the number of recipients N, and (b) be *commitment-binding*, which means informally that T is bound to a unique message. This notion resembles committing AEADs [GLR17], however we operate in a different setting (multi-user vs. single-user) and with a different motivation (bandwidth efficiency vs. abuse reporting).

We show how to build a CmPKE from an mPKE [Kat+20] and a *key-committing* SKE [FOR17; GLR17; Dod+18; Alb+22], which can itself be built using standard symmetric primitives [Alb+22]. Compared to the base mPKE, the overhead is minimal: $\mathsf{ct}_i = \widehat{\mathsf{ct}}_i$, and T is formed of ct_0 and a term of size 2κ bits, which is no larger than a hash digest.

In the proposed protocol, after computing a CmPKE ciphertext $(\mathsf{T}, \vec{\mathsf{ct}} = (\mathsf{ct}_i)_{i\in[N]})$, the sender of a commit message does not authenticate the whole ciphertext, only T. The server sends $(\mathsf{T}, \mathsf{ct}_i)$ to each recipient i, and the commitment-binding property allows i to indirectly verify the authenticity of the message encrypted in ct_i. As a result, the download cost of a commit message is $O(1)$ for all recipients.

More Efficient mPKEs. An mPKE allows one to encrypt a common message to N recipients more efficiently than the naive solution of computing and sending N individual ciphertexts in parallel. Indeed, as each recipient receives $(\mathsf{ct}_0, \widehat{\mathsf{ct}}_i)$, mPKEs provide asymptotic bandwidth savings if $\widehat{\mathsf{ct}}_i$ is smaller than a regular, single-recipient ciphertext ct would be.

While mPKEs based on classical assumptions [Kur02; BBS03] realize $|\widehat{\mathsf{ct}}_i|/|\mathsf{ct}| = 1/2$, existing PKEs based on the post-quantum problems LWE, LWR, SIDH and CSIDH were recently adapted to the mPKE setting in [Kat+20]. These mPKEs achieve ratios $|\widehat{\mathsf{ct}}_i|/|\mathsf{ct}|$ between $1/5$ and $1/169$, which could potentially translate into inversely proportional bandwidth savings. The work of [Kat+20] has two shortcomings; (a)

their mPKEs are direct transpositions of existing PKEs, which were not necessarily designed to minimize $|\widehat{\mathsf{ct}}_i|$, and (b) it does not study the concrete impact of the mPKE setting on cryptanalysis. We address these two shortcomings via a two-pronged approach.

On the constructive side, we note that minimizing the size of uploaded commit messages gives a different optimization target to that of PKEs, specifically we wish to minimize $|\widehat{\mathsf{ct}}_i|$, even at the expense of some controlled growth of $|\mathsf{ct}_0|$. We, therefore, attempt to improve upon the efficiency gains already reported in [Kat+20] by revisiting the designs of the NIST submissions [Nae+20; Sch+20; Ber+20] with our new optimization target in mind. To achieve this we rely on well-known techniques such as coefficient dropping and modulus rounding. We arrive at three new parametrizations; a variant of Frodo640 [Nae+20] called Bilbo640, a variant of Kyber512 [Sch+20] called Ilum512, and a variant of LPRime653 [Ber+20] called LPRime757. Compared to using the NIST submissions as mPKEs we reduce $|\widehat{\mathsf{ct}}_i|$ by 60–80%, which translates to an identical asymptotic reduction in the size of uploaded commit messages. These parameterizations are close to optimal in the sense that $|\widehat{\mathsf{ct}}_i| \in (\kappa, 3\kappa]$ bits. Since in the Lindner–Peikert framework, $\widehat{\mathsf{ct}}_i$ encodes all the information about the message (in our case, a κ-bit symmetric key), it seems difficult to beat the κ-bit threshold without new techniques.

On the cryptanalytic side, we must consider the effect of the mPKE setting on the attack surface. In [Kat+20] theoretical, reduction-based, assurances for the security of the mPKE construction are given. However, the concrete security of the mPKEs derived from NIST submissions is assumed to follow from their concrete security analyses as PKEs. As an example of differences between the two settings, variants of the Arora–Ge [AG11; Alb+15a] and BKW [BKW00] attacks are typically irrelevant to lattice-based PKEs, since they require more 'samples' than provided by the single ciphertext of the PKE, ct. However, in the mPKE setting, the *per recipient* $\widehat{\mathsf{ct}}_i$ ciphertext components each provide samples for an adversary. Therefore the Arora–Ge and BKW attacks should be considered in a concrete security analysis of mPKE parameters. In Section 4.5.3, we describe these attacks in more detail and provide estimates for the concrete security of our reparametrizations in a cryptanalytic model tailored to the mPKE setting. This model targets NIST Security Level I. Schemes satisfying this are conjectured to have comparable security to AES-128 against classical and quantum adversaries. Interestingly, our attempts to improve the efficiency of our mPKEs via re-parametrizing NIST submissions, specifically our use of heavy modulus rounding on the $\widehat{\mathsf{ct}}_i$, naturally *hardens* our parametrizations against these sample-heavy attacks. To display the importance of an mPKE-focused cryptanalysis, we provide an artificial 'Kyber like' parameter set that is *almost* secure as a PKE, but insecure in our mPKE cryptanalytic model.

Security of Chained CmPKE**.** Finally, we provide formal proof establishing that our Chained CmPKE is as secure as TreeKEM. We adopt the state-of-the-art UC security model presented by Alwen et al. [AJM22] that was used to analyze the TreeKEM version 10 in MLS, which is itself an extension of [Alw+20b]. In addition to party corruptions (i.e., compromise party's secret and group secrets), the model captures *active adversaries* who may tamper with or inject messages and deliver messages in an arbitrary order, and *malicious insiders* who may interact with the PKI on behalf of the corrupted parties. On a technical front, to model the sanitizing of the commit messages by the delivery service, we extend the ideal functionality in [AJM22] and modify how the ideal functionality maintains the so-called *history graph*. Our security model is a strict generalization of prior models as it captures them as special cases.

4.2 Committing Multi-Recipient PKE

We consider strengthening of a standard mPKE which we coin a *committing* mPKE (CmPKE). The motivation for this is similar in spirit to those of key committing SKEs or AEADs [FOR17; GLR17; Dod+18; Alb+22],

$\mathsf{Game}^{\mathsf{IND\text{-}CCA}}_{\mathsf{CmPKE},\mathcal{A}}(\kappa)$

1 : $C \leftarrow \emptyset$
2 : $\mathsf{pp} \leftarrow \mathsf{CmSetup}(1^\kappa)$
3 : **foreach** $i \in [N]$ **do**
4 : $\quad(\mathsf{ek}_i, \mathsf{dk}_i) \leftarrow \mathsf{CmGen}(\mathsf{pp})$
5 : $(\mathsf{m}_0, \mathsf{m}_1, S \subseteq [N]) \leftarrow \mathcal{A}^{\mathcal{C}(\cdot),\mathcal{D}(\cdot)}(\mathsf{pp}, (\mathsf{ek}_i)_{i\in[N]})$
6 : $b \leftarrow\$ \{0,1\}$
7 : $(\mathsf{T}^*, \vec{\mathsf{ct}}^* := (\mathsf{ct}^*_i)_{i\in S}) \leftarrow \mathsf{CmEnc}(\mathsf{pp}, (\mathsf{ek}_i)_{i\in S}, \mathsf{m}_b)$
8 : $b' \leftarrow \mathcal{A}^{\mathtt{Corr}(\cdot),\mathtt{CmDec}(\cdot)}(\mathsf{pp}, (\mathsf{ek}_i)_{i\in[N]}, \vec{\mathsf{ct}}^*)$
9 : **if** $C \cap S \neq \emptyset$ **then**
10 : $\quad$**return** b
11 : **return** $[\![b = b']\!]$

Decapsulation Oracle $\mathtt{CmDec}(i, \mathsf{T}, \mathsf{ct})$

1 : **req** $(\mathsf{T}, \mathsf{ct}) \neq (\mathsf{T}^*, \mathsf{ct}^*_i)$
2 : $\mathsf{m} \leftarrow \mathsf{CmDec}(\mathsf{dk}_i, \mathsf{T}, \mathsf{ct})$
3 : **return** m

Corruption Oracle $\mathtt{Corr}(i)$

1 : $C \leftarrow C \cup \{i\}$
2 : **return** dk_i

FIGURE 4.2: Security games for defining the IND-CCA security with adaptive corruption of CmPKE. If the condition following **req** does not hold, the game terminates by returning a random bit.

where we ask a ciphertext to be bound to a unique key and message pair. Although it may sound like an obscure property at first glance, this property has been shown to be vital for establishing security in several practical applications such as Facebook Messenger [Dod+18], (see [Alb+22] for more examples). In a CmPKE, we extend this to the multi-user setting, which requires that if any of the recipients decrypt to a message m, then the other recipients should also decrypt either to m or to $\perp$. Informally, and unlike in the single-user setting, we allow a ciphertext to be decryptable by many recipients (i.e., many different keys) but enforce that their decryption values remain consistent if not $\perp$. Looking ahead, this is a natural property to desire when guaranteeing the weak robustness of a CGKA protocol (i.e., if a user receives a message then it should be consistent with all the other group members, provided that they can process the message).

4.2.1 Definition

In this section, we define the formal syntax and required properties of committing multi-recipient public-key encryption.

Definition 4.2.1 (Committing Multi-Recipient Public-Key Encryption). *A (single-message) committing multi-recipient public-key encryption (*CmPKE*) over a message space $\mathcal{M}$ consists of the following algorithms:*

- $\mathsf{CmSetup}(1^\kappa) \to \mathsf{pp}$: *The setup algorithm takes the security parameter 1^κ as input and outputs a public parameter* pp*. In the following, we assume* pp *is provided to all the algorithms and may omit it for simplicity.*
- $\mathsf{CmGen}(\mathsf{pp}) \to (\mathsf{ek}, \mathsf{dk})$: *The key generation algorithm takes a public parameter* pp *as input and outputs a pair of encryption key and decryption key* $(\mathsf{ek}, \mathsf{dk})$*.*
- $\mathsf{CmEnc}((\mathsf{ek}_i)_{i\in[N]}, \mathsf{m}) \to (\mathsf{T}, \vec{\mathsf{ct}} = (\mathsf{ct}_i)_{i\in[N]})$: *The encryption algorithm takes a public parameter* pp*, N encryption keys $(\mathsf{ek}_i)_{i\in[N]}$, and a message $\mathsf{m} \in \mathcal{M}$ as input and outputs a commitment* T *and N ciphertexts $\vec{\mathsf{ct}} = (\mathsf{ct}_i)_{i\in[N]}$.*

$\mathsf{Game}^{\mathsf{Com\text{-}Bind}}_{\mathsf{CmPKE},\mathcal{A}}(\kappa)$

$\mathsf{pp} \leftarrow \mathsf{CmSetup}(1^\kappa)$
$(\mathsf{T}^*, (\mathsf{dk}_b, \mathsf{ct}_b)_{b\in\{0,1\}}) \leftarrow \mathcal{A}(\mathsf{pp})$
foreach $b \in \{0,1\}$ **do**
  $\mathsf{m}_b \leftarrow \mathsf{CmDec}(\mathsf{dk}_b, \mathsf{T}^*, \mathsf{ct}_b)$
if $\mathsf{dk}_0 = \mathsf{dk}_1$ **then**
  return $[\![\mathsf{ct}_0 \neq \mathsf{ct}_1]\!] \wedge [\![\mathsf{m}_0 \neq \bot]\!] \wedge [\![\mathsf{m}_1 \neq \bot]\!]$
else
  return $[\![\mathsf{m}_0 \neq \mathsf{m}_1]\!] \wedge [\![\mathsf{m}_0 \neq \bot]\!] \wedge [\![\mathsf{m}_1 \neq \bot]\!]$

FIGURE 4.3: Security games for defining the commitment binding property of CmPKE.

- $\mathsf{CmDec}(\mathsf{dk}, \mathsf{T}, \mathsf{ct}_i) \to \mathsf{m}/\bot$: *The decapsulation algorithm takes a decryption key* dk*, a commitment* T*, and a ciphertext* ct_i *as input and outputs either* $\mathsf{m} \in \mathcal{M}$ *or* $\bot \notin \mathcal{M}$*.*

Definition 4.2.2 (Correctness). *We say that a* CmPKE *is* correct *if, for all* $N = \mathsf{poly}(\kappa)$ *and* $\mathsf{m} \in \mathcal{M}$*,*

$$1 - \mathsf{negl}(\kappa) \leq \Pr[\forall i \in [N], \mathsf{m} = \mathsf{CmDec}(\mathsf{dk}_i, \mathsf{T}, \mathsf{ct}_i)],$$

where the probability is taken over $\mathsf{pp} \leftarrow \mathsf{CmSetup}(1^\kappa)$*,* $((\mathsf{ek}_i, \mathsf{dk}_i) \leftarrow \mathsf{CmGen}(\mathsf{pp}))_{i\in[N]}$*, and* $(\mathsf{T}, \vec{\mathsf{ct}} = (\mathsf{ct}_i)_{i\in[N]}) \leftarrow \mathsf{CmEnc}(\mathsf{pp}, (\mathsf{ek}_i)_{i\in[N]}, \mathsf{m})$.[6]

Definition 4.2.3 (Succinctness). *We say a* CmPKE *is* succinct *if in the above Definition 4.2.2, the size of commitment* T *(and all ciphertext* ct_i*) is independent of the number of recipients* N*.*

Definition 4.2.4 (Ciphertext-Spreadness). *We say* CmPKE *is* ciphertext-spread *if*

$$\mathbb{E}\left[\max_{\mathsf{T}^*,\mathsf{ct}^*,\mathsf{m}\in\mathcal{M},i} \Pr_{\mathsf{r}}\left[(\mathsf{T}, (\mathsf{ct}_i)_{i\in[N]}) \leftarrow \mathsf{CmEnc}(\mathsf{pp}, (\mathsf{ek}_i)_{i\in[N]}, \mathsf{m}; \mathsf{r}) : (\mathsf{T}^*, \mathsf{ct}^*) = (\mathsf{T}, \mathsf{ct}_i)\right]\right] \leq \mathsf{negl}(\kappa)$$

for all $\mathsf{pp} \in \mathsf{CmSetup}(1^\kappa)$ *and* $(\mathsf{ek}_i, \mathsf{dk}_i) \in \mathsf{CmGen}(\mathsf{pp})$ *for all* $i \in [N]$*, where the expectation is taken over* $\mathsf{pp} \leftarrow \mathsf{CmSetup}(1^\kappa)$ *and* $(\mathsf{ek}_i, \mathsf{dk}_i) \leftarrow \mathsf{CmGen}(\mathsf{pp})$ *for all* $i \in [N]$*.*

In this work, we only consider a succinct CmPKE. Thus, we omit it for simplicity. We define indistinguishability against chosen ciphertext attacks (IND-CCA) *with adaptive corruption* for CmPKE.

Definition 4.2.5 (IND-CCA with Adaptive Corruption). *The security notion is defined by the game illustrated in Figure 4.2. We say a* CmPKE *is* IND-CCA secure with adaptive corruption *if for any efficient adversaries* $\mathcal{A}$*, we have*

$$\left|\Pr\left[\mathsf{Game}^{\mathsf{IND\text{-}CCA}}_{\mathsf{CmPKE},\mathcal{A}}(\kappa) = 1\right] - \frac{1}{2}\right| \leq \mathsf{negl}(\kappa).$$

If $\mathcal{A}$ *is not given access to the corruption oracle* `Corr`*, this game corresponds to standard* IND-CCA *security.*

[6] In the proof of our CGKA protocol, we require that the adversary cannot find a "bad" randomness that leads to a decryption error. Since we use a PRG modeled as a random oracle to expand the randomness, standard correctness immediately implies that no PPT adversary can find such bad randomness.

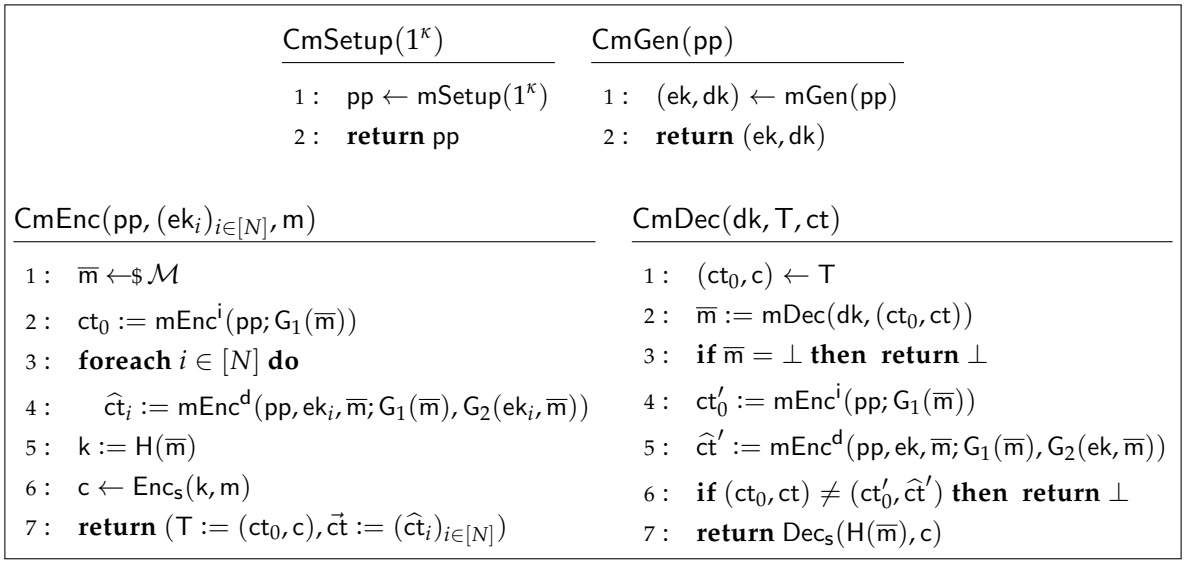


FIGURE 4.4: An IND-CCA secure CmPKE from an IND-CPA secure decomposable mPKE and a one-time IND-CCA secure SKE.

Finally, we define commitment-binding which roughly says that the commitment T is implicitly bound to a unique message. The notion we consider is strong in the sense that the adversary can use an arbitrary decryption key rather than a correctly generated one to break commitment-binding property.

Definition 4.2.6 (Commitment-Binding). *The security notion is defined by the game illustrated in Figure 4.3. We say a* CmPKE *is* commitment-binding *if for any efficient adversaries $\mathcal{A}$, we have*

$$\Pr\left[\mathsf{Game}^{\mathsf{Com\text{-}Bind}}_{\mathsf{CmPKE},\mathcal{A}}(\kappa) = 1\right] \leq \mathsf{negl}(\kappa).$$

Note that independently satisfying succinctness and commitment-binding is trivial. If we run a standard PKE in parallel for all N users and set $\mathsf{T} := \perp$, then we obtain a succinct scheme but this is clearly not commitment-binding. On the other hand, if we add a non-interactive zero-knowledge (NIZK) proof π to further prove that all the PKE ciphertexts encrypt the same message and set $\mathsf{T} := (\pi, \mathsf{ct}_1, \cdots, \mathsf{ct}_N)$ (as in the strongly robust TreeKEM variant of [Alw+20b]), then we obtain a commitment-binding scheme but the commitment is no longer succinct. Therefore, the main non-triviality is making the commitment size $|\mathsf{T}|$ independent of the number of users, while simultaneously allowing the users to be convinced that if $(\mathsf{T}, \mathsf{ct}_i)$ decrypts to a valid message, then any other users' $(\mathsf{T}, \mathsf{ct}_j)$ will also decrypt to the same message (or to $\perp$).

4.2.2 Construction of CmPKE: IND-CCA without Adaptive Corruption

We provide a simple and efficient generic construction of an IND-CCA secure CmPKE (without adaptive corruption) from a decomposable IND-CPA secure mPKE and a one-time IND-CCA secure SKE following the Fujisaki–Okamoto transformation generalized to the multi-recipient setting. This is illustrated in Figure 4.4, where $\mathsf{G}_1, \mathsf{G}_2, \mathsf{H}$ are hash functions modeled as random oracles in the security proof. These oracles can be simulated by a single random oracle by using appropriate domain separation. Here, we assume the output space of H is identical to the secret key space $\mathcal{K}$ of the SKE. The correctness of this CmPKE follows

Game 3 : Decryption Oracle $\mathtt{CmDec}(i, \mathsf{T}, \mathsf{ct})$

1 : **req** $(\mathsf{T}, \mathsf{ct}) \neq (\mathsf{T}^*, \widehat{\mathsf{ct}}_i^*)$
2 : **parse** $(\mathsf{ct}_0, \mathsf{c}) \leftarrow \mathsf{T}$
3 : **if** $(\mathsf{ct}_0, \mathsf{ct}) = (\mathsf{ct}_0^*, \widehat{\mathsf{ct}}_i^*)$ **then**
4 : **return** $\mathsf{Dec}_{\mathsf{s}}(\mathsf{H}(\overline{\mathsf{m}}^*), \mathsf{c})$
5 : $\overline{\mathsf{m}} := \mathsf{mDec}(\mathsf{dk}_i, (\mathsf{ct}_0, \mathsf{ct}))$
6 : **if** $\boxed{\overline{\mathsf{m}} \notin L_{\mathsf{G}}} \vee \overline{\mathsf{m}} = \perp$ **then**
7 : **return** $\perp$
8 : $\mathsf{ct}_0' := \mathsf{mEnc}^{\mathsf{i}}(\mathsf{pp}; \mathsf{G}_1(\overline{\mathsf{m}}))$
9 : $\widehat{\mathsf{ct}}_i' := \mathsf{mEnc}^{\mathsf{d}}(\mathsf{pp}, \mathsf{ek}_i, \overline{\mathsf{m}}; \mathsf{G}_1(\overline{\mathsf{m}}), \mathsf{G}_2(\mathsf{ek}_i, \overline{\mathsf{m}}))$
10 : **if** $(\mathsf{ct}_0, \mathsf{ct}) \neq (\mathsf{ct}_0', \widehat{\mathsf{ct}}')$ **then**
11 : **return** $\perp$
12 : **return** $\mathsf{Dec}_{\mathsf{s}}(\mathsf{H}(\overline{\mathsf{m}}), \mathsf{c})$

Game 4 : Decryption Oracle $\mathtt{CmDec}(i, \mathsf{T}, \mathsf{ct})$

1 : **req** $(\mathsf{T}, \mathsf{ct}) \neq (\mathsf{T}^*, \widehat{\mathsf{ct}}_i^*)$
2 : **parse** $(\mathsf{ct}_0, \mathsf{c}) \leftarrow \mathsf{T}$
3 : **if** $(\mathsf{ct}_0, \mathsf{ct}) = (\mathsf{ct}_0^*, \widehat{\mathsf{ct}}_i^*)$ **then**
4 : **return** $\mathsf{Dec}_{\mathsf{s}}(\mathsf{H}(\overline{\mathsf{m}}^*), \mathsf{c})$
5 : **foreach** $\overline{\mathsf{m}} \in L_{\mathsf{G}}$ **do**
6 : $\mathsf{ct}_0' := \mathsf{mEnc}^{\mathsf{i}}(\mathsf{pp}; \mathsf{G}_1(\overline{\mathsf{m}}))$
7 : $\widehat{\mathsf{ct}}_i' := \mathsf{mEnc}^{\mathsf{d}}(\mathsf{pp}, \mathsf{ek}_i, \mathsf{m}; \mathsf{G}_1(\overline{\mathsf{m}}), \mathsf{G}_2(\mathsf{ek}_i, \overline{\mathsf{m}}))$
8 : **if** $(\mathsf{ct}_0, \mathsf{ct}) = (\mathsf{ct}_0', \widehat{\mathsf{ct}}_i')$ **then**
9 : **return** $\mathsf{Dec}_{\mathsf{s}}(\mathsf{H}(\overline{\mathsf{m}}), \mathsf{c})$
10 : **return** $\perp$

FIGURE 4.5: Procedure of decryption oracles in Game 3 and Game 4. The box indicates the modification from Game 2 to Game 3.

immediately from the correctness of the decomposable mPKE and SKE. The following theorems assert the IND-CCA security and commitment-binding of the CmPKE. The proof for Theorem 4.2.7 is a standard adaptation of the KEM/DEM framework to the multi-user setting. The proof for Theorem 4.2.10 follows naturally from the key committing property of the underlying SKE.

Theorem 4.2.7. *The* CmPKE *in Figure 4.4 is* IND-CCA *secure (resp. with adaptive corruption) assuming the* SKE *is one-time* IND-CCA *secure and the decomposable* mPKE *is* IND-CPA *secure (resp. with adaptive corruption) and ciphertext-spread.*

Proof of Theorem 4.2.7. Let $\mathcal{A}$ be an adversary against the IND-CCA security of CmPKE with advantage ϵ. Without loss of generality, we make a simplifying argument that $\mathcal{A}$'s random oracle queries to G_1 and G_2 are answered as $(\mathsf{G}_1(\mathsf{m}), \mathsf{G}_2(\mathsf{ek}_1, \mathsf{m}), \cdots, \mathsf{G}_2(\mathsf{ek}_N, \mathsf{m}))$, where $(\mathsf{ek}_i)_{i\in[N]}$ are the encryption keys generated by the security game. It is clear that this modification does not weaken $\mathcal{A}$. Moreover, we can always transform an adversary $\mathcal{A}$ that does not conform to this style to one that does. Below, we evaluate the upper bound of $\mathcal{A}$'s advantage ϵ by considering a sequence of games. We denote by S_i the event $b = b'$ in Game i.

Game 0: This is the real IND-CCA security game. By definition $|\Pr[\mathsf{S}_0] - 1/2| = \epsilon$. We assume without loss of generality that the random message $\overline{\mathsf{m}}^* \leftarrow \mathcal{M}$ used to generate the challenge ciphertext is sampled at the beginning of the game.

Game 1: In this game, we modify the random oracle G so that the output is distributed randomly over the space of randomness for which the decomposable mPKE does not fail decryption. That is, we require $\overline{\mathsf{m}} = \mathsf{mDec}(\mathsf{dk}_i, (\mathsf{ct}_0, \widehat{\mathsf{ct}}_i))$ for all $i \in [N]$ and $\overline{\mathsf{m}} \in \mathcal{M}$, where $\mathsf{ct}_0 := \mathsf{mEnc}^{\mathsf{i}}(\mathsf{pp}; \mathsf{G}_1(\overline{\mathsf{m}}))$ and $\widehat{\mathsf{ct}}_i := \mathsf{mEnc}^{\mathsf{d}}(\mathsf{pp}, \mathsf{ek}_i, \overline{\mathsf{m}}; \mathsf{G}_1(\overline{\mathsf{m}}), \mathsf{G}_2(\mathsf{ek}_i, \overline{\mathsf{m}}))$. Due to correctness of the decomposable mPKE, for any $\mathcal{A}$ making at most polynomial random oracle queries, we have $|\Pr[\mathsf{S}_0] - \Pr[\mathsf{S}_1]| \leq \mathsf{negl}(\kappa)$.

(The next Game 2, Game 3 *and* Game 4 *aim to get rid of the secret keys* ek_i *to answer* $\mathcal{A}$*'s decryption oracle queries.)*

Game 2: In this game, the challenger modifies how it answers the decryption oracle query. When $\mathcal{A}$ queries $(i, \mathsf{T} = (\mathsf{ct}_0, \mathsf{c}), \mathsf{ct})$ such that $(\mathsf{ct}_0, \mathsf{ct}) = (\mathsf{ct}_0^*, \widehat{\mathsf{ct}}_i^*)$, the challenger simply returns $\mathsf{Dec}_{\mathsf{s}}(\mathsf{H}(\overline{\mathsf{m}}^*), \mathsf{c})$. This is in contrast to the previous game where the challenger decrypted $(\mathsf{ct}_0, \mathsf{ct})$ using mDec. Nonetheless, since the decomposable mPKE is perfectly correct due to the modification we made in Game 1, this modification does not alter the view of the adversary. In particular, we have $\Pr[\mathsf{S}_1] = \Pr[\mathsf{S}_2]$.

Game 3: In this game, the challenger checks an additional condition when answering the decryption oracle query. This is illustrated in Figure 4.5, where the box indicates the modification. Here, L_{G} is a list that stores the random oracle queries made to G_1 and G_2 by the adversary. We have $\mathsf{m} \in L_{\mathsf{G}}$ if G_1 was queried on m and G_2 was queried on $(\mathsf{ek}, \mathsf{m})$ for any ek. Note that due to our assumption on $\mathcal{A}$, if one of the oracles G_1 or G_2 was queried on m, then so would have the other.

The only difference occurs when $\mathcal{A}$ queries $(i, \mathsf{T} = (\mathsf{ct}_0, \mathsf{c}), \mathsf{ct})$ such that $\overline{\mathsf{m}} := \mathsf{mDec}(\mathsf{dk}_i, (\mathsf{ct}_0, \mathsf{ct}))$ has not been queried to the random oracles G_1 and G_2 but $\mathsf{ct}_0 = \mathsf{mEnc}^{\mathsf{i}}(\mathsf{pp}; \mathsf{G}_1(\overline{\mathsf{m}}))$ and $\mathsf{ct} = \mathsf{mEnc}^{\mathsf{d}}(\mathsf{pp}, \mathsf{ek}_i, \overline{\mathsf{m}}; \mathsf{G}_1(\overline{\mathsf{m}}), \mathsf{G}_2(\mathsf{ek}_i, \overline{\mathsf{m}}))$. Notice $\mathsf{G}_1(\overline{\mathsf{m}})$ and $\mathsf{G}_2(\mathsf{ek}_i, \overline{\mathsf{m}})$ are information theoretically hidden from $\mathcal{A}$ unless $\mathcal{A}$ queries them. Therefore, due to ciphertext-spreadness of the decomposable mPKE, we must have had $(\mathsf{ct}_0, \mathsf{ct}) \neq (\mathsf{ct}_0', \widehat{\mathsf{ct}}_i')$ in the previous game as well. Hence, we have $|\Pr[\mathsf{S}_2] - \Pr[\mathsf{S}_3]| \leq \mathsf{negl}(\kappa)$.

Game 4: In this game, the challenger further modifies how it answers the decryption-oracle query. This is illustrated in Figure 4.5, where notice that the challenger no longer requires the secret keys dk_i to answer the queries.

We check the output of the decryption oracles in Game 3 and Game 4 are identical. Since the two oracles run identically in case $(\mathsf{ct}_0, \mathsf{ct}) = (\mathsf{ct}_0^*, \widehat{\mathsf{ct}}_i^*)$, we only focus on the case that this does not hold. Assume the decryption oracle in Game 3 outputs a non-$\perp$ message m (i.e., $\mathsf{m} = \mathsf{Dec}_{\mathsf{s}}(\mathsf{H}(\overline{\mathsf{m}}), \mathsf{c})$). Then $\overline{\mathsf{m}} \in L_{\mathsf{G}}$ and $(\mathsf{ct}_0, \mathsf{ct}) = (\mathsf{ct}_0', \widehat{\mathsf{ct}}_i')$ hold, where $\overline{\mathsf{m}} := \mathsf{mDec}(\mathsf{dk}_i, (\mathsf{ct}_0, \mathsf{ct}))$. Therefore, the decryption oracle in Game 3 outputs the same non-$\perp$ message m. On the other hand, assume the decryption oracle in Game 4 outputs a non-$\perp$ message m. Then, there exists a $\overline{\mathsf{m}} \in L_{\mathsf{G}}$ such that $\mathsf{ct}_0' := \mathsf{mEnc}^{\mathsf{i}}(\mathsf{pp}; \mathsf{G}_1(\overline{\mathsf{m}}))$ and $\widehat{\mathsf{ct}}_i' := \mathsf{mEnc}^{\mathsf{d}}(\mathsf{pp}, \mathsf{ek}_i, \overline{\mathsf{m}}; \mathsf{G}_1(\overline{\mathsf{m}}), \mathsf{G}_2(\mathsf{ek}_i, \overline{\mathsf{m}}))$ such that $(\mathsf{ct}_0, \mathsf{ct}) = (\mathsf{ct}_0', \widehat{\mathsf{ct}}_i')$. Conditioning on no correctness error occurring, $(\mathsf{ct}_0, \mathsf{ct})$ decrypts to $\overline{\mathsf{m}}$. Therefore, this implies that the decryption oracle in Game 4 outputs the same non-$\perp$ message m. Combining the arguments together, we have $\Pr[\mathsf{S}_3] = \Pr[\mathsf{S}_4]$.

Game 5: In this game, we undo the change we made in Game 1 and alter the output of the random oracles G_1 and G_2 to be over all the randomness space. Due to the same argument, we made before, we have $|\Pr[\mathsf{S}_4] - \Pr[\mathsf{S}_5]| \leq \mathsf{negl}(\kappa)$

(We are now ready to invoke IND-CPA *security of the decomposable* mPKE *and* IND-CCA *security of the* SKE*.)*

Game 6: Let us define QUERY as the event that $\mathcal{A}$ queries the random oracles $\mathsf{H}(\cdot)$, $\mathsf{G}_1(\cdot)$, or $\mathsf{G}_2(\star, \cdot)$ on input $\overline{\mathsf{m}}^*$, where $\star$ denotes an arbitrary element. (Recall the change we made in Game 1 for $\overline{\mathsf{m}}^*$.) In this game, the challenger aborts the game and forces $\mathcal{A}$ to output a random bit when QUERY occurs. We show in Lemma 4.2.8 that we have $|\Pr[\mathsf{S}_5] - \Pr[\mathsf{S}_6]| \leq \mathsf{negl}(\kappa)$ assuming the decomposable mPKE is IND-CPA secure (with adaptive corruption) and the message space $\mathcal{M}$ is sufficiently large. So as not to interrupt the main proof, we postpone the proof of Lemma 4.2.8 to the end.

We finally show in Lemma 4.2.9 that assuming the one-time IND-CCA security of the SKE, we have

$$\Pr[\mathsf{S}_6] = \frac{1}{2} + \mathsf{negl}(\kappa).$$

Combining all the bounds together, we obtain the statement in Theorem 4.2.7.

It remains to prove Lemmata 4.2.8 and 4.2.9 below.

Lemma 4.2.8. *We have* $|\Pr[\mathsf{S}_5] - \Pr[\mathsf{S}_6]| \leq \mathsf{negl}(\kappa)$ *assuming the decomposable* mPKE *is* IND-CPA *secure (with adaptive corruption) and the message space* $\mathcal{M}$ *is super-polynomially large.*

Proof. Since the two games are identical unless QUERY occurs, we have $|\Pr[\mathsf{S}_5] - \Pr[\mathsf{S}_6]| \leq \Pr[\mathsf{QUERY}]$. In the following, we upper bound $\Pr[\mathsf{QUERY}]$. Let us construct an IND-CPA adversary $\mathcal{B}$ which runs $\mathcal{A}$ as a subroutine: On input $(\mathsf{pp}, (\mathsf{ek}_i)_{i\in[N]})$, $\mathcal{B}$ samples two random messages $\overline{\mathsf{m}}_0^*, \overline{\mathsf{m}}_1^* \leftarrow \mathcal{M}$ and a random SKE key $\mathsf{k}^* \leftarrow \mathcal{K}$. It then invokes $\mathcal{A}$ on input $(\mathsf{pp}, (\mathsf{ek}_i)_{i\in[N]})$. $\mathcal{B}$ can simulate the decryption queries as it no longer requires knowledge of the secret key. When $\mathcal{A}$ corrupts a user, $\mathcal{B}$ simply relays the corruption to its own challenger. Finally, when $\mathcal{A}$ submits $(\mathsf{m}_0, \mathsf{m}_1, S \subseteq [N])$ as its challenge, $\mathcal{B}$ submits $(\overline{\mathsf{m}}_0^*, \overline{\mathsf{m}}_1^*, S)$ to its challenger and receives $(\mathsf{ct}_0^*, (\widehat{\mathsf{ct}}_i^*)_{i\in[S]}) \leftarrow \mathsf{mEnc}(\mathsf{pp}, (\mathsf{ek}_i)_{i\in S}, \overline{\mathsf{m}}_b^*)$ for an unknown randomly chosen bit b. $\mathcal{B}$ then samples a random challenge bit $b' \leftarrow \{0,1\}$ and generates $\mathsf{c}^* \leftarrow \mathsf{Enc}_{\mathsf{s}}(\mathsf{k}^*, \mathsf{m}_{b'})$. It finally provides the challenge ciphertext $(\mathsf{T}^* := (\mathsf{ct}_0^*, \mathsf{c}^*), \vec{\mathsf{ct}}^* := (\widehat{\mathsf{ct}}_i^*)_{i\in S})$ to $\mathcal{A}$. $\mathcal{B}$ outputs $\hat{b} := 0$, if $\overline{\mathsf{m}}_0^*$ is queried to $\mathsf{H}(\cdot)$, $\mathsf{G}_1(\cdot)$, or $\mathsf{G}_2(\star, \cdot)$ before $\overline{\mathsf{m}}_1^*$ is; outputs $\hat{b} := 1$, if $\overline{\mathsf{m}}_1^*$ is queried to $\mathsf{H}(\cdot)$, $\mathsf{G}_1(\cdot)$, or $\mathsf{G}_2(\star, \cdot)$ before $\overline{\mathsf{m}}_0^*$ is; and a random $\hat{b}$ when neither $\overline{\mathsf{m}}_0^*$ nor $\overline{\mathsf{m}}_1^*$ are queried. Let us denote GOOD (resp. BAD) the event that $\mathcal{A}$ queries $\overline{\mathsf{m}}_b^*$ (resp. $\overline{\mathsf{m}}_{1-b}^*$) before $\overline{\mathsf{m}}_{1-b}^*$ (resp. $\overline{\mathsf{m}}_b^*$) to $\mathsf{H}(\cdot)$, $\mathsf{G}_1(\cdot)$, or $\mathsf{G}_2(\star, \cdot)$. Moreover, let us denote RAND the event that neither $\overline{\mathsf{m}}_0^*$ nor $\overline{\mathsf{m}}_1^*$ are queried. Observe that until either GOOD or BAD occurs, $\mathcal{B}$ simulates the view of Game 6 and Game 7 perfectly to $\mathcal{A}$. Since QUERY is the event that $\overline{\mathsf{m}}_b^*$ is ever queried throughout the game, we have $\Pr[\mathsf{QUERY}] \leq \Pr[\mathsf{GOOD}] + \Pr[\mathsf{BAD}]$. Moreover, since m_{1-b}^* is completely hidden from $\mathcal{A}$, we have $\Pr[\mathsf{BAD}] \leq \mathsf{negl}(\kappa)$ assuming the message size is super-polynomially large and $\mathcal{A}$ only makes polynomially many random oracle queries.

Using these observations, we can rewrite the advantage of $\mathcal{B}$ as follows:

$$\begin{aligned}
\left|\Pr[b = b'] - \frac{1}{2}\right| &= \left|\Pr[b = b' \wedge \mathsf{GOOD}] + \Pr[b = b' \wedge \mathsf{BAD}] + \Pr[b = b' \wedge \mathsf{RAND}] - \frac{1}{2}\right| \\
&= \left|\Pr[\mathsf{GOOD}] + \frac{1}{2} \cdot \Pr[\mathsf{RAND}] - \frac{1}{2}\right| \\
&= \left|\Pr[\mathsf{GOOD}] + \frac{1}{2} \cdot (1 - \Pr[\mathsf{GOOD}] - \Pr[\mathsf{BAD}]) - \frac{1}{2}\right| \\
&= \left|\frac{1}{2}(\Pr[\mathsf{GOOD}] - \Pr[\mathsf{BAD}])\right| \\
&\geq \frac{1}{2}\left(\Pr[\mathsf{QUERY}] - 2\Pr[\mathsf{BAD}]\right).
\end{aligned}$$

Assuming the hardness of the IND-CPA security of the decomposable mPKE, we have $|\Pr[b = b'] - 1/2| \leq \mathsf{negl}(\kappa)$. Thus, rewriting the inequality and plugging in $\Pr[\mathsf{BAD}] \leq \mathsf{negl}(\kappa)$, we obtain $\Pr[\mathsf{QUERY}] \leq \mathsf{negl}(\kappa)$ as desired. This concludes the proof. □

Lemma 4.2.9. *We have* $\Pr[\mathsf{S}_6] = \frac{1}{2} + \mathsf{negl}(\kappa)$ *assuming the* SKE *is the one-time* IND-CCA *secure.*

Proof. Assume $\mathcal{A}$ has an advantage ϵ in Game 6. We construct an adversary $\mathcal{B}$ that breaks the one-time IND-CCA security of the SKE with the same advantage by internally running $\mathcal{A}$ as follows:

$\mathcal{B}$ generates $(\mathsf{pp}, (\mathsf{ek}_i, \mathsf{dk}_i)_{i\in[N]})$ and samples a random $\overline{\mathsf{m}}^* \leftarrow \mathcal{M}$ used to generate the challenge ciphertext. $\mathcal{B}$ then invokes $\mathcal{A}$ on input $(\mathsf{pp}, (\mathsf{ek}_i)_{i\in[N]})$ as in Game 6. When $\mathcal{A}$ queries any of the random oracles on input $\overline{\mathsf{m}}^*$ (i.e., when event QUERY occurs), then abort as specified by the modification we made in Game 5. When $\mathcal{A}$ queries for a challenge ciphertext on input $(\mathsf{m}_0, \mathsf{m}_1, S \subseteq [N])$, $\mathcal{B}$ first generates $(\mathsf{ct}_0^*, (\widehat{\mathsf{ct}}_i^*))_{i\in[N]}$. It then queries its SKE-challenger for a challenge ciphertext on challenge messages $(\mathsf{m}_0, \mathsf{m}_1)$ and receives back c^*. Finally, $\mathcal{B}$ returns $(\mathsf{T}^* := (\mathsf{ct}_0^*, \mathsf{c}^*), \vec{\mathsf{ct}}^* := (\widehat{\mathsf{ct}}_i^*)_{i\in[N]})$ to $\mathcal{A}$. Here, notice that $\mathcal{B}$ implicitly sets $\mathsf{H}(\overline{\mathsf{m}}^*) = \mathsf{k}^*$, where k^* is the secret key used by the SKE-challenger. When $\mathcal{A}$ queries the decryption oracle on input $(i, \mathsf{T}, \mathsf{ct})$, if $(\mathsf{ct}_0, \mathsf{ct}) \neq (\mathsf{ct}_0^*, \widehat{\mathsf{ct}}_i^*)$, then $\mathcal{B}$ proceeds exactly as in Game 7. Otherwise, it queries its own SKE-decryption oracle on input c (which is guaranteed to be different from c^*) and outputs $\mathcal{A}$ the decryption result. Corruption queries made by $\mathcal{A}$ can be handled as in the real game since $\mathcal{B}$ knows all the user's secrets. Finally, when $\mathcal{A}$ outputs b' at the end of the game, $\mathcal{B}$ outputs b' as its guess.

Conditioning on event QUERY not occurring, $\mathcal{B}$ perfectly simulates Game 6 to $\mathcal{A}$. Therefore, $\mathcal{B}$ has the same advantage of winning the one-time IND-CCA security game of SKE as $\mathcal{A}$ does in winning Game 6. Hence, assuming the one-time IND-CCA security of SKE, we conclude $\Pr[\mathsf{S}_6] = \frac{1}{2} + \mathsf{negl}(\kappa)$. □

The proof of Theorem 4.2.7 is completed. □

Theorem 4.2.10. *The* CmPKE *in Figure 4.4 has commitment-binding property assuming the* SKE *has key-committing property.*

Proof of Theorem 4.2.10. Assume by contradiction that $\mathcal{A}$ breaks commitment-binding of CmPKE. By assumption $\mathcal{A}$ outputs $(\mathsf{T}^*, (\mathsf{dk}_b, \mathsf{ct}_b)_{i\in b})$. Let $\mathsf{T}^* := (\mathsf{ct}_0^*, \mathsf{c}^*)$ and $\mathsf{m}_b \leftarrow \mathsf{CmDec}(\mathsf{dk}_b, \mathsf{T}^*, \mathsf{ct}_b)$ for $b \in \{0,1\}$. Moreover, let $\overline{\mathsf{m}}_b \leftarrow \mathsf{mDec}(\mathsf{dk}_b, \mathsf{ct}_0^*, \mathsf{ct}_b)$ for $b \in \{0,1\}$ be the internal random message decrypted while running CmDec.

We first show that we must have $\overline{\mathsf{m}}_0 \neq \overline{\mathsf{m}}_1$. If this does not hold, then in case $\mathsf{dk}_0 = \mathsf{dk}_1$, we must have $(\mathsf{ct}_0^*, \mathsf{ct}_0) = (\mathsf{ct}_0^*, \mathsf{ct}_1)$ due to the re-encryption check during decryption.[7] However, this does not constitute a valid attack. On the other hand, in case $\mathsf{dk}_0 \neq \mathsf{dk}_1$, we have $\mathsf{Dec}_\mathsf{s}(\mathsf{H}(\overline{\mathsf{m}}_0), \mathsf{c}^*) = \mathsf{Dec}_\mathsf{s}(\mathsf{H}(\overline{\mathsf{m}}_1), \mathsf{c}^*) = \mathsf{m}$. Therefore, this too does not constitute a valid attack either.

Next, conditioning on $\overline{\mathsf{m}}_0 \neq \overline{\mathsf{m}}_1$, we can further assume $\mathsf{H}(\overline{\mathsf{m}}_0) \neq \mathsf{H}(\overline{\mathsf{m}}_1)$ with making negligible difference in the advantage of the adversary since H is modeled as a random oracle. This implies that the adversary implicitly outputs two keys $\mathsf{k}_0 := \mathsf{H}(\overline{\mathsf{m}}_0)$ and $\mathsf{k}_1 := \mathsf{H}(\overline{\mathsf{m}}_1)$ such that $\mathsf{Dec}_\mathsf{s}(\mathsf{k}_0, \mathsf{c}^*) = \mathsf{m}_0$ and $\mathsf{Dec}_\mathsf{s}(\mathsf{k}_1, \mathsf{c}^*) = \mathsf{m}_1$. However, this contradicts the key commitment property of SKE (regardless of m_0 being the same or different from m_1). This concludes the proof.[8] □

Theorem 4.2.11. *The* CmPKE *in Figure 4.4 is ciphertext-spread assuming the decomposable* mPKE *is ciphertext-spread.*

Proof. The ciphertext of the CmPKE in Figure 4.4 consists of the decomposable mPKE ciphertext $(\mathsf{ct}_0, (\widehat{\mathsf{ct}}_i)_{i\in[N]})$ and SKE ciphertext c. Since mPKE is ciphertext-spread, we have

$$\mathbb{E}\left[\max_{\mathsf{ct}_0^*, \mathsf{c}^*, \mathsf{ct}^*, \mathsf{m}\in\mathcal{M}, i} \Pr_{\mathsf{r}}\left[((\mathsf{ct}_0, \mathsf{c}), (\mathsf{ct}_i)_{i\in[N]}) \leftarrow \mathsf{CmEnc}(\mathsf{pp}, (\mathsf{ek}_i)_{i\in[N]}, \mathsf{m}; \mathsf{r}) : (\mathsf{ct}_0^*, \mathsf{c}^*, \mathsf{ct}^*) = (\mathsf{ct}_0, \mathsf{c}, \mathsf{ct}_i)\right]\right]$$
$$\leq \mathbb{E}\left[\max_{\mathsf{ct}_0^*, \mathsf{c}^*, \mathsf{ct}^*, \mathsf{m}\in\mathcal{M}, i} \Pr_{\mathsf{r}}\left[((\mathsf{ct}_0, \mathsf{c}), (\mathsf{ct}_i)_{i\in[N]}) \leftarrow \mathsf{CmEnc}(\mathsf{pp}, (\mathsf{ek}_i)_{i\in[N]}, \mathsf{m}; \mathsf{r}) : (\mathsf{ct}_0^*, \mathsf{ct}^*) = (\mathsf{ct}_0, \mathsf{ct}_i)\right]\right]$$

[7] Here, we assume that a decryption key dk implicitly includes the encryption key ek required for re-encryption.

[8] To be precise, we will provide the adversary $\mathcal{A}$ against the commitment-binding game oracle access to Enc_s and Dec_s, which are both instantiated using the random oracle to formally invoke the key commitment property of SKE.

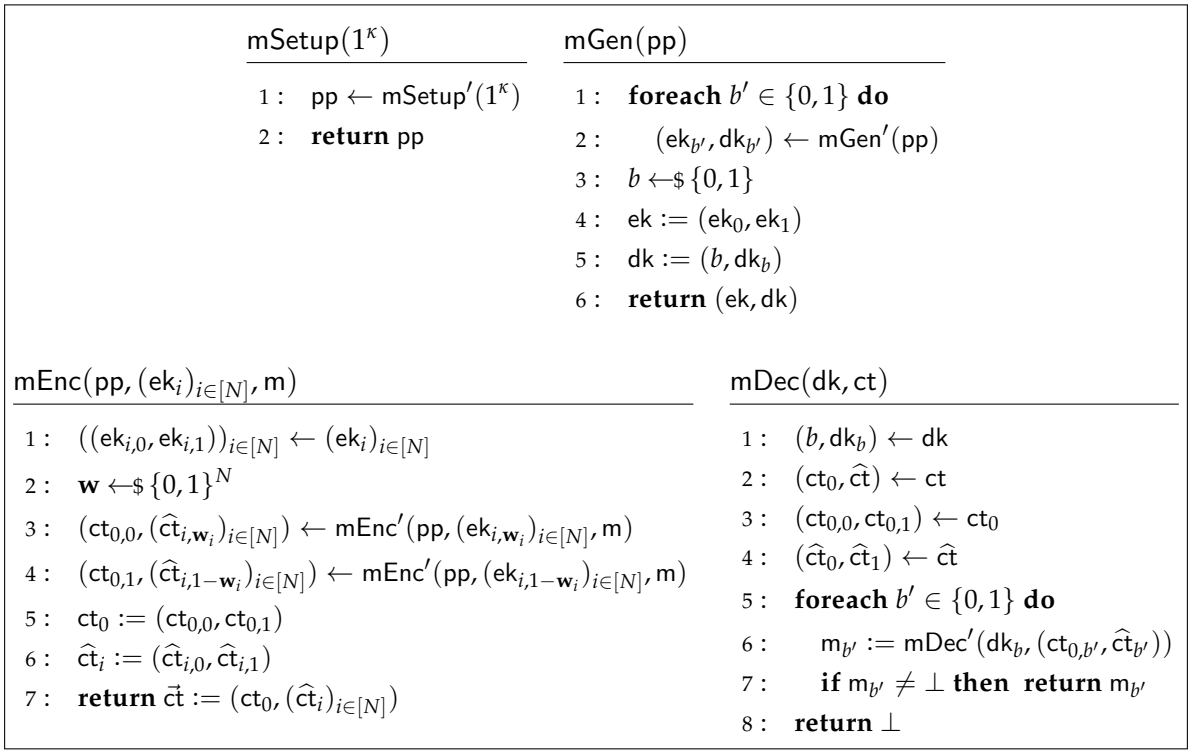


FIGURE 4.6: An IND-CPA secure with adaptive corruption decomposable mPKE from an IND-CPA secure (without adaptive corruption) decomposable mPKE′.

$$\leq \mathbb{E}\left[\max_{\mathsf{ct}^*,\mathsf{m}'\in\mathcal{M}} \Pr_{\mathsf{r}_0,\mathsf{r}'}\left[\mathsf{ct} \leftarrow \left(\mathsf{mEnc}^{\mathsf{i}}(\mathsf{pp};\mathsf{r}_0), \mathsf{mEnc}^{\mathsf{d}}(\mathsf{pp},\mathsf{ek}_i,\mathsf{m}';\mathsf{r}_0,\mathsf{r}')\right) : \mathsf{ct}^* = \mathsf{ct}\right]\right]$$
$$\leq \mathsf{negl}(\kappa).$$

Thus, CmPKE is also ciphertext-spread. □

4.2.3 Construction of CmPKE: IND-CCA with Adaptive Corruption

The construction in Figure 4.4 can be shown to be IND-CCA secure against adaptive corruption by allowing the reduction algorithm to guess the random choices made by the adversary. However, this results in a reduction loss as large as $2^{N \log N}$, where N is the number of recipients. This exponential reduction loss will then be inherited to the CGKA protocol. Although we are unaware of any concrete attacks that take advantage of this large reduction loss, it is natural to ask if there is an efficient *and* provably adaptively secure CmPKE (and hence CGKA) without incurring such a reduction loss.

Due to Theorem 4.2.7, we only need to focus on an IND-CPA secure *with adaptive corruption* decomposable mPKE. Below, we provide a simple generic transformation from any IND-CPA secure decomposable mPKE that is *not* secure against adaptive corruptions into one that is. The overhead is simply doubling the encryption key and ciphertext size, where the transform is a natural adaptation of the Katz–Wang technique [KW03]. The full detail of the construction is provided in Figure 4.6.

It is clear that the construction satisfies correctness. We provide the proof of Lemma 4.2.12, which establishes the IND-CPA security with adaptive corruption.

Lemma 4.2.12. *The decomposable* mPKE *in Figure 4.6 is* IND-CPA *secure with adaptive corruption assuming the decomposable* mPKE' *is* IND-CPA *secure.*

Proof. Let $\mathcal{A}$ be an adversary against the IND-CPA security with adaptive corruption. Consider the following game sequence where the first and last correspond to the case where the challenger uses $b = 0$ and 1 as the challenge, respectively. We denote S_i as the event that $b = b'$ occurs in the game $\mathsf{Game}\ i$.

$\mathsf{Game}\ 0$: This is the real security game where the challenger uses $b = 0$ as its challenge. That is, the challenge ciphertext $\vec{\mathsf{ct}}^*$ encrypts the message m_0.

$\mathsf{Game}\ 1$: We modify how the challenger creates the challenge ciphertext. Let $\mathbf{b} \in \{0,1\}^N$ be the random string associated with the decryption keys of each user. That is, let user i's encryption and decryption keys be $\mathsf{ek}_i := (\mathsf{ek}_{i,0}, \mathsf{ek}_{i,1})$ and $\mathsf{dk}_i := (\mathbf{b}_i, \mathsf{dk}_{i,\mathbf{b}_i})$. Then, when $\mathcal{A}$ outputs $(\mathsf{m}_0, \mathsf{m}_1, S \subseteq [N])$, the challenger creates the challenge ciphertext as

$$(\mathsf{ct}_{0,0}, (\widehat{\mathsf{ct}}_{i,\mathbf{b}_i})_{i\in[S]}) \leftarrow \mathsf{mEnc}'(\mathsf{pp}, (\mathsf{ek}_{i,\mathbf{b}_i})_{i\in[N]}, \mathsf{m}_0)$$
$$(\mathsf{ct}_{0,1}, (\widehat{\mathsf{ct}}_{i,1-\mathbf{b}_i})_{i\in[S]}) \leftarrow \mathsf{mEnc}'(\mathsf{pp}, (\mathsf{ek}_{i,1-\mathbf{b}_i})_{i\in[N]}, \mathsf{m}_0).$$

Recall in the previous game, the challenger sampled a random string $\mathbf{w} \neq \mathbf{b}$ to answer the challenge ciphertext. Due to the winning condition, $\mathcal{A}$ never queries a user $i \in S$ to the corruption oracle. Therefore, $\mathbf{b}_i$ is information-theoretically hidden to $\mathcal{A}$ and the challenge ciphertexts are distributed identically in both games. Therefore, we have $\Pr[\mathsf{S}_0] = \Pr[\mathsf{S}_1]$.

$\mathsf{Game}\ 2$: We further modify how the challenger creates the challenge ciphertext. When $\mathcal{A}$ outputs $(\mathsf{m}_0, \mathsf{m}_1, S \subseteq [N])$, the challenger creates the challenge ciphertext as

$$(\mathsf{ct}_{0,0}, (\widehat{\mathsf{ct}}_{i,\mathbf{b}_i})_{i\in[S]}) \leftarrow \mathsf{mEnc}'(\mathsf{pp}, (\mathsf{ek}_{i,\mathbf{b}_i})_{i\in[N]}, \mathsf{m}_0)$$
$$(\mathsf{ct}_{0,1}, (\widehat{\mathsf{ct}}_{i,1-\mathbf{b}_i})_{i\in[S]}) \leftarrow \mathsf{mEnc}'(\mathsf{pp}, (\mathsf{ek}_{i,1-\mathbf{b}_i})_{i\in[N]}, \mathsf{m}_1).$$

We have $|\Pr[\mathsf{S}_1] - \Pr[\mathsf{S}_2]| \leq \mathsf{negl}(\kappa)$ assuming mPKE' is IND-CPA secure. This can be checked in a straightforward fashion by observing that the only secret information in $\mathsf{Game}\ 2$ is $(\mathsf{dk}_i := (\mathbf{b}_i, \mathsf{dk}_{i,\mathbf{b}_i}))_{i\in[N]}$, which the reduction can simulate on its own. Namely, the reduction embeds the given encryption keys into $\mathsf{ek}_{i,1-\mathbf{b}_i}$.

$\mathsf{Game}\ 3$: We further modify how the challenger creates the challenge ciphertext. When $\mathcal{A}$ outputs $(\mathsf{m}_0, \mathsf{m}_1, S \subseteq [N])$, the challenger creates the challenge ciphertext as

$$(\mathsf{ct}_{0,0}, (\widehat{\mathsf{ct}}_{i,\mathbf{b}_i})_{i\in[S]}) \leftarrow \mathsf{mEnc}'(\mathsf{pp}, (\mathsf{ek}_{i,\mathbf{b}_i})_{i\in[N]}, \mathsf{m}_1)$$
$$(\mathsf{ct}_{0,1}, (\widehat{\mathsf{ct}}_{i,1-\mathbf{b}_i})_{i\in[S]}) \leftarrow \mathsf{mEnc}'(\mathsf{pp}, (\mathsf{ek}_{i,1-\mathbf{b}_i})_{i\in[N]}, \mathsf{m}_0).$$

Swapping the message m_0 and m_1 keeps the distribution of the challenge ciphertext identical following the same argument we made to jump between $\mathsf{Game}\ 0$ and $\mathsf{Game}\ 1$. Hence, $\Pr[\mathsf{S}_2] = \Pr[\mathsf{S}_3]$.

At this point, we simply undo the changes we made. For completeness, we explain the games.

$\mathsf{Game}\ 4$: We make the same change we made in $\mathsf{Game}\ 2$ and swap m_0 to m_1. We have $|\Pr[\mathsf{S}_3] - \Pr[\mathsf{S}_4]| \leq \mathsf{negl}(\kappa)$ assuming mPKE' is IND-CPA secure.

$\mathsf{Game}\ 5$: We make the same change we made in $\mathsf{Game}\ 1$ and use $\mathbf{w}$ instead of $\mathbf{b}$ to answer the challenge ciphertext. We have $\Pr[\mathsf{S}_4] = \Pr[\mathsf{S}_5]$. This corresponds to the real game where $b = 1$ is chosen.

Collecting all the bounds, we conclude $|\Pr[\mathsf{S}_0] - \Pr[\mathsf{S}_5]| \leq \mathsf{negl}(\kappa)$. This completes the proof. □

4.3 Continuous Group Key Agreement

In this section, the syntax and security of continuous group key agreement (CGKA) protocols are defined. We adopt the state-of-the-art syntax and (UC) security model presented in [AJM22], which was used to analyze TreeKEM in MLS version 10. The model presented by [AJM22] is an extension of those presented by [Alw+20b] that further considers *insider* security.

4.3.1 Syntax

Proposal and Commit Paradigm. As in the TreeKEM discussed by the current MLS group, this work follows a 'propose-and-commit' flow where current group members propose to add new members, remove existing ones, or update their own keys by sending *proposal* messages. These proposals only take effect when a group member initiates a new epoch by issuing a commit message, i.e., a special message that commits to the (subset of) proposals. Upon receiving such a commit message, a party applies the committed proposals and transitions to the new epoch.

Akin to the recent specification of TreeKEM and also considered in [AJM22], the proposals must be ordered. Namely, proposals are structured as a vector where it contains all updates, then all removes, and finally, all adds in this order. As done in prior work, the buffering of proposals is delegated to the high-level protocol.

Formal Syntax. We extend the syntax presented in [AJM22] so that the commit and welcome messages can be divided into two parts. Commit (and welcome) message consists of a party *independent* message and a party *dependent* message. When the server receives a request from a party id, it constructs the necessary packets for id from the stored commit message (which is initially created for all the group members) and only sends the portion of the commit message necessary for id. In other words, during a process operation, each receiving party takes a party-independent message and the party-dependent message.

We consider a stateful protocol for a single group that takes the following inputs. Below, assume the protocol knows the party's identity id running the protocol:

Group Creation $(\texttt{Create}, \mathsf{svk})$**:** It initializes a new group state. Only the party id using the verification key svk belongs to this group. In our model, Group Creation is only allowed once.

Add Proposals $(\texttt{Propose}, \texttt{'add'}\text{-}\mathsf{id}_t) \to p$ **:** It outputs a message p proposing to add a party id_t, or $\perp$ if either id is not in the group or it tries to add an id_t that already belongs to the group.

Remove Proposals $(\texttt{Propose}, \texttt{'rem'}\text{-}\mathsf{id}_t) \to p$**:** It outputs a message p proposing to remove a party id_t, or $\perp$if either id is not in the group or it tries to remove an id_t that is not in the group.

Update proposals $(\texttt{Propose}, \texttt{'upd'}\text{-}\mathsf{svk}) \to p$**:** It outputs a message p proposing to update the party id's key material, and optionally the signature key svk, or $\perp$ if id is not in the group.

Commit $(\texttt{Commit}, \vec{p}, \mathsf{svk}) \to (c_0, \vec{c}, w_0, \vec{w})$**:** It commits a vector of proposals $\vec{p}$ and outputs a commit message $(c_0, \vec{c})$. c_0 is a party independent message while $\vec{c} = (\widehat{c}_{\mathsf{id}'})_{\mathsf{id}'}$ is a vector of party dependent messages $\widehat{c}_{\mathsf{id}'}$ designated to id'. If $\vec{p}$ contains at least one add proposal, then it outputs a welcome message $(w_0, \vec{w})$. As in a commit message, w_0 is a party independent message and $\vec{w} = (\widehat{w}_{\mathsf{id}_t})_{\mathsf{id}_t}$ is a vector of party dependent messages. The operation optionally updates the committer's signature key svk.

Process $(\texttt{Process}, c_0, \widehat{c}_{\mathsf{id}}, \vec{p}) \to (\mathsf{id}_c, \mathsf{propSem})$**:** It processes a message $(c_0, \widehat{c}_{\mathsf{id}})$ and committed proposals $\vec{p}$, and advances id to the next epoch. It outputs the committer's identity id_c and the semantics of the applied proposals.

Join $(\texttt{Join}, w_0, \widehat{w}_{\mathsf{id}}) \to (\mathsf{id}_c, \mathsf{mem})$**:** It allows id (who is not yet a group member) to join the group using the welcome message $(w_0, \widehat{w}_{\mathsf{id}})$. It outputs the committer's identity id_c and mem, the set of a pair of identities and signature keys of all group members.

Key $\texttt{Key} \to \mathbf{k}$**:** It outputs the current group key. This can be queried once every epoch by any group member (otherwise returning $\perp$).

We note that the 'add-only' mode of commits in MLS is omitted for simplicity. This allows for a special commit where the proposals $\vec{p}$ consist of all add proposals. In such cases, MLS allows permitting skipping the implicit update performed by the committer. Our construction naturally handles this 'add-only' mode as well.

4.3.2 Security Model

We adopt the universally composable (UC) security model presented in [AJM22] with some modifications. The model of [AJM22] is an extension of the UC model presented in [Alw+20b] that captures the strong notion of *insider security*. Here, a corrupted party can not only send arbitrary network packets but can further interact with the PKI to inject maliciously generated long-term keys and key packages. In our model, since we allow the delivery service (i.e., environment) to sanitize commit messages by delivering to each group member the strict amount of data they need, we further extend [Alw+20b; AJM22] in the following way:

- We extend the input and output interface of the ideal functionality according to the new format of commit messages and welcome messages. The commit function outputs a commit and welcome messages for all the receivers while the process and join functions only take the relevant part of the commit and welcome messages as input. Accordingly, we modify what to store in the commit node of the history graph. (In previous works, since the receiver downloads the same content as those uploaded by the sender, the commit node was simply defined by the uploaded content.) We note that prior constructions can be handled within our new extended model, thus our model is as general as the previous ones.
- We separate the **inj-allowed** predicate into two predicates **sig-inj-allowed** and **mac-inj-allowed**. This is only a conceptual modification but we believe it allows for a more modular proof since we can differentiate between different types of injected messages, i.e., injected by forging a MAC or a signature.

Universal Composable Security. The following description in this sub-section is taken almost verbatim from [AJM22, Sec.2.2 and Sec.3.2]. For further details we refer the readers to [Alw+20b; AJM22].

We formalize security in the generalized universal composability (GUC) framework [Can+07], an extension to the UC framework [Can01]. We moreover use the modification of responsive environments introduced by Camenisch et al. [Cam+16] to avoid artifacts arising from seemingly local operations (such as sampling randomness or producing a ciphertext) to involve the adversary.

The (G)UC framework requires a real-world execution of the protocol to be indistinguishable from an ideal world, to an interactive environment. The real-world experiment consists of the group members executing the protocol (and interacting with the PKI setup). In the ideal world, on the other hand, the protocol is replaced by dummy instances that just forward all inputs and outputs to an ideal functionality characterizing the appropriate guarantees. The functionality interacts with a so-called simulator, that translates the real-world adversary's actions into corresponding ones in the ideal world. Since the ideal functionality is secure by definition, this implies that the real-world execution cannot exhibit any attacks either.

The Corruption Model. We use the — standard for CGKA/SGM but non-standard for UC — corruption model of continuous state leakage (transient passive corruptions) and adversarially chosen randomness of [Alw+20b]. This corruption model allows the adversary to repeatedly corrupt parties by sending them two types of corruption messages: (1) a message `Expose` causes the party to send its current state to the adversary (once), (2) a message $(\texttt{CorrRand}, b)$ sets the party's rand-corrupted flag to b. If b is set, the party's randomness-sampling algorithm is replaced by the adversary providing the coins instead. Ideal functionalities are activated upon corruption and can adjust their behavior accordingly.

Restricted Environments. In order to avoid the so-called commitment problem, caused by adaptive corruptions in simulation-based frameworks, we restrict the environment not to corrupt parties at certain times. We consider a weakened variant of UC security that only quantifies over a restricted set of so-called admissible environments that do not exhibit the commitment problem. Whether an environment is admissible or not is defined by the ideal functionality $\mathcal{F}$ with statements of the form **restrict** *cond* and an environment is called admissible (for $\mathcal{F}$), if it has a negligible probability of violating any such *cond* when interacting with $\mathcal{F}$.

Security via Idealized Services. We consider an ideal CGKA functionality that represents an idealized "CGKA service" agnostic to the usage of the protocol. That is, whenever a party performs a certain group operation (e.g., creating a proposal or commit) the functionality simply hands back an idealized protocol message to that party — it is then up to the environment to deliver those protocol messages to the other group members, thus not making any assumptions on the underlying network or the architecture of the delivery service. Additionally, this also allows us to consider correctness and robustness guarantees, in contrast to more "classical" UC treatments that let the adversary deliver the messages. (Such models typically permit trivial protocols that just reject all messages with the simulator just not delivering them in the ideal world.)

The real world Experiment. In the real world experiment, the parties execute the protocol that furthermore interacts with the Authentication Service (AS) and Key Service (KS) PKI functionalities. For instance, the environment can instruct the Authentication Service (via the party's protocol) to register a new key for a party. As a result, the AS generates a new key pair for the party and hands the public key to the environment, making the secret key available to the party's protocol upon request. The PKI is defined in detail in the next section.

The Ideal World. The ideal world formalizes the security guarantees via the ideal functionality $\mathcal{F}_{\text{CGKA}}$, which internally maintains a so-called history graph. The history graph is a labeled directed graph that acts as a symbolic representation of a group's evolution. It has two types of nodes: commit and proposal nodes, representing all executed commit and propose operations, respectively. Note that each commit node represents an epoch. The nodes' labels, furthermore, keep track of all the additional information relevant to defining security. For instance, proposal nodes have a label that stores the proposed action, and commit nodes have labels that store the epoch's application secret and the set of parties corrupted in the given epoch. Security of the application secrets is then formalized by the functionality choosing a random

and independent key for each commit node whenever security is guaranteed; otherwise, the simulator gets to choose the key. Whether security is guaranteed in a given node, is determined via an explicit safe predicate on the node and the history graph. In addition to the secrecy of the keys, the functionality also formalizes authenticity by appropriately disallowing injections. As the PKI management is exposed to the environment in the real world, we consider "ideal-world variants" of the AS and KS interacting with $\mathcal{F}_{\text{CGKA}}$. Those variants essentially record which keys have been exposed, which in turn is then used to define the safe predicate. The actual keys in the ideal world do not convey any particular meaning beyond serving as identifiers — thus in the ideal world, we can leak all secret keys to the simulator (they are necessary to simulate signatures on protocol messages). We note that this roughly corresponds to treating the PKI setup as local rather than global (in the sense of UC versus GUC).

PKI functionality. We model untrusted PKI, where the adversary can register arbitrary signature keys for any party. This models insider adversary.

Authentication Service (AS). The AS certifies the ownership of a signature key. The functionality $\mathcal{F}_{\text{AS}}$ is defined in Figure 4.7. $\mathcal{F}_{\text{AS}}$ allows a party, identified by id, to register a fresh signature key pair via `register-svk` query and verifies whether a verification key svk is registered by another party via $\mathsf{certSvks}$ query. On registration, the new key pair for a party id is generated by $\mathcal{F}_{\text{AS}}$ using $\mathsf{genSSK}()$ algorithm (which is the parameter of the functionality). If id's current randomness is corrupted (i.e., $\mathsf{Rand}[\mathsf{id}] = \text{'bad'}$), $\mathcal{F}_{\text{AS}}$ asks the adversary to provide the randomness. After registration, the party id receives the new verification key svk. A party id can retrieve its signing keys via `get-ssk` query and delete signing keys via `del-ssk` query.

The adversary can register arbitrary verification keys in the name of any party. Moreover, when a party is corrupted, all signing keys except for the deleted ones are leaked to the adversary. Security is modeled by the ideal-world variant of $\mathcal{F}_{\text{AS}}$, called $\mathcal{F}_{\text{AS}}^{\text{IW}}$. It marks leaked and adversarially registered keys as exposed (see boxes in Figure 4.7).

$\mathcal{F}_{\text{AS}}$ allows the Key Service functionality $\mathcal{F}_{\text{KS}}$ to signal that a certain ssk is leaked. $\mathcal{F}_{\text{KS}}$ sends this signal when the signature key is leaked due to the leakage of key packages.

Finally, $\mathcal{F}_{\text{AS}}^{\text{IW}}$ always leaks all signing keys to the simulator.

Key Service (KS). The KS allows parties to upload one-time key packages used to add them to groups while they are offline.

The KS is formalized by the functionality $\mathcal{F}_{\text{KS}}$ defined in Figure 4.8. Similar to $\mathcal{F}_{\text{AS}}$, a party id can register a key package via `register-kp` query. Upon receiving `register-kp` query, $\mathcal{F}_{\text{KS}}$ generates a new key package using $\mathsf{genKP}(\mathsf{id}, \mathsf{svk}, \mathsf{ssk})$ algorithm (which is the parameter of the functionality), which takes on input the party's identity id and its signature key pair $(\mathsf{svk}, \mathsf{ssk})$ and outputs a key package and the corresponding decryption key. If id's randomness is corrupted, $\mathcal{F}_{\text{KS}}$ uses the randomness provided by the adversary. Moreover, signatures generated with bad randomness may leak the signing key ssk. Hence, $\mathcal{F}_{\text{KS}}$ signals to $\mathcal{F}_{\text{AS}}$ that svk is exposed and sends ssk to the adversary.

Parties can request another party's key package via `get-kp` query. The returned key package is specified by the adversary reflecting that we allow the adversary to maliciously inject key packages that were not registered by honest parties. Finally, parties can retrieve all their (not yet deleted) decryption keys alongside the respective key package via `get-keypair` query. The other queries are analogous to $\mathcal{F}_{\text{AS}}$.

History Graph. As in [AJM22], we use the history graph to manage sent or received messages. A history graph contains proposal nodes and commit nodes. All nodes in the history graph store the following values:

- orig: the identity of the party who created the node, i.e., the message sender.

The functionality is parameterized by a key generation algorithm $\mathsf{genSSK}()$.

Initialization

1 : $\mathsf{RegisteredSvk} \leftarrow \emptyset$; $\boxed{\mathsf{ExposedSvk} \leftarrow \emptyset}$
2 : $\mathsf{SSK}[*,*] \leftarrow \perp$
3 : $\mathsf{Rand}[*] \leftarrow$ 'good'

Inputs from a party id

Input (`register-svk`)

1 : **if** $\mathsf{Rand}[\mathsf{id}] =$ 'good' **then**
2 : $(\mathsf{svk}, \mathsf{ssk}) \leftarrow \mathsf{genSSK}()$
3 : **else**
4 : Send $(\texttt{rnd}, \mathsf{id})$ to $\mathcal{S}$ and receive r
5 : $(\mathsf{svk}, \mathsf{ssk}) \leftarrow \mathsf{genSSK}(\mathsf{r})$
6 : $\boxed{\mathsf{ExposedSvk} \mathrel{+}\!\!\leftarrow \mathsf{svk}}$
7 : $\mathsf{RegisteredSvk} \mathrel{+}\!\!\leftarrow (\mathsf{id}, \mathsf{svk})$
8 : $\mathsf{SSK}[\mathsf{id}, \mathsf{svk}] \leftarrow \mathsf{ssk}$
9 : Send $(\texttt{register-svk}, \mathsf{id}, \mathsf{svk}, \boxed{\mathsf{ssk}})$ to the adversary
10 : Send svk to the party id

Input (`get-ssk`, svk) from id

1 : Send $\mathsf{SSK}[\mathsf{id}, \mathsf{svk}]$ to id

Input (`del-ssk`, svk)

1 : $\mathsf{SSK}[\mathsf{id}, \mathsf{svk}] \leftarrow \perp$

Input (`verify-cert`, id′, svk) from id

1 : Send $(\mathsf{id}', \mathsf{svk}) \in \mathsf{RegisteredSvk}$ to id

Inputs from the adversary

Input (`register-svk`, id, svk)

1 : $\boxed{\textbf{if } (*, \mathsf{svk}) \notin \mathsf{RegisteredSvk} \textbf{ then}}$
2 : $\boxed{\mathsf{ExposedSvk} \mathrel{+}\!\!\leftarrow \mathsf{svk}}$
3 : $\mathsf{RegisteredSvk} \mathrel{+}\!\!\leftarrow (\mathsf{id}, \mathsf{svk})$

Input (`expose-ssk`, id)

1 : $\boxed{\mathsf{ExposedSvk} \mathrel{+}\!\!\leftarrow \{\, \mathsf{svk} \mid \mathsf{SSK}[\mathsf{id}, \mathsf{svk}] \neq \perp \,\}}$
2 : Send $\mathsf{SSK}[\mathsf{id}, *]$ to the adversary

Input $(\texttt{CorrRand}, \mathsf{id}, b)$, $b \in \{$ 'good', 'bad' $\}$

1 : $\mathsf{Rand}[\mathsf{id}] \leftarrow b$

Inputs from $\mathcal{F}_{\mathbf{CGKA}}$ and $\mathcal{F}_{\mathbf{KS}}$

Input (`exposed`, id, svk)

1 : $\boxed{\mathsf{ExposedSvk} \mathrel{+}\!\!\leftarrow \mathsf{svk}}$
2 : Send $\mathsf{SSK}[\mathsf{id}, \mathsf{svk}]$ to the adversary

Inputs from $\mathcal{F}_{\mathrm{CGKA}}$

Input (`has-ssk`, id, svk)

1 : Send $\mathsf{SSK}[\mathsf{id}, \mathsf{svk}] \neq \perp$ to $\mathcal{F}_{\mathrm{CGKA}}$

FIGURE 4.7: The ideal authentication service functionality $\mathcal{F}_{\mathrm{AS}}$ and its variant $\boxed{\mathcal{F}_{\mathrm{AS}}^{\mathrm{IW}}}$ used during the security proof.

The functionality is parameterized by a key-package generation algorithm $\mathsf{genKP}(\mathsf{id}, \mathsf{svk}, \mathsf{ssk})$.

Initialization

1 : $\mathsf{DK}[*,*] \leftarrow \bot; \mathsf{SVK}[*,*] \leftarrow \bot$
2 : $\mathsf{Rand}[*] \leftarrow$ 'good'

Inputs from a party id
Input $(\texttt{register-kp}, \mathsf{svk}, \mathsf{ssk})$

1 : **if** $\mathsf{Rand}[\mathsf{id}] =$ 'good' **then**
2 : $(\mathsf{kp}, \mathsf{dk}) \leftarrow \mathsf{genKP}(\mathsf{id}, \mathsf{svk}, \mathsf{ssk})$
3 : **if** $\mathsf{kp} = \bot$ **then return**
4 : **else**
5 : Send $(\texttt{rnd}, \mathsf{id})$ to $\mathcal{S}$ and receive r
6 : $(\mathsf{kp}, \mathsf{dk}) \leftarrow \mathsf{genKP}(\mathsf{id}, \mathsf{svk}, \mathsf{ssk}; \mathsf{r})$
7 : **if** $\mathsf{kp} = \bot$ **then return**
8 : Send $(\texttt{exposed}, \mathsf{id}, \mathsf{svk})$ to $\mathcal{F}_{\text{AS}}$
9 : $\mathsf{DK}[\mathsf{id}, \mathsf{kp}] \leftarrow \mathsf{dk}; \mathsf{SVK}[\mathsf{id}, \mathsf{kp}] \leftarrow \mathsf{svk}$
10 : Send $(\texttt{register-kp}, \mathsf{id}, \mathsf{svk}, \mathsf{kp}, \boxed{\mathsf{dk}})$ to the adversary
11 : Send kp to the party id

Input $(\texttt{get-dks})$

1 : Send $\{ (\mathsf{kp}, \mathsf{DK}[\mathsf{id}, \mathsf{kp}]) \mid \mathsf{DK}[\mathsf{id}, \mathsf{kp}] \neq \bot \}$ to id

Input $(\texttt{get-kp}, \mathsf{id}')$

1 : Send $(\texttt{get-kp}, \mathsf{id}, \mathsf{id}')$ to $\mathcal{S}$ and receive kp
2 : Send kp to id

Input $(\texttt{del-kp}, \mathsf{kp})$

1 : $\mathsf{DK}[\mathsf{id}, \mathsf{kp}] \leftarrow \bot$

Inputs from the adversary
Input $(\texttt{CorrRand}, \mathsf{id}, b)$, $b \in \{$ 'good', 'bad' $\}$

1 : $\mathsf{Rand}[\mathsf{id}] \leftarrow b$

Inputs from the adversary and $\mathcal{F}_{\text{CGKA}}$
Input $(\texttt{exposed}, \mathsf{id})$

1 : Send $\mathsf{DK}[\mathsf{id}, *]$ to the adversary
2 : **foreach** $\mathsf{svk} \in \mathsf{SVK}[\mathsf{id}, *]$ **s.t.** $\mathsf{svk} \neq \bot$ **do**
3 : Send $(\texttt{exposed}, \mathsf{id}, \mathsf{svk})$ to $\mathcal{F}_{\text{AS}}$

FIGURE 4.8: The ideal key service functionality $\mathcal{F}_{\text{KS}}$ and its variant $\boxed{\mathcal{F}_{\text{KS}}^{\text{IW}}}$ used during the security proof.

- par: the parent commit node, representing the sender's current epoch.
- stat $\in$ { 'good', 'bad', 'adv' }: the status flag indicating whether the secrets corresponding to the node are known to the adversary. 'good' means this node is secure, 'bad' means this node is created with adversarial randomness (hence it is well-formed but the adversary knows the secret), and 'adv' means this node is created by the injected message from the adversary.

Proposal nodes further store the following values:

- act $\in$ { 'upd'-svk, 'add'-id_t-svk_t, 'rem'-id_t }: the proposal action. The history graph also stores the signature verification key svk. 'add'-id_t-svk_t means id_t is added with the verification key svk_t.

Commit nodes further store the following values:

- prop: the ordered list of committed proposals.
- mem: the list of a pair of group members' identities and their signature verification keys.
- vcom: the list of party-specific commitments associated with the node c_0.
- key: the group (application) secret key.
- chall: the flag indicating whether the group key is challenged. That is, chall = `true` if a random group key was generated for this node, and `false` if the key was set by the adversary (or not generated).
- exp: the set keeping track of corrupted parties in this node. It includes a flag whether only their secret state is leaked (the flag is false), or also the current group key is leaked (the flag is true).

For convenience, we define the following helper function.

- indexOf(id): returns the index of id in the list mem.

CGKA Functionality. Using the history graph and the PKI functionality, we introduce the ideal functionality $\mathcal{F}_{\text{CGKA}}$, formally defined in Figures 4.9 to 4.11 with the helper functions in Figures 4.12 to 4.14. $\mathcal{F}_{\text{CGKA}}$ is parameterized by the predicates **safe**, **sig-inj-allowed** and **mac-inj-allowed**, which specify which epoch secrets are secure and when authenticity is guaranteed. The predicates are defined in Figure 4.25. In previous works [Alw+20b; AJM22], **sig-inj-allowed** and **mac-inj-allowed** were handled by a single predicate **inj-allowed** that checked any injection regardless of it being a forgery of the MAC or signature. We intentionally divide the **inj-allowed** predicate into two predicates to make the proof more accessible. This modification is merely conceptual. Moreover, we add an explicit additional check for **sig-inj-allowed** in case id is assigned to a detached root (highlighted in Figure 4.14). This was implicitly checked by the simulator in previous works and we only made it explicit. Namely, the inclusion of this check is aimed to improve the readability of the ideal functionality and has no effect on security.

Below, we extend the input and output interface of the functionality according to the new format of commit messages and welcome messages.

States. $\mathcal{F}_{\text{CGKA}}$ maintains the history graph. It addresses proposal nodes by (idealized) proposal message p and non-root commit nodes by (idealized) proposal messages c_0. We consider the single main group. The root node corresponding to the main group is addressed by the special label root_0. Moreover, other roots may be created without a commit message (e.g., when a party uses an injected welcome message to an

adversarially created epoch, which is not directly related to the main group). Such roots are addressed by the special labels root_{rt} for $rt \in \mathbb{N}$ and their tree are called *detached*.

$\mathcal{F}_{\text{CGKA}}$ also stores a pointer $\mathsf{Ptr}[\mathsf{id}]$ for each party id. $\mathsf{Ptr}[\mathsf{id}]$ indicates id's current commit node (i.e., current epoch). If id currently is not in the group, $\mathsf{Ptr}[\mathsf{id}] = \bot$.

Interfaces. $\mathcal{F}_{\text{CGKA}}$ offers interfaces for creating a group, creating a proposal, committing a list of proposals, processing a commit, joining a group, and retrieving the current group key. We assume the main group is created by the designated party $\mathsf{id}_{\mathsf{creator}}$. Initially, the main group has a single party $\mathsf{id}_{\mathsf{creator}}$, and it can invite additional members. All interfaces except create and join are for group members only (i.e., parties for which $\mathsf{Ptr}[\mathsf{id}] \neq \bot$).

Proposals. When a party id creates a proposal, $\mathcal{F}_{\text{CGKA}}$ notifies the adversary. Then it returns a flag *ack*, a node identifier p (i.e., a message) and a signature verification key svk_t. $\mathcal{F}_{\text{CGKA}}$ allows the adversary to send $\mathit{ack} = \texttt{false}$ to report that the protocol fails, i.e., the output is $p = \bot$. If the protocol succeeds, and if no node with identifier p exists, $\mathcal{F}_{\text{CGKA}}$ creates a new proposal node $\mathsf{Prop}[p]$. For add proposals, it extends the action by the verification key svk_t (specified by the adversary) of the added party id_t.

In certain situations, $\mathcal{F}_{\text{CGKA}}$ may not create a new proposal node. For example, id proposes to remove the same party twice in the same epoch. Another such situation is that a party proposes to update using the same randomness. In these cases, the adversary can specify the preexisting p. $\mathcal{F}_{\text{CGKA}}$ enforces that the states on the existing node are consistent with the expected one using `*consistent-prop`.

Finally, $\mathcal{F}_{\text{CGKA}}$ returns the proposal identifier p to the calling party id.

Commits. When id creates a commit message, it specifies a list of proposals $\vec{p}$, a (possibly fresh) signature verification key svk. Then $\mathcal{F}_{\text{CGKA}}$ forwards all inputs to the adversary and receives a flag *ack* and identifiers c_0 of commit node with a list $\vec{c}$ and w_0 of a welcome node with a list $\vec{w}$. $\vec{c}$ (resp. $\vec{w}$) contains the party dependent information $\widehat{c}_{\mathsf{id}}$ (resp. $\widehat{w}_{\mathsf{id}}$), and it is used when id processes the message c_0 (resp. w_0). The adversary sets $\mathit{ack} := \texttt{false}$ to report that the protocol fails. If the commit protocol succeeds, $\mathcal{F}_{\text{CGKA}}$ first asks the adversary to interpret the injected proposals, i.e., proposal where no node has been created, by calling `*fill-prop`. It then computes the member set resulting from applying $\vec{p}$ by calling `*next-members` (which returns $\bot$ if $\vec{p}$ is invalid).

Then $\mathcal{F}_{\text{CGKA}}$ either creates a new commit node or verifies that the existing node is consistent (cf. `*consistent-com`). It may happen that the existing node is the detached root. In such case, $\mathcal{F}_{\text{CGKA}}$ attaches it to id's current node calling `*attach`. This helper assigns c_0 as the proper identifier of the detached root and deletes the root. Once the detached root is attached, the root's tree achieves the same security guarantee as the main group. Since attaching a detached root changes the history graph, $\mathcal{F}_{\text{CGKA}}$ enforces two invariants: `cons-invariant` enforcing the consistency of the graph, and `auth-invariant` enforcing the authenticity guarantee.

Finally, when add proposals are committed, $\mathcal{F}_{\text{CGKA}}$ records the welcome message that leads the new member to the created commit node. Then $\mathcal{F}_{\text{CGKA}}$ returns $(c_0, \vec{c}, w_0, \vec{w})$ to the calling party id.

Processing Commits. When id processes a commit message, it specifies the commit message $(c_0, \widehat{c})$, where $\widehat{c}$ is the id-dependent message and a list of committed proposals $\vec{p}$. Then $\mathcal{F}_{\text{CGKA}}$ forwards all the inputs to the adversary and receives the interpreted result from $(c_0, \widehat{c})$.

If the processing succeeds, $\mathcal{F}_{\text{CGKA}}$ either creates a new commit node or verifies that the existing node is consistent. If corresponding nodes do not exist, $\mathcal{F}_{\text{CGKA}}$ checks the validity of $\vec{p}$ and creates a new commit node with the committer identity orig' and its signature key svk' which are interpreted by the adversary from $(c_0, \widehat{c})$. If the node $\mathsf{Node}[c_0] \neq \bot$ exists, $\mathcal{F}_{\text{CGKA}}$ enforces that it is a valid successor of id's current node (cf. `*valid-successor`). If c_0 matches a detached root, $\mathcal{F}_{\text{CGKA}}$ attaches them.

Finally, depending on whether c_0 removes id, $\mathcal{F}_{\text{CGKA}}$ either moves id's pointer $\mathsf{Ptr}[\mathsf{id}]$ to the new node or sets the pointer to $\perp$. The calling party receives the committer's identity and the semantics of the applied proposals.

Joining. When a party id joins a group, it specifies the welcome message w_0 and the id-dependent message $\widehat{w}$. Then $\mathcal{F}_{\text{CGKA}}$ forwards all the inputs to the adversary and receives the interpreted result from $(w_0, \widehat{w})$. As usual, the adversary sets *ack* := `false` to report that the protocol fails.

If the processing succeeds, $\mathcal{F}_{\text{CGKA}}$ identifies the commit $c_0 = \mathsf{Wel}[w_0]$ corresponding to w_0. If this is the first time $\mathcal{F}_{\text{CGKA}}$ sees w_0, i.e., $\mathsf{Wel}[w_0] = \perp$, the adversary chooses c_0'. If the commit node for c_0' does not exist (i.e., $\mathsf{Node}[c_0'] = \perp$), $\mathcal{F}_{\text{CGKA}}$ creates a new detached root where all stored values are chosen by the adversary.

Finally, $\mathcal{F}_{\text{CGKA}}$ returns the state of the joining group (the committer's identity and the list of $(\mathsf{id}, \mathsf{svk})$-pair) to the calling party id.

Group keys and Corruptions. Parties can fetch the current group key via `Key` query. The Key is random if the protocol guarantees its secrecy as identified by the **safe** predicate. Otherwise, the key is set by the adversary.

The predicate **safe** uses information that is recorded by $\mathcal{F}_{\text{CGKA}}$. When the state of a current group member id is exposed, $\mathcal{F}_{\text{CGKA}}$ records leakage of the following information.

- The current group key (if not retrieved yet) and any key materials (e.g., encryption key and signing key) to process future messages. This is recorded by adding the pair $(\mathsf{id}, \mathsf{HasKey}[\mathsf{id}])$ to the exposed set of id's current node (cf. line 2 in $(\texttt{Expose}, \mathsf{id})$). The flag $\mathsf{HasKey}[\mathsf{id}]$ indicates whether id currently stores the group key (if the group key has not been calculated yet or was already retrieved, $\mathsf{HasKey}[\mathsf{id}] =$ `false`).

- The key material for updates and commits created by id in the current epoch. This is recorded by setting the status of all child nodes created by id (i.e., nodes with $\mathsf{par} = \mathsf{Ptr}[\mathsf{id}]$) to 'bad' (cf. `*update-stat-after-exp` function).

- The current signature signing key ssk. This is recorded by signaling to $\mathcal{F}_{\text{AS}}$ that svk is exposed and sends ssk to the adversary (cf. line 5 in $(\texttt{Expose}, \mathsf{id})$).

In addition, exposure of party id who is not a group member reveals key packages that will be used to process welcome messages. $\mathcal{F}_{\text{CGKA}}$ signals to $\mathcal{F}_{\text{KS}}$ that key packages (including signing key) are exposed and sends the corresponding decryption keys and signing keys to the adversary (cf. line 6 in $(\texttt{Expose}, \mathsf{id})$).

Adaptive corruptions become a problem if the adversary reveals a key material that can be used to compute a group key that has already been outputted as random by $\mathcal{F}_{\text{CGKA}}$, i.e., the challenge key. Hence, we restrict the environment, so as not to corrupt key materials such that it would cause **safe** switching to false for some commit nodes with $\mathsf{chall} =$ `true`.

Initialization

$\mathsf{Ptr}[*], \mathsf{Node}[*], \mathsf{Prop}[*], \mathsf{Wel}[*] \leftarrow \bot$
$\mathsf{Rand}[*] \leftarrow$ 'good'; $\mathsf{HasKey}[*] \leftarrow \texttt{false}$; $\mathsf{rootCtr} \leftarrow 0$
// Flag is set to `true` if selective downloading is performed.
$\mathsf{flag}_{\mathsf{selDL}} = \texttt{true}$

Inputs from a party $\mathsf{id}_{\mathsf{creator}}$

Input $(\texttt{Create}, \mathsf{svk})$

req $\mathsf{Node}[\mathsf{root}_0] = \bot$
req $\texttt{*valid-svk}(\mathsf{id}_{\mathsf{creator}}, \mathsf{svk})$
$\mathsf{mem} \leftarrow \{ (\mathsf{id}_{\mathsf{creator}}, \mathsf{svk}) \}$
$\mathsf{Node}[\mathsf{root}_0] \leftarrow \texttt{*create-root}(\mathsf{id}_{\mathsf{creator}}, \mathsf{mem}, \mathsf{Rand}[\mathsf{id}_{\mathsf{creator}}])$
$\mathsf{HasKey}[\mathsf{id}_{\mathsf{creator}}] \leftarrow \texttt{true}$; $\mathsf{Ptr}[\mathsf{id}_{\mathsf{creator}}] \leftarrow \mathsf{root}_0$
Send $(\texttt{Create}, \mathsf{id}_{\mathsf{creator}}, \mathsf{svk})$ to the adversary

Inputs from a party id

Input $(\texttt{Propose}, \mathsf{act}^{\dagger})$

req $\mathsf{Ptr}[\mathsf{id}] \neq \bot$
Send $(\texttt{Propose}, \mathsf{id}, \mathsf{act})$ to $\mathcal{S}$ and
  receive (ack, p, svk_t)
req ack
if act = 'upd'-svk **then req** $\texttt{*valid-svk}(\mathsf{id}, \mathsf{svk})$
if act = 'add'-id_t **then** act $\leftarrow$ 'add'-id_t-svk_t
if $\mathsf{Prop}[p] = \bot$ **then**
  $\mathsf{Prop}[p] \leftarrow \texttt{*create-prop}(\mathsf{Ptr}[\mathsf{id}], \mathsf{id}, \mathsf{act}, \mathsf{Rand}[\mathsf{id}])$
else
  $\texttt{*consistent-prop}(p, \mathsf{id}, \mathsf{act})$
if act = 'upd'-svk $\wedge$ $\mathsf{Rand}[\mathsf{id}]$ = 'bad' **then**
  Send $(\texttt{exposed}, \mathsf{id}, \mathsf{svk})$ to $\mathcal{F}_{\mathrm{AS}}$
return p

Input $(\texttt{Commit}, \vec{p}, \mathsf{svk})$

req $\mathsf{Ptr}[\mathsf{id}] \neq \bot$
Send $(\texttt{Commit}, \mathsf{id}, \vec{p}, \mathsf{svk})$ to $\mathcal{S}$ and
  receive $(ack, rt, c_0, \vec{c}, w_0, \vec{w})$
req $\texttt{*succeed-com}(\mathsf{id}, \vec{p}, \mathsf{svk}) \vee ack$
$\texttt{*fill-prop}(\mathsf{id}, \vec{p})$
req $\texttt{*valid-svk}(\mathsf{id}, \mathsf{svk})$
$(\mathsf{mem}, *) \leftarrow \texttt{*next-members}(\mathsf{Ptr}[\mathsf{id}], \mathsf{id}, \vec{p}, \mathsf{svk})$
assert $\mathsf{mem} \neq \bot \wedge (\mathsf{id}, \mathsf{svk}) \in \mathsf{mem}$
// If selective downloading is performed,
// then party-specific $\vec{c}$ has the same size
// as the current member. Otherwise, $\vec{c}$ is $\bot$
if $\mathsf{flag}_{\mathsf{selDL}}$ **then**
  assert $|\vec{c}| = |\mathsf{Node}[\mathsf{Ptr}[\mathsf{id}]].\mathsf{mem}|$
else
  assert $\vec{c} = \bot$
if $\mathsf{Node}[c_0] = \bot \wedge rt = \bot$ **then**
  $\mathsf{Node}[c_0] \leftarrow \texttt{*create-child}(\mathsf{Ptr}[\mathsf{id}], \mathsf{id}, \vec{p}, \vec{c}, \mathsf{mem}, \mathsf{R}_{\mathsf{id}})$
  $\mathsf{R}_{\mathsf{id}} \leftarrow \mathsf{Rand}[\mathsf{id}]$
  if $w_0 \neq \bot$ **then**
    assert $\mathsf{Wel}[w_0] = \bot$
    $\mathsf{Wel}[w_0] \leftarrow c_0$
else
  if $\mathsf{Node}[c_0] = \bot$ **then** $c_0' \leftarrow \mathsf{root}_{rt}$
  else $c_0' \leftarrow c_0$
  $\texttt{*consistent-com}(c_0', \mathsf{id}, \vec{p}, \mathsf{mem})$
  if $c_0' = \mathsf{root}_{rt}$ **then** $\texttt{*attach}(c_0, c_0', \mathsf{id}, \vec{p})$
  if $w_0 \neq \bot$ **then**
    assert $\mathsf{Wel}[w_0] \in \{ \bot, c_0 \}$
    $\mathsf{Wel}[w_0] \leftarrow c_0$
assert `cons-invariant` $\wedge$ `auth-invariant`
if $\mathsf{Rand}[\mathsf{id}]$ = 'bad' **then**
  Send $(\texttt{exposed}, \mathsf{id}, \mathsf{svk})$ to $\mathcal{F}_{\mathrm{AS}}$
return $(c_0, \vec{c}, w_0, \vec{w})$

FIGURE 4.9: The ideal CGKA functionality $\mathcal{F}_{\mathrm{CGKA}}$: `Create`, `Propose` and `Commit` functions. †:act $\in$ { 'upd'-svk, 'add'-id_t, 'rem'-id_t }.

Input $(\texttt{Process}, c_0, \widehat{c}, \vec{p})$

1 : **req** $\mathsf{Ptr}[\mathsf{id}] \neq \bot$
2 : Send $(\texttt{Process}, \mathsf{id}, c_0, \widehat{c}, \vec{p})$ to $\mathcal{S}$ and
receive $(ack, rt, \mathsf{orig}', \mathsf{svk}')$
3 : **req** $\texttt{*succeed-proc}(\mathsf{id}, c_0, \widehat{c}, \vec{p}) \vee ack$
4 : $\texttt{*fill-prop}(\mathsf{id}, \vec{p})$
5 : // If selective downloading is performed,
// then only the member-specific commitment $\widehat{c}$ is accepted
// when c_0 is honestly generated.
if $\mathtt{flag}_{\mathsf{selDL}}$ **then**
6 : **if** $\mathsf{Node}[c_0].\mathsf{stat} =$ 'good' **then**
7 : $\mathsf{index}_{\mathsf{id}} \leftarrow \mathsf{Node}[\mathsf{Ptr}[\mathsf{id}]].\mathsf{indexOf}(\mathsf{id})$
8 : **req** $\widehat{c} = \mathsf{Node}[c_0].\mathsf{vcom}[\mathsf{index}_{\mathsf{id}}]$
9 : **else**
10 : // Otherwise, $\widehat{c}$ must be $\bot$.
11 : **assert** $\widehat{c} = \bot$
12 : **if** $\mathsf{Node}[c_0] = \bot \wedge rt = \bot$ **then**
13 : $(\mathsf{mem}, *) \leftarrow \texttt{*next-members}(\mathsf{Ptr}[\mathsf{id}], \mathsf{orig}', \vec{p}, \mathsf{svk}')$
14 : **assert** $\mathsf{mem} \neq \bot$
15 : $\mathsf{Node}[c_0]$
$\leftarrow \texttt{*create-child}(\mathsf{Ptr}[\mathsf{id}], \mathsf{orig}', \vec{p}, \mathsf{mem}, \text{'adv'})$
16 : **else**
17 : **if** $\mathsf{Node}[c_0] = \bot$ **then** $c_0' \leftarrow \mathsf{root}_{rt}$
18 : **else** $c_0' \leftarrow c_0$
19 : $\mathsf{id}_c \leftarrow \mathsf{Node}[c_0'].\mathsf{orig}; \mathsf{svk}_c \leftarrow \mathsf{Node}[c_0'].\mathsf{mem}[\mathsf{id}_c]$
20 : $(\mathsf{mem}, *) \leftarrow \texttt{*next-members}(\mathsf{Ptr}[\mathsf{id}], \mathsf{id}_c, \vec{p}, \mathsf{svk}_c)$
21 : **assert** $\mathsf{mem} \neq \bot$
22 : $\texttt{*valid-successor}(c_0', \mathsf{id}_c, \vec{p}, \mathsf{mem})$
23 : **if** $c_0' = \mathsf{root}_{rt}$ **then** $\texttt{*attach}(c_0, c_0', \mathsf{id}, \vec{p})$
24 : **if** $\exists p \in \vec{p} : \mathsf{Prop}[p].\mathsf{act} =$ 'rem'-id **then**
25 : $\mathsf{Ptr}[\mathsf{id}] \leftarrow \bot$
26 : **else**
27 : **assert** $(\mathsf{id}, *) \in \mathsf{Node}[c_0].\mathsf{mem}$
28 : $\mathsf{Ptr}[\mathsf{id}] \leftarrow c_0; \mathsf{HasKey}[\mathsf{id}] \leftarrow \texttt{true}$
29 : **assert** $\texttt{cons-invariant} \wedge \texttt{auth-invariant}$
30 : **return** $\texttt{*output-proc}(c_0)$

Input $(\texttt{Join}, w_0, \widehat{w})$

1 : **req** $\mathsf{Ptr}[\mathsf{id}] = \bot$
2 : Send $(\texttt{Join}, \mathsf{id}, w_0, \widehat{w})$ to $\mathcal{S}$ and
receive $(ack, c_0', \mathsf{orig}', \mathsf{mem}')$
3 : **req** $\texttt{*succeed-wel}(\mathsf{id}, w_0, \widehat{w}) \vee ack$
4 : $c_0 \leftarrow \mathsf{Wel}[w_0]$
5 : **if** $c_0 = \bot$ **then**
6 : **if** $\mathsf{Node}[c_0'] \neq \bot$ **then** $c_0 \leftarrow c_0'$
7 : **else**
8 : rootCtr++
9 : // Assume root_i are reserved words
10 : $c_0 \leftarrow \mathsf{root}_{\mathsf{rootCtr}}$
11 : $\mathsf{Node}[c_0]$
$\leftarrow \texttt{*create-root}(\mathsf{orig}', \mathsf{mem}', \text{'adv'})$
12 : $\mathsf{Wel}[w_0] \leftarrow c_0$
13 : **assert** $(\mathsf{id}, *) \in \mathsf{Node}[c_0].\mathsf{mem}$
14 : $\mathsf{Ptr}[\mathsf{id}] \leftarrow c_0$
15 : $\mathsf{HasKey}[\mathsf{id}] \leftarrow \texttt{true}$
16 : **assert** $\texttt{cons-invariant} \wedge \texttt{auth-invariant}$
17 : **return** $(\mathsf{Node}[c_0].\mathsf{orig}, \mathsf{Node}[c_0].\mathsf{mem})$

Input Key

1 : **req** $\mathsf{Ptr}[\mathsf{id}] \neq \bot \wedge \mathsf{HasKey}[\mathsf{id}]$
2 : **if** $\mathsf{Node}[\mathsf{Ptr}[\mathsf{id}]].\mathsf{key} = \bot$ **then**
3 : $\texttt{*set-key}(\mathsf{Ptr}[\mathsf{id}])$
4 : $\mathsf{HasKey}[\mathsf{id}] \leftarrow \texttt{false}$
5 : **return** $\mathsf{Node}[\mathsf{Ptr}[\mathsf{id}]].\mathsf{key}$

FIGURE 4.10: The ideal CGKA functionality $\mathcal{F}_{\text{CGKA}}$: Process and Join functions.

Input $(\texttt{Expose}, \mathsf{id})$

1 : **if** $\mathsf{Ptr}[\mathsf{id}] \neq \bot$ **then**
2 : $\quad \mathsf{Node}[\mathsf{Ptr}[\mathsf{id}]].\mathsf{exp} \mathrel{+}\!\leftarrow (\mathsf{id}, \mathsf{HasKey}[\mathsf{id}])$
3 : $\quad$ `*update-stat-after-exp`(id)
4 : $\quad \mathsf{svk} \leftarrow \mathsf{Node}[\mathsf{Ptr}[\mathsf{id}]].\mathsf{mem}[\mathsf{id}]$
5 : $\quad$ Send $(\texttt{exposed}, \mathsf{id}, \mathsf{svk})$ to $\mathcal{F}_{\mathrm{AS}}$
6 : Send $(\texttt{exposed}, \mathsf{id})$ to $\mathcal{F}_{\mathrm{KS}}$
7 : **restrict** :
8 : $\quad \forall c_0$: **if** $\mathsf{Node}[c_0].\mathsf{chall} = \texttt{true}$ **then** $\mathbf{safe}(c_0) = \texttt{true}$

Input $(\texttt{CorrRand}, \mathsf{id}, b)$, $b \in \{$ 'good', 'bad' $\}$

1 : $\mathsf{Rand}[\mathsf{id}] \leftarrow b$

FIGURE 4.11: The CGKA functionality $\mathcal{F}_{\mathrm{CGKA}}$: Corruptions from the adversary.

`*create-root`$(\mathsf{id}, \mathsf{mem}, \mathsf{stat})$

1 : **return** new node with $\mathsf{par} \leftarrow \bot, \mathsf{orig} \leftarrow \mathsf{id}$, $\mathsf{prop} \leftarrow \bot, \mathsf{mem} \leftarrow \mathsf{mem}, \mathsf{stat} \leftarrow \mathsf{stat}$.

`*create-child`$(c_0, \mathsf{id}, \vec{p}, \vec{c}, \mathsf{mem}, \mathsf{stat})$

1 : **return** new node with $\mathsf{par} \leftarrow c_0, \mathsf{orig} \leftarrow \mathsf{id}$, $\mathsf{prop} \leftarrow \vec{p}, \mathsf{vcom} \leftarrow \vec{c}, \mathsf{mem} \leftarrow \mathsf{mem}, \mathsf{stat} \leftarrow \mathsf{stat}$.

`*create-prop`$(c_0, \mathsf{id}, \mathsf{act}, \mathsf{stat})$

1 : **return** new node with $\mathsf{par} \leftarrow c_0, \mathsf{orig} \leftarrow \mathsf{id}$, $\mathsf{act} \leftarrow \mathsf{act}, \mathsf{stat} \leftarrow \mathsf{stat}$.

`*fill-prop`$(\mathsf{id}, \vec{p})$

1 : **foreach** $p \in \vec{p}$ **s.t.** $\mathsf{Prop}[p] = \bot$ **do**
2 : $\quad$ Send $(\texttt{Propose}, p)$ to $\mathcal{S}$ and
3 : $\quad\quad$ receive $(\mathsf{orig}, \mathsf{act})$
4 : $\quad \mathsf{Prop}[p] \leftarrow$ `*create-prop`$(\mathsf{Ptr}[\mathsf{id}], \mathsf{orig}, \mathsf{act}, \text{'adv'})$

`*output-proc`(c_0)

1 : $\mathsf{id}_c \leftarrow \mathsf{Node}[c_0].\mathsf{orig}$
2 : $\mathsf{svk}_c \leftarrow \mathsf{Node}[c_0].\mathsf{mem}[\mathsf{id}_c]$
3 : $p \leftarrow \mathsf{Node}[c_0].\mathsf{prop}$
4 : $(*, \mathsf{propSem}) \leftarrow$ `*next-members`$(c_0, \mathsf{id}_c, p, \mathsf{svk}_c)$
5 : **return** $(\mathsf{Node}[c_0].\mathsf{orig}, \mathsf{propSem})$

`*valid-svk`$(\mathsf{id}, \mathsf{svk}')$

1 : **if** $\mathsf{Ptr}[\mathsf{id}] \neq \bot$ **then**
2 : $\quad \mathsf{svk} \leftarrow \mathsf{Node}[\mathsf{Ptr}[\mathsf{id}]].\mathsf{mem}[\mathsf{id}]$
3 : $\quad$ **if** $\mathsf{svk} \neq \bot \wedge \mathsf{svk} = \mathsf{svk}'$ **then return** `true`
4 : Send $(\texttt{has-ssk}, \mathsf{id}, \mathsf{svk}')$ to $\mathcal{F}_{\mathrm{AS}}$ and receive *ack*
5 : **return** *ack*

`*set-key`(c_0)

1 : **if** $\mathbf{safe}(c_0)$ **then**
2 : $\quad \mathsf{Node}[c_0].\mathsf{key} \leftarrow\!\$\, \mathcal{K}; \mathsf{Node}[c_0].\mathsf{chall} \leftarrow \texttt{true}$
3 : **else**
4 : $\quad$ Send $(\texttt{Key}, \mathsf{id})$ to $\mathcal{S}$ and receive k
5 : $\quad \mathsf{Node}[c_0].\mathsf{key} \leftarrow \mathsf{k}; \mathsf{Node}[c_0].\mathsf{chall} \leftarrow \texttt{false}$

`*update-stat-after-exp`(id)

1 : **foreach** p **s.t.** $\mathsf{Prop}[p] \neq \bot \wedge \mathsf{Prop}[p].\mathsf{par} = \mathsf{Ptr}[\mathsf{id}] \wedge \mathsf{Prop}[p].\mathsf{orig} = \mathsf{id} \wedge \mathsf{Prop}[p].\mathsf{act} =$ 'upd'-$*$ **do**
2 : $\quad \mathsf{Prop}[p].\mathsf{stat} \leftarrow$ 'bad'
3 : **foreach** c_0 **s.t.** $\mathsf{Node}[c] \neq \bot \wedge \mathsf{Node}[c].\mathsf{par} = \mathsf{Ptr}[\mathsf{id}] \wedge \mathsf{Node}[c].\mathsf{orig} = \mathsf{id}$ **do**
4 : $\quad \mathsf{Node}[c].\mathsf{stat} \leftarrow$ 'bad'

FIGURE 4.12: The helper functions for $\mathcal{F}_{\mathrm{CGKA}}$: Creating and maintaining the history graph.

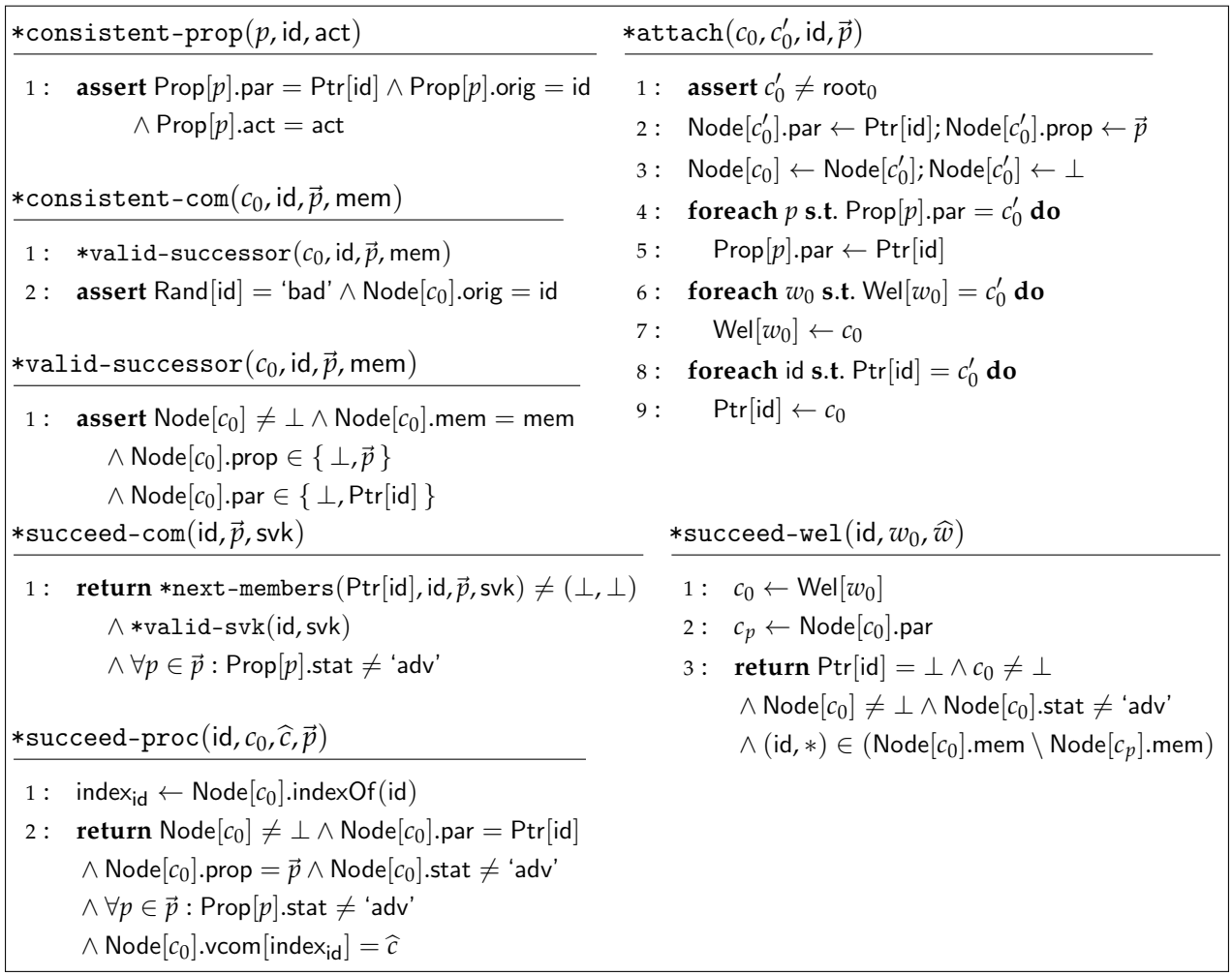


FIGURE 4.13: The helper functions $\mathcal{F}_{\text{CGKA}}$: Checking consistency and correctness.

`*next-members`$(c_0, \mathsf{id}_c, \vec{p}, \mathsf{svk}_c)$

1 : **if** $\mathsf{Node}[c_0] \neq \bot \wedge (\mathsf{id}_c, *) \in \mathsf{Node}[c_0].\mathsf{mem}$
$\wedge \forall p \in \vec{p} : \mathsf{Prop}[p] \neq \bot \wedge \mathsf{Prop}[p].\mathsf{par} = c_0$
$\wedge \vec{p} = \vec{p}_{\text{'upd'}} \| \vec{p}_{\text{'rem'}} \| \vec{p}_{\text{'add'}}$
for some $\vec{p}_{\text{'upd'}}, \vec{p}_{\text{'rem'}}, \vec{p}_{\text{'add'}}$
$\wedge \forall \mathsf{act}(\forall p \in \vec{p}_{\mathsf{act}} : \mathsf{Prop}[p].\mathsf{act} = \text{act-}*)$ **then**
2 : $\mathsf{mem} \leftarrow \mathsf{Node}[c_0].\mathsf{mem}$
3 : mem -$\leftarrow (\mathsf{id}_c, *); \mathsf{mem}$ +$\leftarrow (\mathsf{id}_c, \mathsf{svk}_c)$
4 : $L \leftarrow \{ \mathsf{id}_c \}$ // set of updated parties
5 : **foreach** $p \in \vec{p}_{\text{'upd'}}$ **do**
6 : $(\mathsf{id}_s, \text{'upd'-svk}) \leftarrow (\mathsf{Prop}[p].\mathsf{orig}, \mathsf{Prop}[p].\mathsf{act})$
7 : **if** $\neg((\mathsf{id}_s, *) \in \mathsf{mem} \wedge \mathsf{id}_s \notin L)$ **then**
8 : **return** $(\bot, \bot)$
9 : mem -$\leftarrow (\mathsf{id}_s, *); \mathsf{mem}$ +$\leftarrow (\mathsf{id}_s, \mathsf{svk})$
10 : L +$\leftarrow \mathsf{id}_s$
11 : **foreach** $p \in \vec{p}_{\text{'rem'}}$ **do**
12 : $(\mathsf{id}_s, \text{'rem'-id}_t) \leftarrow (\mathsf{Prop}[p].\mathsf{orig}, \mathsf{Prop}[p].\mathsf{act})$
13 : **if** $\neg((\mathsf{id}_s, *) \in \mathsf{mem} \wedge (\mathsf{id}_t \in \mathsf{mem} \wedge \mathsf{id}_t \notin L))$ **then**
14 : **return** $(\bot, \bot)$
15 : mem -$\leftarrow (\mathsf{id}_t, *)$
16 : **foreach** $p \in \vec{p}_{\text{'add'}}$ **do**
17 : $(\mathsf{id}_s, \text{'add'-id}_t\text{-svk}_t) \leftarrow (\mathsf{Prop}[p].\mathsf{orig}, \mathsf{Prop}[p].\mathsf{act})$
18 : **if** $\neg((\mathsf{id}_s, *) \in \mathsf{mem} \wedge (\mathsf{id}_t, *) \notin \mathsf{mem})$ **then**
19 : **return** $(\bot, \bot)$
20 : mem +$\leftarrow (\mathsf{id}_t, \mathsf{svk}_t)$
21 : $P \leftarrow ((\mathsf{Prop}[p].\mathsf{orig}, \mathsf{Prop}[p].\mathsf{act}) : p \in \vec{p})$
22 : **return** (mem, P)
23 : **else**
24 : **return** $(\bot, \bot)$

FIGURE 4.14: The helper functions for $\mathcal{F}_{\text{CGKA}}$: Determining the group state after applying a commit.

`auth-invariant`

return `true` **iff**
(a) $\forall c_0$ with $c_p := \mathsf{Node}[c_0].\mathsf{par}, c_p \neq \bot$ and $\mathsf{id} := \mathsf{Node}[c_0].\mathsf{orig}$,
if $\mathsf{Node}[c_0].\mathsf{stat} =$ 'adv' **then sig-inj-allowed**$(c_p, \mathsf{id}) \wedge$ **mac-inj-allowed**(c_p) and
(b) $\forall p$ with $c_p := \mathsf{Prop}[p].\mathsf{par}$ and $\mathsf{id} := \mathsf{Prop}[p].\mathsf{orig}$,
if $\mathsf{Prop}[p].\mathsf{stat} =$ 'adv' **then sig-inj-allowed**$(c_p, \mathsf{id}) \wedge$ **mac-inj-allowed**(c_p) and
(c) $\forall \mathsf{root}_{rt} \neq \bot$ with $\mathsf{id} := \mathsf{Node}[\mathsf{root}_{rt}].\mathsf{orig}$, **sig-inj-allowed**$(\mathsf{root}_{rt}, \mathsf{id})$

`cons-invariant`

return `true` **iff**
(a) $\forall c_0$ **s.t.** $\mathsf{Node}[c_0].\mathsf{par} \neq \bot : (\mathsf{Node}[c_0].\mathsf{prop} \neq \bot \wedge \forall p \in \mathsf{Node}[c_0].\mathsf{prop} : \mathsf{Prop}[p].\mathsf{par} = \mathsf{Node}[c_0].\mathsf{par})$ and
(b) $\forall \mathsf{id}$ **s.t.** $\mathsf{Ptr}[\mathsf{id}] \neq \bot : \mathsf{id} \in \mathsf{Node}[\mathsf{Ptr}[\mathsf{id}]].\mathsf{mem}$ and
(c) the history graph contains no cycle

FIGURE 4.15: The history graph invariants for $\mathcal{F}_{\text{CGKA}}$. We explicitly add the authentication invariant concerning welcome messages which are highlighted in gray.

4.4 Proposed Protocol: Chained CmPKE

This section presents the proposed protocol. At a conceptual level, there are two core differences with TreeKEM:

1. Instead of being arranged as the leaves of a (binary) tree, group members are arranged in a set. This is similar to Chained mKEM [BBN19a]. Alternatively, it can be interpreted as TreeKEM using a tree of arity N and depth 1.

2. Instead of being a passive bulletin board, the delivery service may *edit* a commit message uploaded by a member before forwarding it to any of the $(N-1)$ other group members.

The impact of the first change on *uploading* commit messages is illustrated in Figure 4.1. A member may initiate a new epoch t by encrypting a commit secret $\mathsf{comSecret}^{(t)}$ directly to the $(N-1)$ encryption keys of the other group members using a CmPKE. There is no tree structure anymore and, as an immediate consequence, removing a user no longer leads to "blanking" a node.

The second change is implemented via the use of a CmPKE. Instead of signing the whole CmPKE ciphertext $(\mathsf{T}, \vec{\mathsf{ct}} = (\mathsf{ct}_i)_{i\in[N]})$ embedded in a commit message, the uploader of the message only signs T. The delivery service is expected to forward $(\mathsf{T}, \mathsf{ct}_i)$ to the recipient i. Any tampering on T by the server can be detected by a recipient by checking the signature, and any tampering on ct_i can be detected during the CmPKE decryption procedure. In particular, it achieves the same level of security as provided by TreeKEM.

4.4.1 High-level overview of Chained CmPKE

We first provide a high-level description of the proposed protocol in Figure 4.16. We reuse most of the terminology and function names used by [Alw+20b; AJM22] and highlight the major algorithmic changes below. A complete description is given in Section 4.4.2.

Low-Level Primitives. The main changes relate to two classes of low-level primitives.

The first class captures procedures related to (left-balanced binary) trees: simple ones such as computing the parent or children of a node, determining whether it is the root, an internal node, or a leaf, etc., or more complex ones such as computing its path, co-path, or resolution. A list of 27 such procedures is given in [AJM22, Tab. 1 and 3]. Removing binary trees trivializes or removes these procedures.

The second class relates to public-key encryption. As we replace a standard PKE with a CmPKE, the main effects are that the encryption procedure now takes as input a list of encryption keys $(\mathsf{ek}_i)_i$ instead of a single key, and the presence of a commitment T as an additional output (resp. input) of the encryption (resp. decryption) procedure.

Ripple Effects on Mid-Level Procedures. More notions and procedures related to trees are heavily simplified. For example, treeHash becomes memberHash, and its computation now entails hashing a set in lexicographical order, instead of a binary tree (`*set-tree-hash` becomes `*set-member-hash`). As there is no longer an internal node to authenticate, parentHash and its computation (`*set-parent-hash` and `*parent-hash`) are no longer necessary.

Impact at the Top Level. Since the group is no longer arranged in a binary tree structure but in a set, each user now possess a single encryption keypair instead of $\lceil \log N \rceil$. This simplifies top level procedures (`Commit`, `Process`, `Join`), which refresh these keypairs.

In TreeKEM, commit messages may contain encryptions of *path secrets* (to the resolution of the sibling of each concerned node, via `*rekey-path`) or a path secret on the least common ancestor node of the sender

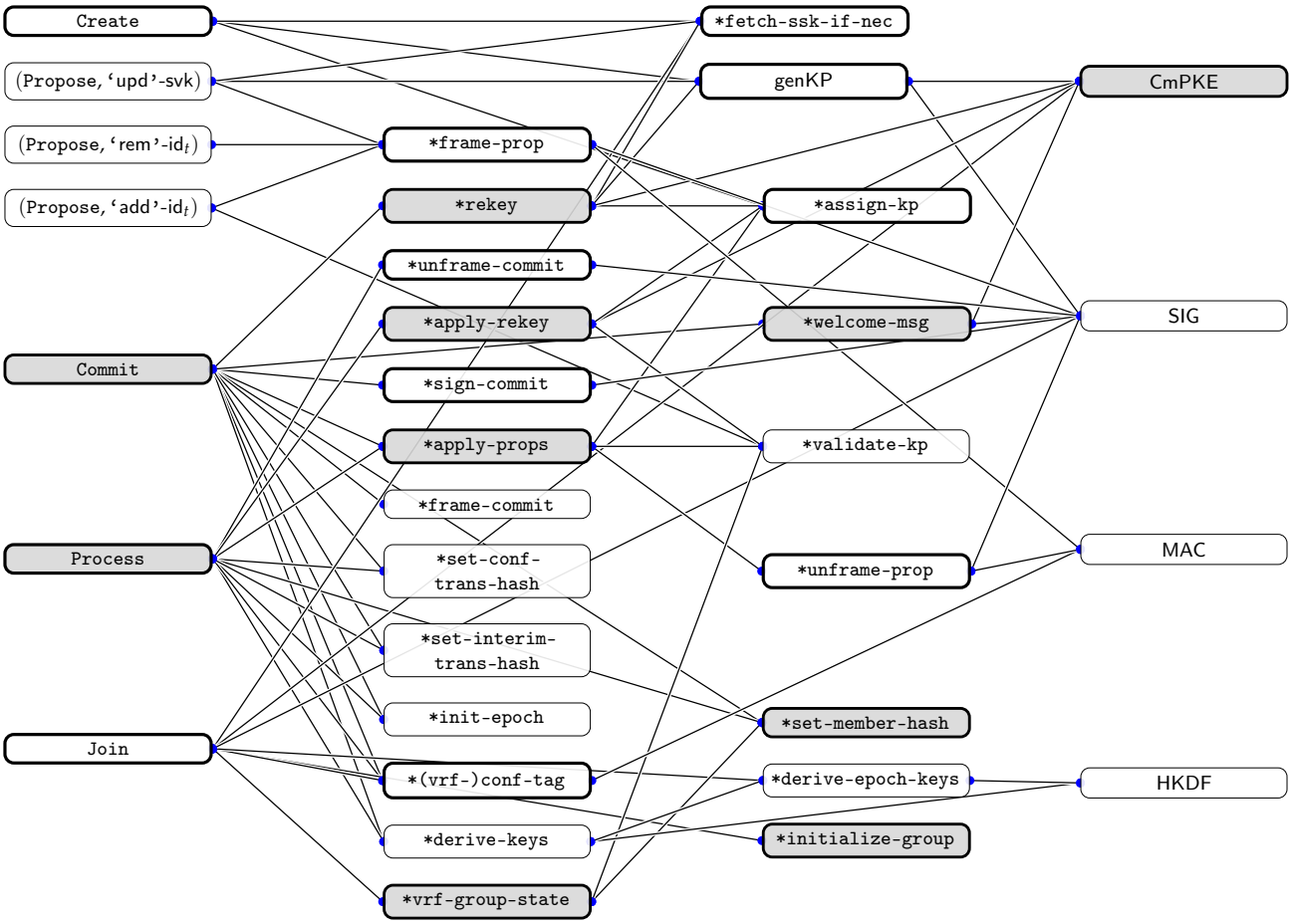


FIGURE 4.16: Call graph of Chained CmPKE. We use the notations `func`, **`func`** and **`func`** to denote functions that undergo respectively minimal, moderate and strong changes compared to [Alw+20b; AJM22].

and each new group member (a common *joiner secret* is also sent to new group members, via `*welcome-msg`). Encryption of path secrets produces $\Omega(\log N)$ ciphertexts, see Figure 4.1 and Footnote 2.

In Chained CmPKE, there is no path secret; instead, a common comSecret is encrypted to all recipients via a single call to CmEnc, producing one multi-recipient ciphertext $(\mathsf{T}, \vec{\mathsf{ct}} = (\widehat{\mathsf{ct}}_{\mathsf{id}'})_{\mathsf{id}' \in \mathsf{receivers}})$, see Figure 4.1. Similarly, a common joinerSecret may be encrypted to newly added members. In each case, the sender of the commit message signs data that includes T, but not $\vec{\mathsf{ct}}$.

As input to `Process` and `Join`, receivers of a commit message will not receive the full package. Precisely, instead of including a full CmPKE ciphertext $(\mathsf{T}, \vec{\mathsf{ct}} = (\widehat{\mathsf{ct}}_{\mathsf{id}'})_{\mathsf{id}' \in \mathsf{receivers}})$, the recipient id only downloads $(\mathsf{T}, \widehat{\mathsf{ct}}_{\mathsf{id}})$ from the server. We call this *selective* (or designated) downloading as the recipient only needs to download a part of the commit message it requires. Since the data signed by the sender includes T but not $\vec{\mathsf{ct}}$, each recipient can verify the signature. Intuitively, the commitment-binding property (Definition 4.2.6) then guarantees the authenticity of $\widehat{\mathsf{ct}}_{\mathsf{id}}$ despite it not being directly signed.

Asymptotic Bandwidth Efficiency.

We now discuss the bandwidth efficiency of the proposed protocols. We leave out elements that reflect logical group operations (e.g., a bitstring encoding "id *has been added to G*") or symmetric key cryptography (e.g., hashes or MAC tags), as they add negligible overheads (compared to public key cryptography) to all solutions.

The bottleneck of both TreeKEM and our solution resides in commit messages, as these are processed on a daily basis (as the output of `Commit`, and the input of `Process`) and contain a significant amount of public key material. We recall that we note ek an encryption key, ct_0 the (ek-independent) part of an mPKE ciphertext, $\widehat{\mathsf{ct}}_i$ the part of a ciphertext dependent of ek_i and sig a signature, and note $|x|$ the bytesize of x. We consider a group of N members, in a epoch with no new member.

TreeKEM**.** The size of an uploaded commit message is dominated by $2 \cdot |\mathsf{sig}| + \lceil \log N \rceil \cdot |\mathsf{ek}| + \Omega(\log N) \cdot (|\mathsf{ct}_0| + |\widehat{\mathsf{ct}}_i|)$.[9] Since all ciphertexts in the commit message are signed jointly by a single signature, recipients need to download all ciphertexts to verify the signature.

Chained CmPKE**.** The size of an uploaded commit message is dominated by $2 \cdot |\mathsf{sig}| + |\mathsf{ek}| + |\mathsf{ct}_0| + N \cdot |\widehat{\mathsf{ct}}_i| + 2\kappa$. The term 2κ stems from our construction of a CmPKE instead of a mPKE (Theorem 4.2.7). This is no larger than a hash digest, and we henceforth ignore it. Since each user performs a selective downloading, the size of a *downloaded* commit message is reduced by a factor $O(N)$, as it is now dominated by $2 \cdot |\mathsf{sig}| + |\mathsf{ek}| + |\mathsf{ct}_0| + |\widehat{\mathsf{ct}}_i|$.

New Members. In both TreeKEM and the proposed protocol, newly added members use the `Join` procedure to process *welcome messages*. These contain all encryption keys $\mathsf{ek}_{\mathsf{id}}$: N in our case (included in memberPublicInfo), and at most $(2N - 1)$ in TreeKEM due to the use of a binary tree. In both cases, the size of a welcome message is dominated by these keys and is $O(N)$. Overall, it seems unlikely that joining a group will be a bandwidth bottleneck, as each member of a group typically performs this operation once, whereas the number of commit messages may be unbounded.

We note that welcome messages encrypt-then-sign a common joinerSecret to the (public) encryption keys of all new members. If an epoch contains k new members, this entails an overhead $|\mathsf{sig}| + k \cdot (|\mathsf{ct}_0| + |\widehat{\mathsf{ct}}_i|)$ for TreeKEM. In the proposed protocol, this is done via a CmPKE, which entails a smaller overhead $|\mathsf{sig}| + |\mathsf{ct}_0| + k \cdot |\widehat{\mathsf{ct}}_i|$.

[9] In both TreeKEM and Chained CmPKE, a commit message contains *two* signatures: one authenticates ciphertexts, and one signs the committer's new encryption key(s) ("*tree signing*" in [AJM22]). A commit message may contain an optional welcome message, which is then signed by a third signature. Our improvements target the first signature (ciphertexts), and are orthogonal to the other two.

Two Alternative Protocols. We briefly present two protocols that also achieve a bandwidth complexity $O(N)$ and $O(1)$ for uploading and downloading commit messages, using only generic primitives.

The first protocol, that we refer to as a *Parallel* KEM, encrypts the same comSecret to all group members using $(N-1)$ parallel (non-committing, single-recipient) PKEs. A distinct signature $\mathsf{sig}_{\mathsf{id}}$ is computed for each distinct $\mathsf{ct}_{\mathsf{id}}$. The cost of an upload is $|\mathsf{ek}| + N(|\mathsf{ct}| + 2 \cdot |\mathsf{sig}|) = O(N)$ and, since each ciphertext is individually authenticated, the cost of a download is $|\mathsf{ek}| + |\mathsf{ct}| + 2 \cdot |\mathsf{sig}| = O(1)$. See P. KEMs in Table 4.1.

Since any PKE is also a decomposable mPKE for $\mathsf{ct}_0 = \perp$, a slightly more involved solution is to build a CmPKE from any single-recipient PKE as a special case of Theorem 4.2.7. Once we have a CmPKE, the construction, which we refer to as *Committing* PKE*s*, is identical to ours. The cost of an upload is now $|\mathsf{ek}| + N \cdot |\mathsf{ct}| + 2 \cdot |\mathsf{sig}| = O(N)$, and the cost of a download remains $|\mathsf{ek}| + |\mathsf{ct}| + 2 \cdot |\mathsf{sig}| = O(1)$, see C. PKE in Table 4.1.

Applying Our Techniques to TreeKEM**.** We can apply to the TreeKEM protocol the two techniques leveraged here: selective downloading and mPKEs.

Thanks to the tree-based structure of TreeKEM, each user can perform selective downloading to retrieve only one ciphertext per commit message. Indeed a similar idea to selective downloading was proposed for TreeKEM [Bar18], but to the best of our knowledge it has never been implemented or formally analyzed. One possible reason for this is because unlike in Chained CmPKE, TreeKEM has the added complexity of maintaining the public keys associated to the internal nodes of the tree. Specifically, a user only needs to know the public keys associated to the internal nodes along its path to the root in order to process commit message, however, it may need to know more if it wants to upload commit messages. Notice the nodes that the user needs to know is not fixed in advance since add/remove/update proposals may adaptively change the topology of the tree. Consequently, a user may need to download additional key materials when performing a commit (which we call *on-the-fly* downloading). Hence, although we believe it is possible to further lower the download cost for TreeKEM using similar ideas, this would entail more server-side bookkeeping of the tree structure and the associating public keys for each internal nodes, which would likely add complexity to the protocol description and security proof. We leave it as an interesting future research to assess the full benefit of such an approach.

Combining TreeKEM with mPKEs/mKEMs was done in [Kat+20], which considered a variant of TreeKEM with trees of arity m. This reduces the number of encryption keys per commit message to $\lceil \log_m N \rceil$ in the best case (unblanked tree), which is still $\Omega(\log N)$ for any constant value of m. Note that setting $m = N$ results in a flat tree, which yields a protocol similar to Chained CmPKE. So while it is possible to apply our techniques to TreeKEM, we found that doing so with the goal of minimizing the total bandwidth cost leads to a protocol very similar to ours, which a posteriori validates our design choices.

Why Efficient mPKE**s Matter.** It may not be obvious that our solution represents an improvement upon Parallel KEMs and Committing PKEs, since all three achieve the same asymptotic bandwidth efficiency: $O(N)$ in upload, $O(1)$ in download. However, a perk of post-quantum cryptography is its ability to provide mPKEs for which the ek_i-dependent part $\widehat{\mathsf{ct}}_i$ of ciphertexts are extremely compact, as illustrated in Table 4.8. The proposed protocol directly benefits from this fact, since the size of uploaded commit messages is $\sim |\widehat{\mathsf{ct}}_i| \cdot N$. In Section 4.5, we propose lattice-based mPKEs inspired by the (possibly alternative) finalists to standardization by NIST Kyber [Sch+20], NTRU LPRime [Ber+20] and FrodoKEM [Nae+20]. Our mPKEs make $\widehat{\mathsf{ct}}_i$ as small as $\{48, 32, 24\}$ bytes. Concretely, this allows the proposed protocol to reduce the *upload* bandwidth cost by two to three orders of magnitude compared to Parallel KEMs and Committing PKEs.

4.4.2 Description of Chained CmPKE

In this section, we provide a more in-depth exposition of our Chained CmPKE protocol.

TABLE 4.1: Bandwidth cost of a commit message to a group of N members (with no newly added member) in terms of public key cryptography. For schemes that use single-recipient PKEs/KEMs, we assume $|\mathsf{ct}| = |\mathsf{ct}_0| + |\mathsf{ct}_i|$. All logarithms are in base 2. The notation $\lceil \log N \rceil$ expresses that for the row labelled [AJM22] the best-case complexity is $\lceil \log N \rceil$, and the worst-case is N.

Scheme	Upload				Download (per recipient)				Total (1 upload, then $(N-1)$ downloads)			
	$\lvert\mathsf{ek}\rvert$	$\lvert\mathsf{ct}_0\rvert$	$\lvert\widehat{\mathsf{ct}}_i\rvert$	$\lvert\mathsf{sig}\rvert$	$\lvert\mathsf{ek}\rvert$	$\lvert\mathsf{ct}_0\rvert$	$\lvert\widehat{\mathsf{ct}}_i\rvert$	$\lvert\mathsf{sig}\rvert$	$\lvert\mathsf{ek}\rvert$	$\lvert\mathsf{ct}_0\rvert$	$\lvert\widehat{\mathsf{ct}}_i\rvert$	$\lvert\mathsf{sig}\rvert$
[AJM22]	$\lceil \log N \rceil$	$\lceil \log N \rceil$	$\lceil \log N \rceil$	2	$\lceil \log N \rceil$	$\lceil \log N \rceil$	$\lceil \log N \rceil$	2	$N\lceil \log N \rceil$	$N\lceil \log N \rceil$	$N\lceil \log N \rceil$	$2N$
Ours	1	1	$(N-1)$	2	1	1	1	2	N	N	$2(N-1)$	$2N$
P. KEMs	1	$(N-1)$	$(N-1)$	N	1	1	1	2	N	$2(N-1)$	$2(N-1)$	$3N-2$
C. PKEs	1	$(N-1)$	$(N-1)$	2	1	1	1	2	N	$2(N-1)$	$2(N-1)$	$2N$

As already explained in Section 4.4, unlike TreeKEM, we no longer require to maintain a tree structure since the structure we maintain is a depth-1 tree (which is much like a comb). This makes the description of the proposed protocol much simpler relative to TreeKEM and relieves us from "blanking" nodes when updating and removing users from the group. Effectively, the security analysis is also simpler since we no longer need to keep track of the exposed/unexposed secrets assigned to the internal nodes of the tree.

Moreover, during a commit protocol, the committer does not sign the whole ciphertext but only the part that binds the message, i.e., the commitment T in CmPKE. The delivery server is expected to parse the uploaded commit message and forward the relevant parts to the receivers.

Below we describe our Chained CmPKE protocol and provide details on the differences between TreeKEM version 10 of MLS formalized by [AJM22].

Protocol States. Each user holds a group state G. It consists of the variables listed in Table 4.2. The G.member array stores the information of the group members. The index of G.member is specified by the party identities and each entry consists of the variables listed in Table 4.4. The member hash G.memberHash is the hash of all key packages stored in G.member.

The group state contains three hashes: *confirmation transcript hash* (confTransHash), *confirmation transcript hash without committer identifier* (confTransHash-w.o-'id_c') and *interim transcript hash* (interimTransHash). Roughly, these hashes maintain the consistency between the previous and current epoch and are used to enforce a consistent view within the group members.

If a group member issues an update proposal or commit message that did not get confirmed by the server, the corresponding secrets are stored in G.pendUpd and G.pendCom, respectively. When a member receives a message which has been created by itself, it retrieves the corresponding secrets from G.pendUpd or G.pendCom (rather than processing it from scratch).

For readability, we define the useful helper methods corresponding to the group state, listed in Table 4.3. In the security proof, G additionally stores the variables listed in Table 4.5

Differences from TreeKEM. All variables except for G.member, G.memberHash and G.confTransHash-w.o-'id_c' are defined identically to TreeKEM. G.member corresponds to the left-balanced binary tree τ considered in [AJM22], restricted to arity N and depth 1. Namely, G.member only maintains a simply array rather than a tree. G.memberHash is a replacement of treeHash in TreeKEM. We newly define the hash value G.confTransHash-w.o-'id_c', which is used in the join protocol to confirm the sender of the welcome message.

TABLE 4.2: The protocol state of Chained CmPKE.

G.gid	The identifier of the group.
G.epoch	The current epoch number.
G.confTransHash	The confirmed transcript hash.
G.confTransHash-w.o-'$\mathsf{id_c}$'	The confirmed transcript hash without the committer identity.
G.interimTransHash	The interim transcript hash for the next epoch.
G.member$[*]$	A mapping associating party id with its state.
G.memberHash	A hash of the public part of G.member$[*]$.
G.certSvks$[*]$	A mapping associating the set of validated signature verification keys to each party.
G.pendUpd$[*]$	A mapping associating the secret keys for each pending update proposal issued by id.
G.pendCom$[*]$	A mapping associating the new group state for each pending commit issued by id.
G.id	The identity of the party.
G.ssk	The current signing key.
G.appSecret	The current epoch's shared key.
G.membKey	The key used to MAC proposal packages.
G.initSecret	The next epoch's init secret.

TABLE 4.3: The helper methods on the protocol state.

G.clone()	Returns (independent) copy of G.
G.memberIDs()	Returns the list of party ids sorted by dictionary order.
G.memberIDsvks()	Returns the list of party ids and its associating svk sorted by dictionary order in the ids.
G.memberPublicInfo()	Returns the public part of G.member$[*]$.
G.groupCont()	Returns (G.gid, G.epoch, G.memberHash, G.confTransHash).

TABLE 4.4: The party id's state stored in G.member[id] and helper method.

id	The identity of the party.
ek	The encryption key of a CmPKE scheme.
dk	The corresponding decryption key.
svk	The signature verification key of a signature scheme.
sig	The signature for (id, ek, svk) under the signature singing key corresponding to svk.
kp()	Returns (id, ek, svk, sig) (if G.member[id] $\neq \perp$).

TABLE 4.5: The protocol state maintained only during the security proof.

G.joinerSecret	The current epoch's joiner secret.
G.comSecret	The current epoch's commit secret.
G.confKey	The key used to MAC for commit and welcome messages.
G.confTag	The MAC tag included either in the commit or welcome message.
G.membTags	The set of MAC tags included in the proposal messages.

Protocol Algorithms. The main protocol is depicted in Figures 4.18 to 4.20. The associated helper functions are depicted in Figures 4.21 to 4.24. His a hash function (modeled as a random oracle). In these figures, the differences from TreeKEM version 10 in MLS considered by [AJM22] are highlighted in gray.

(0) Key Generation. The algorithms `get-ssk` and `get-kp` depicted in Figure 4.17 are used by the Authentication Service and Key Service functionalities $\mathcal{F}_{\text{AS}}$ Figure 4.7 and $\mathcal{F}_{\text{KS}}$ Figure 4.8, respectively.

The algorithm `get-ssk` generates a fresh signature key pair. The algorithm `get-kp` generates a fresh CmPKE key pair and outputs the decryption key and the *key package*. The key package kp contains the party's identity id, the CmPKE encryption key ek, and the verification key svk along id's signature for them.

Differences from TreeKEM. These key generation algorithms are defined identically to TreeKEM.

(1) Group Creation. The group is created (by the designated party $\mathsf{id}_{\mathsf{creator}}$ in our model) using the input $(\texttt{Create}, \mathsf{svk})$. This input initializes the group state and creates a new group with the single member $\mathsf{id}_{\mathsf{creator}}$. The group creator fetches the corresponding signing key ssk from $\mathcal{F}_{\text{AS}}$ using the helper function `*fetch-ssk-if-nec`.

Differences from TreeKEM. The group creation protocol is defined identically to TreeKEM except that party $\mathsf{id}_{\mathsf{creator}}$ maintains a simpler group protocol state G compared to TreeKEM. Note that, unlike TreeKEM, the proposed protocol initializes a random joiner secret and derives the epoch secrets from it. Then, it computes the confirmation tag confTag for the initial group. This is because confTag is necessary to discuss the security of the protocol.

(2) Proposals. The protocol first prepares a preliminary proposal message P.

- To create an update proposal, the protocol generates a fresh key package together with the corresponding decryption key dk. The key package kp is included in the proposal and dk is stored in G.pendUpd. When a new verification key svk is used, the protocol fetches the corresponding signing key ssk from $\mathcal{F}_{\text{AS}}$. (ssk is also stored in G.pendUpd.)
- To create an add proposal, the protocol fetches the key package for the added party from $\mathcal{F}_{\text{KS}}$. The proposal consists of the key package which includes the added party's identity.
- The remove proposal consists of the identity of a removed party.

All proposals are framed using `*frame-prop`. It first signs the proposal P together with the string 'proposal', the group context including confTransHash, and the sender's identity. This signature prevents impersonation by another group member. In addition, to ensure the PCS security and group membership of the sender, everything including the signature is MACed using the membership key. The MAC tag ties the proposal to a specific group/epoch since the signature key may be shared across groups and is long-lived. In summary, to inject or modify messages, the adversary must corrupt both the sender's signing key and the current epoch secrets. The actual proposal message p consists of everything except the G.memberHash and G.confTransHash since the other components can be retrieved from the protocol state of the recipients.

Differences from TreeKEM. The proposed protocol is defined identically to TreeKEM.

(3) Commits. To create a new commit message, a party id runs the protocol on input $(\texttt{Commit}, \vec{p}, \mathsf{svk})$. The protocol first initializes the next epoch's group state by copying the current one. It then applies the proposals $\vec{p}$ using `*apply-props`. It verifies the validity of the MAC tag and signature in each proposal. The protocol then derives id's new CmPKE key pair and a new *commit secret* using the helper function `*rekey`. It outputs a fresh commit secret, a fresh key package kp for the committer, and a CmPKE ciphertext $(\mathsf{T}, \vec{\mathsf{ct}})$ encrypting the commit secret. Note that the commit secret will be shared among existing users who are not removed in the next epoch.

The commit message consists of two parts: a party-independent message c_0 and a party-dependent message $\widehat{c}$. The protocol first prepares a preliminary commit message C_0 including the list of the hash of all the applied proposals propIDs, the key package kp, and the commitment T. This commit message is signed alongside the group context using `*sign-commit`. Afterward, the protocol derives the epoch secrets using `*derive-keys` and computes the confirmation tag (see `*gen-conf-tag`). c_0 is constructed from C_0, the signature, and the confirmation tag. Then, the protocol prepares the party-dependent message $\widehat{c}$. It is set as $(\mathsf{id}, \widehat{\mathsf{ct}}_{\mathsf{id}})$, or $(\mathsf{id}, \perp)$ if the party id is removed in the next epoch. (Here, $\vec{c}$ is the list of $\widehat{c}$.)

If new members are added, the protocol creates a welcome message using the function `*welcome-msg`. The welcome message also consists of two parts: a party independent message w_0 and a party dependent message $\widehat{w}$. It first encrypts the joiner secret (which will be used to derive epoch secrets) with the added members' encryption keys, and obtains a CmPKE ciphertext $(\mathsf{T}, \vec{\mathsf{ct}} = (\widehat{\mathsf{ct}}_{\mathsf{id}_t})_{\mathsf{id}_t \in \mathsf{addedMem}})$. Then the protocol composes a group information groupInfo which contains the public part of the group state, the confirmation tag, and the sender's identity. groupInfo and T are signed by the sender's signing key and w_0 is set as $(\mathsf{groupInfo}, \mathsf{T}, \mathsf{sig})$. Then, the protocol prepares the party-dependent message $\widehat{w}$. It is set as $(\mathsf{id}, \mathsf{kphash}, \widehat{\mathsf{ct}}_{\mathsf{id}})$ where kphash is the hash of the used key package. (Here, $\vec{w}$ is the list of $\widehat{w}$.)

Finally, the protocol computes the interim transcript hash for the next epoch by hashing the current confirmation hash and the newly generated confirmation tag. The next epoch's state is stored in G.pendCom.

Differences from TreeKEM. The following summarizes the differences between Chained CmPKE and TreeKEM.

1. Our `*apply-props` simply rewrites entries in G.member: if id is deleted, it sets G.member[id] to $\perp$; if id is added, it stores its key package in a new entry; if id is updated, it replaces the old key package with the new one. In contrast, TreeKEM additionally runs the 'blank node' operation after updating the leaf nodes. That is, the committer blanks the nodes on the path from the updated or removed leaf to the root.

2. Our `*rekey` operation simply encrypts a new comSecret with the recipients' CmPKE encryption keys. In contrast, TreeKEM runs a 'path update' operation to derive comSecret. It refreshes all PKE keys along the path from the committer's leaf to the root. Each path secret is then encrypted to the resolution of the sibling of the concerned node. Here, the secret on the root is used as comSecret.

3. Chained CmPKE signs only T, rather than T *and* $(\widehat{\mathsf{ct}}_{\mathsf{id}})_{\mathsf{id} \in \mathsf{receivers}}$. This allows the delivery server to send only the message needed for each user, and effectively lowers the downloaded package size from $O(N)$ to $O(1)$. In contrast, in TreeKEM, all the ciphertexts (each encrypting a path secret) are signed. The size of the downloaded package is therefore $O(\log N)$ in the best case (i.e., full non-blanked tree) and $O(N)$ in the worst case (i.e., heavily blanked tree).

4. Our commit message consists of two parts: c_0 is a party-independent message and will be sent to all the recipients. $\widehat{c}_{\mathsf{id}}$ is a party-dependent message that contains the identity of a single recipient id and the ciphertext corresponding $\widehat{\mathsf{ct}}_{\mathsf{id}}$. This is only sent to the specific party id. In contrast, in TreeKEM, a commit message is viewed as a monolithic bloc and the commit message is sent to all the recipients without any modification. This corresponds to setting $c_0 = \perp$ and $\widehat{\mathsf{ct}}_{\mathsf{id}} = \widehat{\mathsf{ct}}_{\mathsf{id}'}$ for all $\mathsf{id}, \mathsf{id}' \in$ receivers in our new ideal functionality.

5. Our welcome message only encrypts a new joinerSecret with the added members' CmPKE encryption keys. There is no need to send the secrets assigned to the internal nodes of a tree as in TreeKEM. Analogous to the commit message, the welcome message also consists of two parts.

The other process (e.g., generating hash values, re-keying) are identical.

(4) Process. Consider the input $(\texttt{Process}, c_0, \widehat{c}, \vec{p})$. If the party id is the creator of the received commit message c_0, then the protocol simply retrieves the new epoch state from $G.\mathsf{pendCom}$; otherwise, it proceeds as follows.

First, the protocol unframes the message, i.e., verifies the signature and checks that it belongs to the correct group and epoch (cf. `*unframe-commit` in Figure 4.24). Next, it verifies whether $\vec{p}$ matches the committed proposals in c_0. If so, it applies them using `*apply-props`.

If id is not removed, the protocol derives a new epoch secret. It decrypts the ciphertext using `*apply-rekey` (it also applies the committer's new key package) and computes the epoch secret using `*derive-keys`. Finally, it verifies the confirmation tag in c_0 and derives a new interim transcript hash.

Differences from TreeKEM. There are two differences. First is the input message. Chained CmPKE allows the server to sanitize commit messages by delivering to each group member the strict amount of data they need. Namely, the server only sends $(c_0, \widehat{c}_{\mathsf{id}})$ to the party id, and hence, party id only receives the ciphertext it needs to update its protocol state. This reduces the party's download cost and the server's bandwidth.

Second is the `*apply-rekey` function. To obtain comSecret, Chained CmPKE simply decrypts the ciphertext. In contrast, TreeKEM decrypts the ciphertext, which contains the secret on the least common ancestor of the committer and the recipient, and then runs the 'path update' operation to recover comSecret (i.e., root secret).

(5) Join. Upon receiving an input $(w_0, \widehat{w})$, the protocol initializes a new group state and copies the public group information from w_0. Then it checks the validity of the confirmation hash and interim transcript hash by recomputing these hashes from the received information. It also verifies the signature and the validity of the member list and each group member's key package. If the information is valid, the protocol decrypts the joiner secret. To this end, it fetches all its key package and decryption key pairs from $\mathcal{F}_{\mathrm{KS}}$ and determines the one that has been used for the welcome message by checking the hash of the key package.

Finally, it derives the epoch secrets from the joiner secret and verifies the confirmation tag.

Differences from TreeKEM. As for the commit message, new member receives the sanitized message $(w_0, \widehat{w}_{\mathsf{id}})$. Chained CmPKE simply decrypts the ciphertext and derives the epoch secret from the decrypted joinerSecret. In contrast, in TreeKEM, the welcome message contains the secret of the least common ancestor of the committer and the recipient. The receiver then runs the 'path update' operation in order to derive the decryption keys of its parents. This process does not appear in Chained CmPKE.

Chained CmPKE checks the validity of the confirmation hash in the welcome message by using confTransHash-w.o-'id_c' and id_c. This allows the recipient of the welcome message to verify that id_c has computed the confirmation hash.

(6) Key. Upon input $(\texttt{Key})$, the protocol outputs the application secret of the current epoch and deletes it from the local state.

Differences from TreeKEM. This key protocol is the same as TreeKEM.

genSSK()

1 : $(\mathsf{svk}, \mathsf{ssk}) \leftarrow \mathsf{SIG.KeyGen}(\mathsf{pp}_{\mathsf{SIG}})$
2 : **return** $(\mathsf{svk}, \mathsf{ssk})$

genKP(id, svk, ssk)

1 : $s \leftarrow\$ \{0,1\}^{\kappa}$
2 : $(\mathsf{ek}, \mathsf{dk}) \leftarrow \mathsf{CmGen}(\mathsf{pp}_{\mathsf{CmPKE}}; \mathsf{H}(s))$
3 : $\mathsf{sig} \leftarrow \mathsf{SIG.Sign}(\mathsf{pp}_{\mathsf{SIG}}, \mathsf{ssk}, (\mathsf{id}, \mathsf{ek}, \mathsf{svk}))$
4 : $\mathsf{kp} \leftarrow (\mathsf{id}, \mathsf{ek}, \mathsf{svk}, \mathsf{sig})$
5 : **return** $(\mathsf{kp}, \mathsf{dk})$

FIGURE 4.17: Key generation algorithms of Chained CmPKE.

Input (Create, svk)

1 : **req** $G = \bot \wedge \mathsf{id} = \mathsf{id}_{\mathsf{creator}}$
2 : $G.\mathsf{gid} \leftarrow\$ \{0,1\}^{\kappa}; G.\mathsf{joinerSecret} \leftarrow\$ \{0,1\}^{\kappa}$
3 : $G.\mathsf{epoch} \leftarrow 0$
4 : $G.\mathsf{member}[*] \leftarrow \bot; G.\mathsf{memberHash} \leftarrow \bot$
5 : $G.\mathsf{confTransHash\text{-}w.o\text{-}`id_c'} \leftarrow \bot$
6 : $G.\mathsf{confTransHash} \leftarrow \bot$
7 : $G.\mathsf{certSvks}[*] \leftarrow \emptyset$
8 : $G.\mathsf{pendUpd}[*] \leftarrow \bot; G.\mathsf{pendCom}[*] \leftarrow \bot$
9 : $G.\mathsf{id} \leftarrow \mathsf{id}$
10 : **try** $\mathsf{ssk} \leftarrow$ `*fetch-ssk-if-nec`(G, svk)
11 : $(\mathsf{kp}, \mathsf{dk}) \leftarrow \mathsf{genKP}(\mathsf{id}, \mathsf{svk}, \mathsf{ssk})$
12 : $G \leftarrow$ `*assign-kp`$(G, \mathsf{id}, \mathsf{kp})$
13 : $G.\mathsf{ssk} \leftarrow \mathsf{ssk}$
14 : $G.\mathsf{member}[\mathsf{id}].\mathsf{dk} \leftarrow \mathsf{dk}$
15 : $G.\mathsf{memberHash} \leftarrow$ `*derive-member-hash`(G)
16 : $(G, \mathsf{confKey})$
$\quad \leftarrow$ `*derive-epoch-keys`$(G, G.\mathsf{joinerSecret})$
17 : $\mathsf{confTag} \leftarrow$ `*gen-conf-tag`$(G, \mathsf{confKey})$
18 : $G \leftarrow$ `*set-interim-trans-hash`$(G, \mathsf{confTag})$

Input (Propose, 'upd'-svk)

1 : **req** $G \neq \bot$
2 : **try** $\mathsf{ssk} \leftarrow$ `*fetch-ssk-if-nec`(G, svk)
3 : $(\mathsf{kp}, \mathsf{dk}) \leftarrow \mathsf{genKP}(\mathsf{id}, \mathsf{svk}, \mathsf{ssk})$
4 : $P \leftarrow (\text{'upd'}, \mathsf{kp})$
5 : $p \leftarrow$ `*frame-prop`(G, P)
6 : $G.\mathsf{pendUpd}[p] \leftarrow (\mathsf{ssk}, \mathsf{dk})$
7 : **return** p

Input (Propose, 'add'-id_t)

1 : **req** $G \neq \bot \wedge \mathsf{id}_t \notin G.\mathsf{memberIDs}()$
2 : Send $(\texttt{get-kp}, \mathsf{id}_t)$ to $\mathcal{F}_{\mathrm{KS}}$ and receive kp_t
3 : **req** $\mathsf{kp}_t \neq \bot$
4 : **try** $G \leftarrow$ `*validate-kp`$(G, \mathsf{kp}_t, \mathsf{id}_t)$
5 : $P \leftarrow (\text{'add'}, \mathsf{kp}_t)$
6 : $p \leftarrow$ `*frame-prop`(G, P)
7 : **return** p

Input (Propose, 'rem'-id_t)

1 : **req** $G \neq \bot \wedge \mathsf{id}_t \in G.\mathsf{memberIDs}()$
2 : $P \leftarrow (\text{'rem'}, \mathsf{id}_t)$
3 : $p \leftarrow$ `*frame-prop`(G, P)
4 : **return** p

FIGURE 4.18: Main protocol of Chained CmPKE: Create and Propose. The major changes from [AJM22] are highlighted in gray.

Input $(\texttt{Commit}, \vec{p}, \mathsf{svk})$

1 : **req** $G \neq \bot$
2 : $G' \leftarrow$ `*init-epoch`(G)
3 : **try** $(G', upd, rem, add) \leftarrow$ `*apply-props`$(G, G', \vec{p})$
4 : **req** $(*, \text{'rem'-id}) \notin rem \wedge (\mathsf{id}, *) \notin upd$
5 : // Recipients of the welcome message
6 : $\mathsf{addedMem} \leftarrow \{\, \mathsf{id}_t \mid (*, \text{'add'-id}_t\text{-}*) \in add \,\}$
7 : // Recipients of the new commit secret
8 : $\mathsf{receivers} \leftarrow G'.\mathsf{memberIDs}() \setminus \mathsf{addedMem}$
9 : **try** $(G', \mathsf{comSecret}, \mathsf{kp}, \mathsf{T}, \vec{\mathsf{ct}} = (\widehat{\mathsf{ct}}_{\mathsf{id}})_{\mathsf{id} \in \mathsf{receivers}}) \leftarrow$ `*rekey`$(G', \mathsf{receivers}, \mathsf{id}, \mathsf{svk})$
10 : $G' \leftarrow$ `*set-member-hash`(G')
11 : $\mathsf{propIDs} \leftarrow ()$
12 : **foreach** $p \in \vec{p}$ **do** $\mathsf{propIDs} \mathrel{+}\!\!\leftarrow \mathsf{H}(p)$
13 : $C_0 \leftarrow (\mathsf{propIDs}, \mathsf{kp}, \mathsf{T})$
14 : $\mathsf{sig} \leftarrow$ `*sign-commit`(G, C_0)
15 : $G' \leftarrow$ `*set-conf-trans-hash`$(G, G', \mathsf{id}, C_0, \mathsf{sig})$
16 : $(G', \mathsf{confKey}, \mathsf{joinerSecret}) \leftarrow$ `*derive-keys`$(G, G', \mathsf{comSecret})$
17 : $\mathsf{confTag} \leftarrow$ `*gen-conf-tag`$(G', \mathsf{confKey})$
18 : $c_0 \leftarrow$ `*frame-commit`$(G, C_0, \mathsf{sig}, \mathsf{confTag})$
19 : $G' \leftarrow$ `*set-interim-trans-hash`$(G', \mathsf{confTag})$
20 : $\vec{c} \leftarrow \emptyset$
21 : **foreach** $\mathsf{id} \in G.\mathsf{memberIDs}()$ **do**
22 : **if** $\mathsf{id} \in \mathsf{receivers}$ **then** $\vec{c} \mathrel{+}\!\!\leftarrow \widehat{c}_{\mathsf{id}} = (\mathsf{id}, \widehat{\mathsf{ct}}_{\mathsf{id}})$
23 : **else** $\vec{c} \mathrel{+}\!\!\leftarrow \widehat{c}_{\mathsf{id}} = (\mathsf{id}, \bot)$
24 : **if** $add \neq ()$ **then**
25 : $(G', w_0, \vec{w}) \leftarrow$ `*welcome-msg`$(G', \mathsf{addedMem}, \mathsf{joinerSecret}, \mathsf{confTag})$
26 : **else**
27 : $w_0 \leftarrow \bot; \vec{w} \leftarrow \emptyset$
28 : $G.\mathsf{pendCom}[c_0] \leftarrow (G', \vec{p}, upd, rem, add)$
29 : **return** $(c_0, \vec{c}, w_0, \vec{w})$

Input `Key`

1 : **req** $G \neq \bot$
2 : $\mathsf{k} \leftarrow G.\mathsf{appSecret}$
3 : $G.\mathsf{appSecret} \leftarrow \bot$
4 : **return** k

FIGURE 4.19: Main protocol of Chained CmPKE: Commit and Key. The major changes from [AJM22] are highlighted in gray.

Input $(\texttt{Process}, c_0, \widehat{c}, \vec{p})$

1 : **req** $G \neq \bot$
2 : $(\mathsf{id}_c, C_0, \mathsf{sig}, \mathsf{confTag}) \leftarrow$ `*unframe-commit`(G, c_0)
3 : **if** $\mathsf{id}_c = \mathsf{id}$ **then**
4 : **parse** $(G', \vec{p}', upd, rem, add) \leftarrow G.\mathsf{pendCom}[c_0]$
5 : **req** $\vec{p} = \vec{p}'$
6 : **return** $(\mathsf{id}_c, upd \| rem \| add)$
7 : **parse** $(\mathsf{propIDs}, \mathsf{kp}_c, \mathsf{T}) \leftarrow C_0$
8 : **parse** $(\mathsf{id}', \widehat{\mathsf{ct}}_{\mathsf{id}'}) \leftarrow \widehat{c}$
9 : **req** $\mathsf{id} = \mathsf{id}'$
10 : **for** $i \in 1, \ldots, |\vec{p}|$ **do**
11 : **req** $\mathsf{H}(\vec{p}[i]) = \mathsf{propIDs}[i]$
12 : $G' \leftarrow$ `*init-epoch`(G)
13 : **try** $(G', upd, rem, add) \leftarrow$ `*apply-props`$(G, G', \vec{p})$
14 : **req** $(*, \mathsf{id}_c) \notin rem \wedge (\mathsf{id}_c, *) \notin upd$
15 : **if** $(*, \text{'rem'-id}) \in rem$ **then**
16 : $G' \leftarrow \bot$
17 : **else**
18 : $G' \leftarrow$ `*set-conf-trans-hash`$(G, G', \mathsf{id}_c, C_0, \mathsf{sig})$
19 : $(G', \mathsf{comSecret}) \leftarrow$ `*apply-rekey`$(G', \mathsf{id}_c, \mathsf{kp}_c, \mathsf{T}, \widehat{\mathsf{ct}}_{\mathsf{id}})$
20 : $G' \leftarrow$ `*set-member-hash`(G')
21 : $(G', \mathsf{confKey}, \mathsf{joinerSecret}) \leftarrow$ `*derive-keys`$(G, G', \mathsf{comSecret})$
22 : **req** `*vrf-conf-tag`$(G', \mathsf{confKey}, \mathsf{confTag})$
23 : $G' \leftarrow$ `*set-interim-trans-hash`$(G', \mathsf{confTag})$
24 : **return** $(\mathsf{id}_c, upd \| rem \| add)$

Input $(\texttt{Join}, w_0, \widehat{w})$

1 : **req** $G = \bot$
2 : **parse** $(\mathsf{groupInfo}, \mathsf{T}, \mathsf{sig}) \leftarrow w_0$
3 : **parse** $(\mathsf{id}', \mathsf{kphash}, \widehat{\mathsf{ct}}_{\mathsf{id}'}) \leftarrow \widehat{w}$
4 : **req** $\mathsf{id} = \mathsf{id}'$
5 : **try** $(G, \mathsf{confTag}, \mathsf{id}_c) \leftarrow$ `*initialize-group`$(G, \mathsf{id}, \mathsf{groupInfo})$
6 : $\boxed{\mathsf{CTHash} = \mathsf{H}(G.\text{confTransHash-w.o-'}\mathsf{id_c}', \mathsf{id}_c)}$
7 : $\boxed{\mathsf{ITHash} = \mathsf{H}(G.\mathsf{confTransHash}, \mathsf{confTag})}$
8 : $\boxed{\textbf{req } G.\mathsf{confTransHash} = \mathsf{CTHash}}$
9 : $\boxed{\textbf{req } G.\mathsf{interimTransHash} = \mathsf{ITHash}}$
10 : $\mathsf{svk} \leftarrow G.\mathsf{member}[\mathsf{id}_c].\mathsf{svk}$
11 : **req** $\mathsf{SIG.Verify}(\mathsf{svk}, \mathsf{sig}, (\mathsf{groupInfo}, \mathsf{ct}_0))$
12 : **try** $G \leftarrow$ `*vrf-group-state`(G)
13 : $\mathsf{svk} \leftarrow G.\mathsf{member}[\mathsf{id}].\mathsf{svk}$
14 : Send $(\texttt{get-ssk}, \mathsf{svk})$ to $\mathcal{F}_{\mathrm{AS}}$ and receive ssk
15 : $G.\mathsf{ssk} \leftarrow \mathsf{ssk}$
16 : Send `get-dks` to $\mathcal{F}_{\mathrm{KS}}$ and receive kbs
17 : $\mathsf{joinerSecret} \leftarrow \bot$
18 : **foreach** $(\mathsf{kp}, \mathsf{dk}) \in \mathsf{kbs}$ **do**
19 : **if** $\mathsf{H}(\mathsf{kp}) = \mathsf{kphash}$ **then**
20 : **req** $G.\mathsf{member}[\mathsf{id}].\mathsf{kp}() = \mathsf{kp}$
21 : $G.\mathsf{member}[\mathsf{id}].\mathsf{dk} \leftarrow \mathsf{dk}$
22 : $\mathsf{joinerSecret} \leftarrow \mathsf{CmDec}(\mathsf{dk}, \mathsf{T}, \widehat{\mathsf{ct}}_{\mathsf{id}})$
23 : **req** $\mathsf{joinerSecret} \neq \bot$
24 : $(G, \mathsf{confKey}) \leftarrow$ `*derive-epoch-keys`$(G, \mathsf{joinerSecret})$
25 : **req** `*vrf-conf-tag`$(G, \mathsf{confKey}, \mathsf{confTag})$
26 : **return** $(\mathsf{id}_c, G.\mathsf{memberIDsvks}())$

FIGURE 4.20: Main protocol of Chained CmPKE: Process and Join. The major changes from [AJM22] are highlighted in gray. The boxed components are missing from prior works, which we believe are required to satisfy the UC functionality. Please see the proof for more detail.

***init-epoch(G)**

$G' \leftarrow G.\text{clone}()$
$G'.\text{epoch} \leftarrow G.\text{epoch} + 1$
$G'.\text{pendUpd}[*] \leftarrow \bot; G'.\text{pendCom}[*] \leftarrow \bot$
return G'

***rekey$(G', \text{receivers}, \text{id}, \text{svk})$**

try $\text{ssk} \leftarrow$ *fetch-ssk-if-nec(G', svk)
$(\text{kp}, \text{dk}) \leftarrow \text{genKP}(\text{id}, \text{svk}, \text{ssk})$
$G' \leftarrow$ *assign-kp(G', kp)
$G'.\text{ssk} \leftarrow \text{ssk}; G'.\text{member}[\text{id}].\text{dk} \leftarrow \text{dk}$
$\text{comSecret} \leftarrow\!\!\$\ \{0,1\}^\kappa$
// Note that receivers is non-empty
$\vec{\text{ek}} \leftarrow (G.\text{member}[\text{id}'].\text{ek})_{\text{id}' \in \text{receivers}}$
$(\mathsf{T}, \vec{\text{ct}} = (\widehat{\text{ct}}_{\text{id}'})_{\text{id}' \in \text{receivers}}) \leftarrow \text{CmEnc}(\vec{\text{ek}}, \text{comSecret})$
return $(G', \text{comSecret}, \text{kp}, \mathsf{T}, \vec{\text{ct}})$

***apply-rekey$(G', \text{id}_c, \text{kp}_c, \mathsf{T}, \text{ct})$**

$\text{dk} \leftarrow G'.\text{member}[G'.\text{id}].\text{dk}$
$\text{comSecret} \leftarrow \text{CmDec}(\text{dk}, \mathsf{T}, \text{ct})$
try $G' \leftarrow$ *validate-kp$(G', \text{kp}_c, \text{id}_c)$
$G' \leftarrow$ *assign-kp(G', kp_c)
return $(G', \text{comSecret})$

***apply-props$(G, G', \vec{p})$**

$upd, rem, add \leftarrow ()$
foreach $p \in \vec{p}$ **do**
  try $(\text{id}_s, P) \leftarrow$ *unframe-prop(G, p)
  parse $(\text{type}, \text{val}) \leftarrow P$
  if $\text{type} =$ 'upd' **then**
    req $\text{id}_s \in G.\text{memberIDs}()$
    req $(\text{id}_s, *) \notin upd \wedge rem = () \wedge add = ()$
    try $G' \leftarrow$ *validate-kp$(G', \text{val}, \text{id}_s)$
    $G' \leftarrow$ *assign-kp(G', val)
    if $\text{id}_s = G.\text{id}$ **then**
      parse $(\text{ssk}, \text{dk}) \leftarrow G.\text{pendUpd}[p]$
      $G'.\text{ssk} \leftarrow \text{ssk}$
      $G'.\text{member}[G.\text{id}].\text{dk} \leftarrow \text{dk}$
    $\text{svk} \leftarrow G'.\text{member}[\text{id}_s].\text{svk}$
    $upd \mathrel{+}\!\!\leftarrow (\text{id}_s, \text{'upd'-svk})$
  elseif $\text{type} =$ 'rem' **then**
    parse $\text{id}_t \leftarrow \text{val}$
    req $\text{id}_t \neq \text{id}_s \wedge \text{id}_t \in G'.\text{memberIDs}()$
    req $(\text{id}_t, *) \notin upd \wedge add = ()$
    $G'.\text{member}[\text{id}_t] \leftarrow \bot$
    $rem \mathrel{+}\!\!\leftarrow (\text{id}_s, \text{'rem'-id}_t)$
  elseif $\text{type} =$ 'add' **then**
    parse $(\text{id}_t, *, \text{svk}_t, *, *) \leftarrow \text{val}$
    req $\text{id}_t \notin G'.\text{memberIDs}()$
    try $G' \leftarrow$ *validate-kp$(G', \text{val}, \text{id}_t)$
    $G' \leftarrow$ *assign-kp(G', val)
    $add \mathrel{+}\!\!\leftarrow (\text{id}_s, \text{'add'-id}_t\text{-svk}_t)$
  else
    return $\bot$
return (G', upd, rem, add)

FIGURE 4.21: Helper functions of Chained CmPKE: Commit and Process. The major changes from [AJM22] are highlighted in gray.

***welcome-msg**$(G', \text{addedMem}, \text{joinerSecret}, \text{confTag})$

1 : $\vec{\text{ek}} \leftarrow (G'.\text{member}[\text{id}_t].\text{ek})_{\text{id}_t \in \text{addedMem}}$
2 : $(\mathsf{T}, \vec{\text{ct}} = (\widehat{\text{ct}}_{\text{id}_t})_{\text{id}_t \in \text{addedMem}})$
$\leftarrow \text{CmEnc}(\text{pp}_{\text{CmPKE}}, \vec{\text{ek}}, \text{joinerSecret})$
3 : $\text{groupInfo} \leftarrow (G'.\text{gid}, G'.\text{epoch}, G'.\text{memberPublicInfo}(), G'.\text{memberHash}, \boxed{G'.\text{confTransHash-w.o-'id}_c\text{'}}, G'.\text{confTransHash}, G'.\text{interimTransHash}, \text{confTag}, G'.\text{id})$
4 : $\text{sig} \leftarrow \text{SIG.Sign}(\text{pp}_{\text{SIG}}, G'.\text{ssk}, (\text{groupInfo}, \mathsf{T}))$
5 : $w_0 \leftarrow (\text{groupInfo}, \mathsf{T}, \text{sig})$
6 : $\vec{w} \leftarrow \emptyset$
7 : **foreach** $\text{id}_t \in \text{addedMem}$ **do**
8 : $\quad \text{kphash}_t \leftarrow \mathsf{H}(G'.\text{member}[\text{id}_t].\text{kp}())$
9 : $\quad \vec{w} \mathrel{+}\!\leftarrow \widehat{w}_{\text{id}_t} = (\text{id}_t, \text{kphash}_t, \widehat{\text{ct}}_{\text{id}_t})$
10 : **return** $(G', w_0, \vec{w})$

***initialize-group**$(G, \text{id}, \text{groupInfo})$

1 : **parse** $(\text{gid}, \text{epoch}, \text{member}, \text{memberHash}, \boxed{\text{confTransHash-w.o-'id}_c\text{'}}, \text{confTransHash}, \text{interimTransHash}, \text{confTag}, \text{id}_c) \leftarrow \text{groupInfo}$
2 : $(G.\text{gid}, G.\text{epoch}, G.\text{member}, G.\text{memberHash}, \boxed{G.\text{confTransHash-w.o-'id}_c\text{'}}, G.\text{confTransHash}, G.\text{interimTransHash}) \leftarrow (\text{gid}, \text{epoch}, \text{member}, \text{memberHash}, \boxed{\text{confTransHash-w.o-'id}_c\text{'}}, \text{confTransHash}, \text{interimTransHash})$
3 : $G.\text{certSvks}[*] \leftarrow \emptyset$
4 : $G.\text{pendUpd}[*] \leftarrow \perp$
5 : $G.\text{pendCom}[*] \leftarrow \perp$
6 : $G.\text{id} \leftarrow \text{id}$
7 : **return** $(G, \text{confTag}, \text{id}_c)$

***vrf-group-state**(G)

1 : **req** $G.\text{memberHash} = $ ***derive-member-hash**(G)
2 : $\text{mem} \leftarrow G.\text{memberIDs}()$
3 : **foreach** $\text{id} \in \text{mem}$ **do**
4 : $\quad \text{kp} \leftarrow G.\text{member}[\text{id}].\text{kp}()$
5 : $\quad$ **try** $G \leftarrow$ ***validate-kp**$(G, \text{kp}, \text{id})$
6 : **return** G

***fetch-ssk-if-nec**(G, svk)

1 : $\text{svk}_{\text{id}} \leftarrow G.\text{member}[G.\text{id}].\text{svk}$
2 : **if** $\text{svk}_{\text{id}} \neq \text{svk}$ **then**
3 : $\quad$ Send $(\texttt{get-ssk}, \text{svk})$ to $\mathcal{F}_{\text{AS}}$ and receive ssk
4 : **else**
5 : $\quad \text{ssk} \leftarrow G.\text{ssk}$
6 : **return** ssk

***validate-kp**$(G, \text{kp}, \text{id})$

1 : **parse** $(\text{id}', \text{ek}, \text{svk}, \text{sig}) \leftarrow \text{kp}$
2 : **req** $\text{id} = \text{id}'$
3 : **if** $\text{svk} \notin G.\text{certSvks}[\text{id}]$ **then**
4 : $\quad$ Send $(\texttt{verify-cert}, \text{id}', \text{svk})$ to $\mathcal{F}_{\text{AS}}$ and receive *succ*
5 : $\quad$ **req** *succ*
6 : $\quad G.\text{certSvks}[\text{id}] \mathrel{+}\!\leftarrow \text{svk}$
7 : **req** $\text{SIG.Verify}(\text{svk}, \text{sig}, (\text{id}, \text{ek}, \text{svk}))$
8 : **return** G

***assign-kp**(G, kp)

1 : **parse** $(\text{id}, \text{ek}, \text{svk}, \text{sig}) \leftarrow \text{kp}$
2 : $G.\text{member}[\text{id}].\text{ek} \leftarrow \text{ek}$
3 : $G.\text{member}[\text{id}].\text{svk} \leftarrow \text{svk}$
4 : $G.\text{member}[\text{id}].\text{sig} \leftarrow \text{sig}$
5 : **return** G

FIGURE 4.22: Helper functions of Chained CmPKE: Join and key material related. The major changes from [AJM22] are highlighted in gray. The boxed components are missing from prior works, which we believe are required to satisfy the UC functionality. Please see the proof for more detail.

`*gen-conf-tag`$(G, \mathsf{confKey})$

1 : $\mathsf{confTag} \leftarrow \mathsf{MAC.Gen}(\mathsf{confKey}, G.\mathsf{confTransHash})$
2 : **return** confTag

`*vrf-conf-tag`$(G, \mathsf{confKey}, \mathsf{confTag})$

1 : $\mathit{succ} \leftarrow \mathsf{MAC.Verify}(\mathsf{confKey}, \mathsf{confTag}, G.\mathsf{confTransHash})$
2 : **return** succ

`*set-member-hash`(G)

1 : $G.\mathsf{memberHash} \leftarrow$ `*derive-member-hash`(G)
2 : **return** G

`*derive-member-hash`(G)

1 : // mem is sorted by dictionary order
2 : $\mathsf{KP} \leftarrow ()$; $\mathsf{mem} \leftarrow G.\mathsf{memberIDs}()$
3 : **foreach** $\mathsf{id} \in \mathsf{mem}$ **do**
4 : $\quad \mathsf{KP} \mathbin{+\!\!\leftarrow} G.\mathsf{member}[\mathsf{id}].\mathsf{kp}()$
5 : **return** $\mathsf{H}(\mathsf{KP})$

`*set-conf-trans-hash`$(G, G', \mathsf{id}_c, C_0, \mathsf{sig})$

1 : $\boxed{\mathsf{comCont} \leftarrow (G.\mathsf{gid}, G.\mathsf{epoch}, \text{'commit'}, C_0, \mathsf{sig})}$
2 : $\boxed{G'.\mathsf{confTransHash\text{-}w.o\text{-}`id_c'} \leftarrow \mathsf{H}(G.\mathsf{interimTransHash}, \mathsf{comCont})}$
3 : $\boxed{G'.\mathsf{confTransHash} \leftarrow \mathsf{H}(G'.\mathsf{confTransHash\text{-}w.o\text{-}`id_c'}, \mathsf{id}_c)}$
4 : **return** G'

`*set-interim-trans-hash`$(G', \mathsf{confTag})$

1 : $G'.\mathsf{interimTransHash} \leftarrow \mathsf{H}(G'.\mathsf{confTransHash}, \mathsf{confTag})$
2 : **return** G'

`*derive-keys`$(G, G', \mathsf{comSecret})$

1 : $s \leftarrow \mathsf{HKDF.Extract}(G.\mathsf{initSecret}, \mathsf{comSecret})$
2 : $\mathsf{joinerSecret} \leftarrow \mathsf{HKDF.Expand}(s, \text{'joi'})$
3 : $(G', \mathsf{confKey}) \leftarrow$ `*derive-epoch-keys`$(G', \mathsf{joinerSecret})$
4 : **return** $(G', \mathsf{confKey}, \mathsf{joinerSecret})$

`*derive-epoch-keys`$(G', \mathsf{joinerSecret})$

1 : $\mathsf{GC} \leftarrow G'.\mathsf{groupCont}()$
2 : $\mathsf{confKey} \leftarrow \mathsf{HKDF.Expand}(\mathsf{joinerSecret}, \mathsf{GC} \| \text{'conf'})$
3 : $G'.\mathsf{appSecret} \leftarrow \mathsf{HKDF.Expand}(\mathsf{joinerSecret}, \mathsf{GC} \| \text{'app'})$
4 : $G'.\mathsf{membKey} \leftarrow \mathsf{HKDF.Expand}(\mathsf{joinerSecret}, \mathsf{GC} \| \text{'memb'})$
5 : $G'.\mathsf{initSecret} \leftarrow \mathsf{HKDF.Expand}(\mathsf{joinerSecret}, \mathsf{GC} \| \text{'init'})$
6 : **return** $(G', \mathsf{confKey})$

FIGURE 4.23: Helper functions of Chained CmPKE: computing tags, hash values, group secrets and framing and unframing packets. The major changes from [AJM22] are highlighted in gray. The boxed components are missing from prior works, which we believe are required to satisfy the UC functionality. Please see the proof for more detail.

*frame-prop(G, P)

1 : propCont $\leftarrow$ $(G.\mathsf{groupCont}(), G.\mathsf{id}, \text{'proposal'}, P)$
2 : sig $\leftarrow$ $\mathsf{SIG.Sign}(\mathsf{pp}_{\mathsf{SIG}}, G.\mathsf{ssk}, \mathsf{propCont})$
3 : membTag $\leftarrow$ $\mathsf{MAC.Gen}(G.\mathsf{membKey}, (\mathsf{propCont}, \mathsf{sig}))$
4 : **return** $(G.\mathsf{gid}, G.\mathsf{epoch}, G.\mathsf{id}, \text{'proposal'}, P, \mathsf{sig}, \mathsf{membTag})$

*unframe-prop(G, p)

1 : **parse** $(\mathsf{gid}, \mathsf{epoch}, \mathsf{id}_s, \mathsf{contType}, P, \mathsf{sig}, \mathsf{membTag}) \leftarrow p$
2 : **req** $\mathsf{contType} = \text{'proposal'} \wedge \mathsf{gid} = G.\mathsf{gid}$
$\wedge\ \mathsf{epoch} = G.\mathsf{epoch}$
3 : propCont $\leftarrow$ $(G.\mathsf{groupCont}(), \mathsf{id}_s, \text{'proposal'}, P)$
4 : **req** $G.\mathsf{member}[\mathsf{id}_s] \neq \bot$
$\wedge\ \mathsf{SIG.Verify}(G.\mathsf{member}[\mathsf{id}_s].\mathsf{svk}, \mathsf{sig}, \mathsf{propCont})$
$\wedge\ \mathsf{MAC.Verify}(G.\mathsf{membKey}, \mathsf{membTag}, (\mathsf{propCont}, \mathsf{sig}))$
5 : **return** (id_s, P)

*sign-commit(G, C_0)

1 : comCont $\leftarrow$ $(G.\mathsf{groupCont}(), G.\mathsf{id}, \text{'commit'}, C_0)$
2 : sig $\leftarrow$ $\mathsf{SIG.Sign}(\mathsf{pp}_{\mathsf{SIG}}, G.\mathsf{ssk}, \mathsf{comCont})$
3 : **return** sig

*frame-commit$(G, C_0, \mathsf{sig}, \mathsf{confTag})$

1 : **return** $(G.\mathsf{gid}, G.\mathsf{epoch}, G.\mathsf{id}, \text{'commit'}, C_0, \mathsf{sig}, \mathsf{confTag})$

*unframe-commit(G, c_0)

1 : **parse** $(\mathsf{gid}, \mathsf{epoch}, \mathsf{id}_c, \mathsf{contType}, C_0, \mathsf{sig}, \mathsf{confTag}) \leftarrow c_0$
2 : **req** $\mathsf{contType} = \text{'commit'} \wedge \mathsf{gid} = G.\mathsf{gid}$
$\wedge\ \mathsf{epoch} = G.\mathsf{epoch}$
3 : comCont $\leftarrow$ $(G.\mathsf{groupCont}(), \mathsf{id}_c, \text{'commit'}, C_0)$
4 : $\mathsf{svk}_c \leftarrow G.\mathsf{member}[\mathsf{id}_c].\mathsf{svk}$
5 : **req** $G.\mathsf{member}[\mathsf{id}_c] \neq \bot$
$\wedge\ \mathsf{SIG.Verify}(\mathsf{svk}_c, \mathsf{sig}, \mathsf{comCont})$
6 : **return** $(\mathsf{id}_c, C_0, \mathsf{sig}, \mathsf{confTag})$

FIGURE 4.24: Helper functions of Chained CmPKE: computing tags, hash values, group secrets and framing and unframing packets.

4.4.3 Security of Chained CmPKE

In this section, we provide the full security proof of our proposed protocol Chained CmPKE described in Section 4.4.2. We first explain the **safe** predicate used within the ideal functionality $\mathcal{F}_{\text{CGKA}}$ to exclude trivial attacks. The full security proof is provided subsequently.

Safety Predicates. Whether security is guaranteed in a given node (i.e, epoch) is determined via an explicit **safe** predicate on the node and the state of the history graph. This is the same approach taken by prior works [Alw+20b; AJM22]. Here, in addition to the secrecy of the keys, the functionality also implicitly formalizes authenticity by appropriately disallowing injections.

The safety predicate, depicted in Figure 4.25, is defined using recursive deducing rules **know**(c, id) and **know**$(c, \text{'epoch'})$.

know(c, id)**:** It indicates that the adversary knows id's key materials (e.g., decryption key) at epoch c. It consists of four conditions. Conditions (a) or (b) is true if id's key materials at epoch c are known to the adversary because they are exposed at c (Condition (a)) or injected by the adversary at c (Condition (b)). Conditions (c) and (d) reflect the fact that id's key materials will not change unless id commits, updates, is added, or is removed. If c does not change id's key materials, **know**(c, id) implies **know**$(\mathsf{Node}[c].\mathsf{par}, \mathsf{id})$ (Condition (c)). If a child c' does not change id's key materials, **know**(c, id) implies **know**(c', id) (Condition (d)).

know$(c, \text{'epoch'})$**:** It indicates that the adversary knows the epoch secrets (e.g., confirmation key) except for the application secret at epoch c. The adversary knows the epoch secrets if it corrupts a party at c, or if it computes them from the corrupted information. The latter is formalized by the ***can-traverse** predicate, which consists of three conditions. The first three conditions of ***can-traverse** reflect the fact that the epoch secrets (or the joiner secret to be more precise) can be computed from welcome messages: Condition (a) is true if a committer processes an injecting add proposals at c; Condition (b) is true if the adversary corrupts the decryption key in the key package used at c, and Condition (c) reflects the fact that the joiner secret is leaked if its ciphertext is generated with bad randomness. The last Condition (d) reflects the fact that the epoch secrets are derived from the initial secret at c's parent node and the commit secret.

The **safe** predicate indicates whether the adversary knows the application secret at epoch c. Since the application secret is leaked via a corruption query only if $\mathsf{HasKey}[\mathsf{id}] = \texttt{true}$ (i.e., a party did not output the application secret via `Key` query), **safe** checks whether $(*, \texttt{true}) \in \mathsf{Node}[c].\mathsf{exp}$. On the other hand, the other epoch secrets are always leaked when a party at c is corrupted. Thus, **know**$(c, \text{'epoch'})$ simply checks $\mathsf{Node}[c].\mathsf{exp} \neq \emptyset$.

The **sig-inj-allowed** and **mac-inj-allowed** predicates concern the authenticity of the signature and MAC, respectively. Since MAC keys (i.e., membership key and confirmation key) are a part of the epoch secrets, **mac-inj-allowed** is implied by **know**$(c, \text{'epoch'})$. These two predicates are used in `auth-invariant` (see Figure 4.14). Condition (a) and (b) of `auth-invariant` reflect the fact that injecting commit or proposal messages needs both a signing key of the sender and the MAC key. Condition (c) of `auth-invariant`, which was previously missing in [AJM22], reflects the fact that injecting welcome messages needs a signing key of the sender. Condition (c) was implicitly handled by the simulator within the security proof in [AJM22], but we believe explicitly including this condition makes the intuition of disallowing injection clear in the ideal functionality.

Security Statement. We prove our main theorem Theorem 4.4.1 that establishes the security of Chained CmPKE. Below, if we assume the CmPKE to be only IND-CCA secure, then it satisfies adaptive security

Knowledge of party's secrets.

$\mathbf{know}(c, \mathsf{id}) \iff$
(a) $(\mathsf{id}, *) \in \mathsf{Node}[c].\mathsf{exp} \vee$
(b) $^*\mathbf{secrets\text{-}injected}(c, \mathsf{id}) \vee$
(c) $(\mathsf{Node}[c].\mathsf{par} \neq \bot \wedge \mathbf{know}(\mathsf{Node}[c].\mathsf{par}, \mathsf{id})) \wedge \neg {}^*\mathbf{secrets\text{-}replaced}(c, \mathsf{id}) \vee$
(d) $\exists c' : (\mathsf{Node}[c'].\mathsf{par} = c \wedge \mathbf{know}(c', \mathsf{id})) \wedge \neg {}^*\mathbf{secrets\text{-}replaced}(c', \mathsf{id})$

$^*\mathbf{secrets\text{-}injected}(c, \mathsf{id}) \iff$
(a) $(\mathsf{Node}[c].\mathsf{orig} = \mathsf{id} \wedge \mathsf{Node}[c].\mathsf{stat} \neq \text{'good'}) \vee$
(b) $\exists p \in \mathsf{Node}[c].\mathsf{prop} : (\mathsf{Prop}[p].\mathsf{act} = \text{'upd'-}* \wedge \mathsf{Prop}[p].\mathsf{orig} = \mathsf{id} \wedge \mathsf{Prop}[p].\mathsf{stat} \neq \text{'good'}) \vee$
(c) $\exists p \in \mathsf{Node}[c].\mathsf{prop} : (\mathsf{Prop}[p].\mathsf{act} = \text{'add'-id-svk} \wedge \mathsf{svk} \in \mathsf{ExposedSvk})$

$^*\mathbf{secrets\text{-}replaced}(c, \mathsf{id}) \iff$
$\mathsf{Node}[c].\mathsf{orig} = \mathsf{id} \vee$
$\exists p \in \mathsf{Node}[c].\mathsf{prop} : \mathsf{Prop}[p].\mathsf{act} \in \{\text{'add'-id-}*, \text{'rem'-id}\} \vee$
$\exists p \in \mathsf{Node}[c].\mathsf{prop} : (\mathsf{Prop}[p].\mathsf{act} = \text{'upd'-}* \wedge \mathsf{Prop}[p].\mathsf{orig} = \mathsf{id})$

Knowledge of epoch secrets.

$\mathbf{know}(c, \text{'epoch'}) \iff \mathsf{Node}[c].\mathsf{exp} \neq \emptyset \vee {}^*\mathbf{can\text{-}traverse}(c)$

$^*\mathbf{can\text{-}traverse}(c) \iff$
(a) $\exists p \in \mathsf{Node}[c].\mathsf{prop} : (\mathsf{Prop}[p].\mathsf{act} = \text{'add'-id-svk} \wedge \mathsf{svk} \in \mathsf{ExposedSvk}) \vee$
(b) $^*\mathbf{reused\text{-}welcome\text{-}key\text{-}leaks}(c) \vee$
(c) $\mathsf{Node}[c].\mathsf{stat} = \text{'bad'} \wedge \exists p \in \mathsf{Node}[c].\mathsf{prop} : \mathsf{Prop}[p].\mathsf{act} = \text{'add'-}* \vee$
(d) $(c = \mathsf{root}_* \vee \mathbf{know}(\mathsf{Node}[c].\mathsf{par}, \text{'epoch'})) \wedge \exists (\mathsf{id}, *) \in \mathsf{Node}[c].\mathsf{mem} : \mathbf{know}(c, \mathsf{id})$

$^*\mathbf{reused\text{-}welcome\text{-}key\text{-}leaks}(c) \iff$
$\exists \mathsf{id}, p \in \mathsf{Node}[c].\mathsf{prop} : \mathsf{Prop}[p].\mathsf{act} = \text{'add'-id-}* \wedge$
$\exists c_d : c_d$ is a descendant of $c \wedge (\mathsf{id}, *) \in \mathsf{Node}[c_d].\mathsf{exp} \wedge$
no node c_h exists on c–c_d path s.t. $^*\mathbf{secrets\text{-}replaced}(c_h, \mathsf{id}) = \mathtt{true}$

Safe and can-inject.

$\mathbf{safe}(c) \iff \neg((*, \mathtt{true}) \in \mathsf{Node}[c].\mathsf{exp} \vee {}^*\mathbf{can\text{-}traverse}(c))$
$\mathbf{sig\text{-}inj\text{-}allowed}(c, \mathsf{id}) \iff \mathsf{Node}[c].\mathsf{mem}[\mathsf{id}] \in \mathsf{ExposedSvk}$
$\mathbf{mac\text{-}inj\text{-}allowed}(c) \iff \mathbf{know}(c, \text{'epoch'})$

FIGURE 4.25: The safety predicate for Chained CmPKE.

with an exponential security loss, while if we assume the CmPKE to be IND-CCA secure with adaptive corruption, then it satisfies adaptive security with only a polynomial security loss.

Theorem 4.4.1. *Assuming that* CmPKE *is* IND-CCA *secure (resp. with adaptive corruption) and has commitment-binding property, and* SIG *is* sEUF-CMA *secure, the* Chained CmPKE *protocol selectively (resp. adaptively) securely realizes the ideal functionality $\mathcal{F}_{CGKA}$, where $\mathcal{F}_{CGKA}$ uses the safety predicate from Figure 4.25, in the $(\mathcal{F}_{AS}, \mathcal{F}_{KS}, \mathcal{G}_{RO})$-hybrid model, where calls to the hash function* H, HKDF, *and* MAC *are replaced by a call to the global random oracle $\mathcal{G}_{RO}$.*

Remark 4.4.2 (Modeling HKDF and MAC as Random Oracle). Our proof relies on a variant of the *generalized selective decryption* (GSD) security as in the prior works [Kle+21; Alw+20b; AJM22], and it requires that HKDF.Expand and HKDF.Extract are modeled as a random oracle. More precisely, the reduction is expected to be able to extract a valid MAC secret key from the signature. To this end, we also model MAC as a random oracle to incorporate the MAC function into the GSD security.[10] We consider that the MAC tag is the hash value of the MAC key k and the message m, that is, $\mathsf{tag} := \mathsf{RO}(\mathsf{k}, \mathsf{m})$ where RO is a random oracle.

We first provide an overview of the proof before diving into the formal proof.

Proof Overview. The high level structure of the proof is similar to [Alw+20b; AJM22] who considered the UC security of TreeKEM. The main difference is due to the new **safe** predicate we introduce in order to differentiate between two types of injection attacks: one using signature schemes (see **sig-inj-allowed** in Figure 4.14) and the other using MAC (see **mac-inj-allowed** in Figure 4.14). Previously, these two types of injection attacks were handled within one hybrid but we differentiate them in hope to make the proof more clear.

Below, we provide an overview of the six hybrids we consider to establish security. We first consider the real world, denoted as Hybrid 1, where the environment $\mathcal{Z}$ is interacting with the real parties and the adversary $\mathcal{A}$. (To be more precise, $\mathcal{Z}$ is interacting with a simulator that internally runs all the real parties and adversary as in the real world).

In Hybrid 2, we swap the ideal authentication and key service $(\mathcal{F}_{\text{AS}}, \mathcal{F}_{\text{KS}})$ to their "ideal world" variant $(\mathcal{F}_{\text{AS}}^{\text{IW}}, \mathcal{F}_{\text{KS}}^{\text{IW}})$, which provides all the secret keys (i.e., secret keys of the signature scheme and CmPKE scheme) to the simulator. Since these functionalities are not accessible from $\mathcal{Z}$, one can think of these functions as being simulated by $\mathcal{S}$. In particular, this modification is only conceptual.

In Hybrid 3, we plug in a variant of the ideal functionality $\mathcal{F}_{\text{CGKA}}$ in between $\mathcal{Z}$ and the simulator, where the secrets are always set by the simulator and injections are always allowed (i.e., whether `auth-invariant` hold is never checked). This modification concerns the consistency between the protocol states of each user id and the history graph generated by the ideal functionality $\mathcal{F}_{\text{CGKA}}$. For instance, if id_1 and id_2 are assigned to the same node in the history graph, that is $\mathsf{Ptr}[\mathsf{id}_1] = \mathsf{Ptr}[\mathsf{id}_2]$, then we want their views in the real protocol to be identical, e.g., they agree on the same group member and group secret. Moreover, this hybrid establishes the correctness of the protocol.

In Hybrid 4, we modify the **sig-inj-allowed** predicate to be those used by the actual ideal functionality $\mathcal{F}_{\text{CGKA}}$. This establishes that an adversary cannot inject a malicious message that amounts to breaking the security of the signature scheme.

In Hybrid 5, we modify the **mac-inj-allowed** predicate to be consistent with those used by the actual ideal functionality $\mathcal{F}_{\text{CGKA}}$. This establishes that an adversary cannot inject a malicious message without knowing the MAC keys. As in most previous proofs [Kle+21; Alw+20b; AJM22], we rely on a variant

[10] Previous work [AJM22] assumes the standard EUF-CMA security of MAC, but did not provide a concrete proof. It seems it would be difficult to prove UC security by only assuming the standard EUF-CMA security of MAC.

of the *generalized selective decryption* (GSD) security, which we formally introduce as the Chained CmPKE *conforming GSD* security in Section 4.7. At a high level, the GSD security extracts the essence of the secrecy guarantee of the group secret and simplifies the proof. In this part, we first prove that if $\mathcal{Z}$ can distinguish between Hybrids 4 and 5, then it can be used to break the Chained CmPKE conforming GSD security. We then show in Section 4.7 that no efficient adversary can break the Chained CmPKE conforming GSD security assuming the security of CmPKE, which proves that Hybrids 4 and 5 remain the same in the view of $\mathcal{Z}$. We note that the variant of GSD security we introduce in this work is much more tailored to the CGKA setting than those previously considered and allows for a much simpler proof.

In Hybrid 6, we use the original **safe** predicate to be those used by the actual ideal functionality $\mathcal{F}_{\text{CGKA}}$. This establishes that the application secret looks random as long as **safe** is true for the epoch. We prove that if $\mathcal{Z}$ can distinguish between Hybrids 5 and 6, then it can be used to break the Chained CmPKE conforming GSD security of CmPKE. At this point, the functionality that sits between $\mathcal{Z}$ and the simulator is exactly $\mathcal{F}_{\text{CGKA}}$, thus we complete the proof. □

We then provide the full proof of Theorem 4.4.1.

Proof. We now provide a more formal proof of the above overview. Below, we use a sequence of hybrids explained above. We gradually modify the behavior of the simulator and denote the simulator in Hybrid i as $\mathcal{S}_i$. The first (resp. last) hybrid provides the environment $\mathcal{Z}$ the view of the real (resp. ideal) world. Below, when we say "the simulator aborts", we mean that the simulator terminates the simulation and does not respond to further queries made by the environment $\mathcal{Z}$.

Hybrid 1. This is the real world, where we make a syntactic change. We consider a simulator $\mathcal{S}_1$ that interacts with a dummy functionality $\mathcal{F}_{\text{dummy}}$ and $(\mathcal{F}_{\text{AS}}, \mathcal{F}_{\text{KS}})$. $\mathcal{F}_{\text{dummy}}$ sits between the environment $\mathcal{Z}$ and $\mathcal{S}_1$, and simply routs all messages without any modification. $\mathcal{S}_1$ internally runs the real world parties and adversary $\mathcal{A}$ by routing all messages sent from $\mathcal{F}_{\text{dummy}}$, which corresponds to those from $\mathcal{Z}$.

Hybrid 2. This change concerns the authentication and key service. In this world, $(\mathcal{F}_{\text{AS}}, \mathcal{F}_{\text{KS}})$ is replaced with $(\mathcal{F}_{\text{AS}}^{\text{IW}}, \mathcal{F}_{\text{KS}}^{\text{IW}})$. Since these functions are not accessible by $\mathcal{Z}$, this modification is undetectable from $\mathcal{Z}$. Hence, the view of $\mathcal{Z}$ in Hybrid 1 and Hybrid 2 are identical.[11]

Hybrid 3. This change concerns consistency guarantees. We replace $\mathcal{F}_{\text{dummy}}$ with a variant of $\mathcal{F}_{\text{CGKA}}$, denoted as $\mathcal{F}_{\text{CGKA,3}}$, where **safe** (resp. **sig-inj-allowed** and **mac-inj-allowed**) always returns false (resp. true). In other words, all application secrets are set by the simulator and injections are always allowed. The simulator $\mathcal{S}_3$ sets all messages and keys according to the protocol.

Hybrid 4. This change concerns the security of the signature scheme. We further modify $\mathcal{F}_{\text{CGKA,3}}$ to use the original **sig-inj-allowed** predicate, denoted as $\mathcal{F}_{\text{CGKA,4}}$. $\mathcal{F}_{\text{CGKA,4}}$ halts if a message is injected even if the sender's signing key is not exposed. The simulator $\mathcal{S}_4$ is identical to $\mathcal{S}_3$.

Hybrid 5. This change concerns the security of the MAC. We further modify $\mathcal{F}_{\text{CGKA,4}}$ to use the original **mac-inj-allowed** predicate, denoted as $\mathcal{F}_{\text{CGKA,5}}$. $\mathcal{F}_{\text{CGKA,5}}$ halts if a proposal or commit message is injected even if the corresponding MAC key is not exposed. The simulator $\mathcal{S}_5$ is identical to $\mathcal{S}_4$.

[11] Since $(\mathcal{F}_{\text{AS}}, \mathcal{F}_{\text{KS}})$ are local functions, we can instead simply assume that the simulator simulates these functionalities rather than replacing them. We use $(\mathcal{F}_{\text{AS}}^{\text{IW}}, \mathcal{F}_{\text{KS}}^{\text{IW}})$ to be consistent with the presentation provided in [AJM22].

Hybrid 6. This change concerns the confidentiality of application secrets. We further modify $\mathcal{F}_{\text{CGKA},5}$ where it uses the original **safe** predicate, denoted as $\mathcal{F}_{\text{CGKA},6}$. The simulator $\mathcal{S}_6$ is identical to $\mathcal{S}_5$ except that it sets only those application secrets for which **safe** is false. This functionality corresponds to the ideal functionality $\mathcal{F}_{\text{CGKA}}$.

We show indistinguishability of Hybrids 2 to 6 in Lemmata 4.4.3, 4.4.19, 4.4.23 and 4.4.33. This completes the proof of the main theorem. □

From Hybrid 2 to 3: Lemma 4.4.3. To show Lemma 4.4.3, we first consider additional hybrids (Hybrids 2-1 to 2-7) in between Hybrids 2 and 3 and show that each adjacent hybrids are indistinguishable. The most technically involved proof is showing the indistinguishability of Hybrids 2-4 and 2-5. All other hybrids are simply provided to make the proof between Hybrids 2-4 and 2-5 readable by taking care of subtleties such as decryption error, collisions in random oracles, and so on. Namely, for those interested readers, we believe it would be informative to check the proof between Hybrids 2-4 and 2-5 (Lemma 4.4.6) before checking the other hybrids.

Intermediate Hybrids. Here, we first provide the additional hybrids.

Hybrid 2-0 := **Hybrid 2.** This is identical to Hybrid 2.

Hybrid 2-1. [No collision in RO] This change concerns the collision resistance of the random oracle. Recall that all queries regarding the hash function H is simulated using the (global) random oracle. In this hybrid, we consider a simulator $\mathcal{S}_{2\text{-}1}$ that aborts when a collision ever occurs in the random oracle. Since $\mathcal{Z}$ only makes at most polynomially many queries, this makes negligible change to the view of $\mathcal{Z}$. Hence, the view of $\mathcal{Z}$ in Hybrid 2-0 and Hybrid 2-1 are negligibly different.

Hybrid 2-2. [Unique $\mathsf{confTag}$/$\mathsf{membTag}$ **in** L_{prop}/L_{com}/L_{wel} **]** This concerns the uniqueness of $\mathsf{membTag}$ included in a proposal message and the uniqueness of $\mathsf{confTag}$ included in commit and welcome messages. We consider a simulator $\mathcal{S}_{2\text{-}2}$ defined exactly as $\mathcal{S}_{2\text{-}1}$ except that it maintains three lists L_{prop}, L_{com}, and L_{wel} all initially set to $\emptyset$ and performs the following additional checks.

Checks regarding L_{prop}. Let G be the protocol state of party id *before* being invoked by $\mathcal{S}_{2\text{-}2}$. There are three checks $\mathcal{S}_{2\text{-}2}$ performs. First, when $\mathcal{S}_{2\text{-}2}$ invokes party id on input $(\texttt{Propose}, \mathsf{act})$, if id outputs a proposal message p, then $\mathcal{S}_{2\text{-}2}$ extracts $\mathsf{membTag}$ included in p (which is guaranteed to exist) and checks if there exists an entry $(p', \mathsf{membKey}, \mathsf{membTag}) \in L_{\mathsf{prop}}$ such that $(p', \mathsf{membKey}) \neq (p, G.\mathsf{membKey})$. If so $\mathcal{S}_{2\text{-}2}$ aborts. Otherwise it updates the list $L_{\mathsf{prop}} \xleftarrow{+} (p, G.\mathsf{membKey}, \mathsf{membTag})$. Second, when $\mathcal{S}_{2\text{-}2}$ invokes party id on input $(\texttt{Commit}, \vec{p}, \mathsf{svk})$, if id outputs non-$\perp$, then $\mathcal{S}_{2\text{-}2}$ extracts $\mathsf{membTag}$ included in each $p \in \vec{p}$ (which is guaranteed to exist) and performs the same procedure above for each p. Finally, when $\mathcal{S}_{2\text{-}2}$ invokes party id on input $(\texttt{Propose}, c_0, \widehat{c}, \vec{p})$, if id outputs non-$\perp$, then $\mathcal{S}_{2\text{-}2}$ extracts $\mathsf{membTag}$ included in each $p \in \vec{p}$ (which is guaranteed to exist) and performs the same procedure above for each p.

Checks regarding L_{com}. Let G be the protocol state of party id *after* being invoked by $\mathcal{S}_{2\text{-}2}$. There are two checks $\mathcal{S}_{2\text{-}2}$ performs. First, when $\mathcal{S}_{2\text{-}2}$ invokes party id on input $(\texttt{Commit}, \vec{p}, \mathsf{svk})$, if id outputs $(c_0, \vec{c}, w_0, \vec{w})$, then $\mathcal{S}_{2\text{-}2}$ extracts $\mathsf{confTag}$ included in c_0 (which is guaranteed to exist). Moreover, let $G' = G.\mathsf{pendCom}[c_0]$. Then, $\mathcal{S}_{2\text{-}2}$ checks if there exists an entry $(c_0', \mathsf{confKey}, \mathsf{confTransHash}, \mathsf{confTag}) \in L_{\mathsf{com}}$ where we have $(c_0', \mathsf{confKey}, \mathsf{confTransHash}) \neq (c_0, G'.\mathsf{confKey}, G'.\mathsf{confTransHash})$. If so $\mathcal{S}_{2\text{-}2}$ aborts, and otherwise it updates the list $L_{\mathsf{com}} \xleftarrow{+} (c_0, G'.\mathsf{confKey}, G'.\mathsf{confTransHash}, \mathsf{confTag})$.

Second, when $\mathcal{S}_{2\text{-}2}$ invokes party id on input $(\texttt{Propose}, c_0, \widehat{c}, \vec{p})$, if id outputs non-$\perp$, then $\mathcal{S}_{2\text{-}2}$ extracts confTag included in c_0 (which is guaranteed to exist) and checks if there exists an entry $(c_0', \mathsf{confKey}, \mathsf{confTransHash}, \mathsf{confTag}) \in L_{\mathsf{com}}$ where $(c_0', \mathsf{confKey}, \mathsf{confTransHash}) \neq (c_0, G.\mathsf{confKey}, G.\mathsf{confTransHash})$. If so $\mathcal{S}_{2\text{-}2}$ aborts. Otherwise it updates $L_{\mathsf{com}} \mathrel{+}\!\leftarrow (c_0, G.\mathsf{confKey}, G.\mathsf{confTransHash}, \mathsf{confTag})$.

Checks regarding L_{wel}. Let G be the protocol state of party id *after* being invoked by $\mathcal{S}_{2\text{-}2}$. There are two checks $\mathcal{S}_{2\text{-}2}$ performs. First, when $\mathcal{S}_{2\text{-}2}$ invokes party id on input $(\texttt{Commit}, \vec{p}, \mathsf{svk})$, if id outputs $(c_0, \vec{c}, w_0, \vec{w})$, then $\mathcal{S}_{2\text{-}2}$ parses w_0 as $(\mathsf{groupInfo}, \mathsf{T}, \mathsf{sig})$ and extracts confTag included in groupInfo (which is guaranteed to exist). Moreover, let $G' = G.\mathsf{pendCom}[c_0]$. It then checks if there exists an entry $(\mathsf{groupInfo}, \mathsf{confKey}, \mathsf{confTransHash}, \mathsf{confTag}) \in L_{\mathsf{wel}}$ where $(\mathsf{groupInfo}', \mathsf{confKey}, \mathsf{confTransHash}) \neq (\mathsf{groupInfo}, G'.\mathsf{confKey}, G'.\mathsf{confTransHash})$. If so $\mathcal{S}_{2\text{-}2}$ aborts and otherwise it updates the list $L_{\mathsf{wel}} \mathrel{+}\!\leftarrow (\mathsf{groupInfo}, G'.\mathsf{confKey}, G'.\mathsf{confTransHash}, \mathsf{confTag})$. Second, when $\mathcal{S}_{2\text{-}2}$ invokes party id on input $(\texttt{Join}, w_0, \widehat{w})$, if id outputs non-$\perp$, then $\mathcal{S}_{2\text{-}2}$ parses w_0 as $(\mathsf{groupInfo}, \mathsf{T}, \mathsf{sig})$ and extracts confTag included in groupInfo and checks if there exists an entry $(\mathsf{groupInfo}, \mathsf{confKey}, \mathsf{confTransHash}, \mathsf{confTag}) \in L_{\mathsf{wel}}$ where $(\mathsf{groupInfo}', \mathsf{confKey}, \mathsf{confTransHash}) \neq (\mathsf{groupInfo}, G.\mathsf{confKey}, G.\mathsf{confTransHash})$. If so $\mathcal{S}_{2\text{-}2}$ aborts. Otherwise it updates $L_{\mathsf{wel}} \mathrel{+}\!\leftarrow (\mathsf{groupInfo}, G.\mathsf{confKey}, G.\mathsf{confTransHash}, \mathsf{confTag})$.

Checks regarding L_{com} and L_{wel}. Every time $\mathcal{S}_{2\text{-}2}$ updates L_{com} or L_{wel}, it checks if there exists $(c_0, \mathsf{confKey}, \mathsf{confTransHash}, \mathsf{confTag}) \in L_{\mathsf{com}}$ and $(\mathsf{groupInfo}, \mathsf{confKey}', \mathsf{confTransHash}', \mathsf{confTag}') \in L_{\mathsf{wel}}$ such that $\mathsf{confTag} = \mathsf{confTag}'$ but $(\mathsf{id}_c, \mathsf{confKey}, \mathsf{confTransHash}) \neq (\mathsf{id}_c', \mathsf{confKey}', \mathsf{confTransHash}')$, where id_c and id_c' are the (purported) user identity included in c_0 and groupInfo, respectively. If so, $\mathcal{S}_{2\text{-}2}$ aborts.

We show in Lemma 4.4.4 that Hybrid 2-1 and Hybrid 2-2 are indistinguishable to $\mathcal{Z}$ assuming collision resistance of MAC.

Hybrid 2-3. [Unique c_0 with good randomness] This concerns the uniqueness of a commit message with good randomness. Assume party id outputs $(c_0, \vec{c}, w_0, \widehat{w})$ (where possibly $(w_0, \widehat{w}) = (\perp, \perp)$) on input $(\texttt{Commit}, \vec{p}, \mathsf{svk})$, where $\mathsf{Rand}[\mathsf{id}] = \text{'good'}$. Let $G' = G.\mathsf{pendCom}[c_0]$ and G be the protocol state after being invoked by $\mathcal{S}_{2\text{-}2}$. We consider a simulator $\mathcal{S}_{2\text{-}3}$ that aborts if the same $(c_0, G'.\mathsf{confKey}, G'.\mathsf{confTransHash})$ is already included in L_{com}. Recall that in the previous hybrid, we did not abort in case the same entry was found. $\mathcal{S}_{2\text{-}3}$ is otherwise defined identically to $\mathcal{S}_{2\text{-}2}$.

Now, in case $\mathsf{Rand}[\mathsf{id}] = \text{'good'}$, due to the ciphertext-spreadness of CmPKE (see Definition 4.2.4), c_0 has high min-entropy. Therefore, the probability of c_0 already being in L_{com} is negligibly small. Hence, Hybrid 2-2 and Hybrid 2-3 are indistinguishable to $\mathcal{Z}$.

Hybrid 2-4. [Consistency of no-join] This concerns the consistency of confTag included in commit and welcome messages. Assume party id outputs $(c_0, \vec{c}, \perp, \perp)$ on input $(\texttt{Commit}, \vec{p}, \mathsf{svk})$. That is, there are no newly added members to the group. If any party id$'$ ever correctly processes $(\texttt{Join}, w_0, \widehat{w})$ (i.e., id$'$ outputs $(\mathsf{id}_c, \mathsf{mem})$) and w_0 includes the same confTag as the one included in c_0, then $\mathcal{S}_{2\text{-}4}$ aborts. Otherwise, $\mathcal{S}_{2\text{-}4}$ is identical to the previous simulator. Informally, this implies that confTag implicitly commits to the information of the group members and if confTag was generated as a result of no new additions, then confTag cannot be used as a welcome message. We show in Lemma 4.4.5 that Hybrid 2-3 and Hybrid 2-4 are indistinguishable to $\mathcal{Z}$ assuming collision resistance of MAC.

Hybrid 2-5. [Adding consistency checks] This change concerns consistency guarantees. We replace $\mathcal{F}_{\text{dummy}}$ with a variant of $\mathcal{F}_{\text{CGKA}}$, denoted as $\mathcal{F}_{\text{CGKA,2-5}}$, where **safe** (resp. **sig-inj-allowed** and **mac-inj-allowed**)

always returns false (resp. true), and the correctness conditions `*succeed-com`, `*succeed-proc`, and `*succeed-wel` always output false. In other words, all application secrets are set by the simulator, injections are always allowed, and the protocol does not need to satisfy correctness. However, the simulator $\mathcal{S}_{2\text{-}5}$ does set all messages and keys according to the protocol. We show in Lemma 4.4.6 that Hybrid 2-4 and Hybrid 2-5 are identical.

Hybrid 2-6. [No correctness error] This change concerns the correctness of the signature scheme and encryption scheme. We replace $\mathcal{F}_{\text{CGKA,2-5}}$ with $\mathcal{F}_{\text{CGKA,2-6}}$, where the only difference is that the correctness conditions `*succeed-com`, `*succeed-proc`, and `*succeed-wel` defined as those in the ideal functionality $\mathcal{F}_{\text{CGKA}}$. At a high level, these correctness conditions guarantee that if the real protocol is run as expected, then there should be no correctness error. Moreover, this should hold true even if bad randomness is used.[12] We show in Lemma 4.4.17 that Hybrid 2-5 and Hybrid 2-6 are identical.

Hybrid 2-7. [Unique $\widehat{c}$ for each c_0 and id] This change concerns the uniqueness of party-dependent commitments $\widehat{c}$. We replace $\mathcal{F}_{\text{CGKA,2-6}}$ with $\mathcal{F}_{\text{CGKA,2-7}}$, where the only difference is that $\mathcal{F}_{\text{CGKA,2-7}}$ always outputs $\perp$ when $\widehat{c} \neq \mathsf{Node}[c_0].\mathsf{vcom}[\mathsf{index}_{\mathsf{id}}]$ in `Process` protocol. This condition says, for each c_0 and id, the designated $\widehat{c}$ in `Commit` is the only party-dependent commitment that id accepts. We show in Lemma 4.4.18 that Hybrid 2-6 and Hybrid 2-7 are identical.

Hybrid 2-8 := Hybrid 3. [Removing abort conditions] This is identical to Hybrid 3. The only difference between Hybrid 2-7 is that the simulator $\mathcal{S}_{2\text{-}8} = \mathcal{S}_3$ no longer aborts the simulation. Specifically, we remove all the abort conditions checked by the simulator that was introduced from moving to Hybrid 2-0 to 2-5. Using the same arguments to move through Hybrid 2-0 to Hybrid 2-5, Hybrid 2-7 and Hybrid 2-8 remain indistinguishable.

The following is the main lemma of this section which proves indistinguishability between Hybrid 2 and 3. The proof is a direct consequence of the argument made in Section 4.4.3 and the subsequent Lemmata 4.4.4 to 4.4.6.

Lemma 4.4.3. *Hybrid 2 and Hybrid 3 are indistinguishable assuming the collision resistance of* MAC*, the correctness of* CmPKE *and* SIG*, and the ciphertext-spreadness of* CmPKE*.*

From Hybrid 2-1 to 2-2: Lemma 4.4.4.

Lemma 4.4.4. *Hybrid 2-1 and Hybrid 2-2 are indistinguishable assuming* MAC *is collision-resistant.*

Proof. We first consider the case $\mathcal{S}_{2\text{-}2}$ aborts while checking the list L_{prop}. $\mathcal{S}_{2\text{-}2}$ checks the list during either a propose, commit, or process query. Assume $\mathcal{S}_{2\text{-}2}$ was invoking party id on a propose query. By correctness of the propose protocol, if id outputs $p = (\mathsf{gid}, \mathsf{epoch}, \mathsf{id}, \text{'proposal'}, P, \mathsf{sig}, \mathsf{membTag})$, then we have the following

- $G.\mathsf{gid} = \mathsf{gid}$;

[12] Unlike classical schemes (e.g., ElGamal encryption), there are correctness errors in post-quantum schemes such as those based on lattices. Looking ahead, we argue that no adversary can find a bad randomness that leads to a correctness error by requiring the underlying cryptographic primitives to expand the randomness through a hash function model as a random oracle. Concretely, the adversary can only control the random seed, which is then expanded via a hash function (or more precisely a PRG) modeled as a random oracle.

- $G.\mathsf{epoch} = \mathsf{epoch}$;
- $G.\mathsf{groupCont}() = (G.\mathsf{gid}, G.\mathsf{epoch}, G.\mathsf{memberHash}, G.\mathsf{confTransHash})$;
- $\mathsf{membTag} = \mathsf{MAC.Gen}(G.\mathsf{membKey}, (G.\mathsf{groupCont}(), \mathsf{id}, \text{'proposal'}, P, \mathsf{sig}))$,

where recall G is the protocol state of id. Notice that the entire description of p is included as a message of membTag. Then if there exists $(p', \mathsf{membKey}') \neq (p, G.\mathsf{membKey})$ then it can be used to break collision resistance of MAC. The proof for the other cases where $\mathcal{S}_{2\text{-}2}$ was invoking party id on a commit or process query is identical to the above. Therefore, assuming collision resistance of MAC, the abort condition regarding L_{prop} cannot occur.

We next consider the case $\mathcal{S}_{2\text{-}2}$ aborts while checking the list L_{com}. $\mathcal{S}_{2\text{-}2}$ checks the list during either a commit or process query. Assume $\mathcal{S}_{2\text{-}2}$ was invoking party id on a commit query. By correctness of the commit protocol, if id outputs $c_0 = (\mathsf{gid}, \mathsf{epoch}, \mathsf{id}_c, \text{'commit'}, C_0 = (\mathsf{propIDs}, \mathsf{kp}, \mathsf{T}), \mathsf{sig}, \mathsf{confTag})$, then we have the following

- $G'.\mathsf{gid} = G.\mathsf{gid} = \mathsf{gid}$;
- $G'.\mathsf{epoch} = G.\mathsf{epoch} + 1 = \mathsf{epoch} + 1$;
- $G'.\mathsf{confTransHash\text{-}w.o\text{-}'id_c'} = \mathsf{H}(G.\mathsf{interimTransHash}, (G.\mathsf{gid}, \mathsf{epoch}, \text{'commit'}, C_0, \mathsf{sig}))$;
- $G'.\mathsf{confTransHash} = \mathsf{H}(G'.\mathsf{confTransHash\text{-}w.o\text{-}'id_c'}, \mathsf{id}_c)$;
- $G'.\mathsf{groupCont}() = (G'.\mathsf{gid}, G'.\mathsf{epoch}, G'.\mathsf{memberHash}, G'.\mathsf{confTransHash})$;
- $G'.\mathsf{confKey} = \mathsf{H}(G'.\mathsf{joinerSecret}, G'.\mathsf{groupCont}(), \text{'conf'})$;
- $\mathsf{confTag} = \mathsf{MAC.Gen}(G'.\mathsf{confKey}, G'.\mathsf{confTransHash})$;

where recall G' is the pending protocol state of id included in $G.\mathsf{pendCom}[c_0]$. Due to the modification we made in Hybrid 2-1 [*No collision in RO*], c_0 is the unique commitment that leads to $G'.\mathsf{confKey}$. Namely, for any $(c_0', \mathsf{confKey}, \mathsf{confTransHash}, \mathsf{confTag}) \in L_{\mathsf{com}}$, we have $\mathsf{confKey} \neq G'.\mathsf{confKey}$ if $c_0' \neq c_0$. Then, regardless of $c_0' \neq c_0$ or $c_0' = c_0$, we would have $(\mathsf{confKey}, \mathsf{confTransHash}) \neq (G'.\mathsf{confKey}, G'.\mathsf{confTransHash})$. Hence, the abort condition in L_{com} does not occur. The proof for the other case where $\mathcal{S}_{2\text{-}2}$ was invoking party id on a process query is identical to the above, where the only difference is that G' is the updated protocol state of id rather then the pending protocol state. Therefore, assuming collision resistance of MAC, the abort condition regarding L_{com} cannot occur.

We next consider the case $\mathcal{S}_{2\text{-}2}$ aborts while checking the list L_{wel}. $\mathcal{S}_{2\text{-}2}$ checks the list during either a commit or join query. Assume $\mathcal{S}_{2\text{-}2}$ was invoking party id on a commit query. Then, by correctness of the commit protocol, if id outputs $w_0 = (\mathsf{groupInfo} = (\mathsf{gid}, \mathsf{epoch}, \mathsf{memberPublicInfo}, \mathsf{memberHash}, \mathsf{confTransHash}$ $\mathsf{-w.o.-'id}_c\mathsf{'}, \mathsf{confTransHash}, \mathsf{interimTransHash}, \mathsf{confTag}, \mathsf{id}_c), \mathsf{T}, \mathsf{sig})$ then we have the above listed relations considered during L_{com}. Notice due to the modification we made in Hybrid 2-1 [*No collision in RO*], $(\mathsf{gid}$, $\mathsf{epoch}, \mathsf{memberHash}, \mathsf{confTransHash\text{-}w.o\text{-}'id_c'}, \mathsf{id}_c)$ is the unique pair that leads to a valid memberPublicInfo and interimTransHash[13], where recall memberHash is set via the helper function `*derive-member-hash` (see Figure 4.23). This in particular implies for any $(\mathsf{groupInfo}', \mathsf{confKey}, \mathsf{confTransHash}, \mathsf{confTag}) \in L_{\mathsf{wel}}$, we have $\mathsf{confKey} \neq G'.\mathsf{confKey}$ if $\mathsf{groupInfo}' \neq \mathsf{groupInfo}$. Then, regardless of $\mathsf{groupInfo}' \neq \mathsf{groupInfo}$ or $\mathsf{groupInfo}' = \mathsf{groupInfo}$, we have $(\mathsf{confKey}, \mathsf{confTransHash}) \neq (G'.\mathsf{confKey}, G'.\mathsf{confTransHash})$. Hence, the

[13]Here, we explicitly rely on the new $\mathsf{confTransHash\text{-}w.o\text{-}'id_c'}$ satisfying $\mathsf{confTransHash} = \mathsf{H}(\mathsf{confTransHash\text{-}w.o\text{-}'id_c'}, \mathsf{id}_c)$.

abort condition in L_{wel} does not occur. The proof for the other case where $\mathcal{S}_{2\text{-}2}$ was invoking party id on a join query is identical to the above. Therefore, assuming collision resistance of MAC, the abort condition regarding L_{wel} cannot occur.

We finally consider the case $\mathcal{S}_{2\text{-}2}$ aborts while checking both of the lists L_{com} and L_{wel}. It is clear that we cannot have $\mathsf{confTag} = \mathsf{confTag}'$ while $(\mathsf{confKey}, \mathsf{confTransHash}) \neq (\mathsf{confKey}', \mathsf{confTransHash}')$ since this can be directly used to break collision resistance of MAC. However, recall $\mathsf{confTransHash}$ is created by hashing $\mathsf{confTransHash\text{-}w.o\text{-}`id}_c\text{'}$ and id_c. Therefore, due to the modification we made in Hybrid 2-1 [*No collision in RO*], id_c and id'_c must be the same as well. This establishes $(\mathsf{id}_c, \mathsf{confKey}, \mathsf{confTransHash}) = (\mathsf{id}'_c, \mathsf{confKey}', \mathsf{confTransHash}')$.

This completes the proof. □

From Hybrid 2-3 to 2-4: Lemma 4.4.5.

Lemma 4.4.5. *Hybrid 2-3 and Hybrid 2-4 are indistinguishable assuming* MAC *is collision-resistant.*

Proof. Let G_{id} and $G'_{\mathsf{id}'}$ be the protocol states of id and id' after they execute the commit and join query, respectively. Moreover, let G'_{id} be the pending protocol state stored in $G_{\mathsf{id}}.\mathsf{pendCom}[c_0]$. Then, by the correctness of the protocol, since the commit c_0 created by id does not include new parties, we must have $G'_{\mathsf{id}}.\mathsf{memberIDsvks}() \neq G'_{\mathsf{id}'}.\mathsf{memberIDsvks}()$. Then, due to the modification we made in Hybrid 2-1 [*No collision in RO*] and taking into consideration of how $\mathsf{confKey}$ is generated, we must have $G'_{\mathsf{id}}.\mathsf{confKey} \neq G'_{\mathsf{id}'}.\mathsf{confKey}$. However, this cannot happen since otherwise $\mathcal{S}_{2\text{-}4}$ can break collision resistance of MAC by outputting $(G'_{\mathsf{id}}.\mathsf{confKey}, G'_{\mathsf{id}}.\mathsf{confTransHash}, G'_{\mathsf{id}'}.\mathsf{confKey}, G'_{\mathsf{id}'}.\mathsf{confTransHash}, \mathsf{confTag})$.

This completes the proof. □

From Hybrid 2-4 to 2-5: Lemma 4.4.6. This is the technically most involved lemma which checks the consistency between the real protocol and the ideal protocol. The proof consists of three parts: we first formally define the behavior of simulator $\mathcal{S}_{2\text{-}5}$ in Hybrid 2-5 (see *Part 1*); we then provide supporting propositions that establish consistencies between the protocol states and the history graph (see *Part 2*); finally, using the supporting propositions, we analyze the simulation provided by $\mathcal{S}_{2\text{-}5}$ provides an identical view to $\mathcal{Z}$ as in Hybrid 2-4 (see *Part 3*).

Part 1. Description of the Simulator $\mathcal{S}_{2\text{-}5}$. Throughout the hybrid, $\mathcal{S}_{2\text{-}5}$ creates the same history graph created within $\mathcal{F}_{\mathrm{CGKA,2\text{-}5}}$. That is, it initializes $\mathsf{Ptr}[*]$, $\mathsf{Node}[*]$, $\mathsf{Prop}[*]$, and so on and maintains the same view as $\mathcal{F}_{\mathrm{CGKA,2\text{-}5}}$. Moreover, throughout this hybrid, we augment the protocol state G of party id to also maintain the values presented in Table 4.5. Although these values are deleted once the protocol state is updated in the real protocol, e.g., after processing a process query, we can keep these without loss of generality as they are never provided to the environment $\mathcal{Z}$ or the adversary $\mathcal{A}$. In particular, they will simply be helpful objects to discuss the consistency of the simulation.

The description of $\mathcal{S}_{2\text{-}5}$ consists of how it answers each queries made by the ideal functionality $\mathcal{F}_{\mathrm{CGKA,2\text{-}5}}$. Here, note that any queries made by $\mathcal{Z}$ to $\mathcal{S}_{2\text{-}5}$ will be simply relayed to the internally simulated $\mathcal{A}$. Moreover, $\mathcal{S}_{2\text{-}5}$ aborts the simulation whenever any of the checks we included in Hybrids 2-1 to 2-5 are triggered.

(1) Create query from $\mathsf{id}_{\mathsf{creator}}$. This concerns the case when $\mathcal{Z}$ queries $(\mathtt{Create}, \mathsf{svk})$ to $\mathcal{F}_{\mathrm{CGKA,2\text{-}5}}$. If $\mathcal{F}_{\mathrm{CGKA,2\text{-}5}}$ outputs $(\mathtt{Create}, \mathsf{id}_{\mathsf{creator}}, \mathsf{svk})$ to $\mathcal{S}_{2\text{-}5}$, $\mathcal{S}_{2\text{-}5}$ simply runs the simulated party $\mathsf{id}_{\mathsf{creator}}$ on input $(\mathtt{Create}, \mathsf{svk})$.

(2) Propose query from id**.** This concerns the case when $\mathcal{Z}$ queries $(\texttt{Propose}, \mathsf{act})$ for some $\mathsf{act} \in \{\text{'upd'-svk}, \text{'add'-id}_t, \text{'rem'-id}_t\}$ to $\mathcal{F}_{\text{CGKA,2-5}}$. If $\mathsf{Ptr}[\mathsf{id}] \neq \perp$, then $\mathcal{F}_{\text{CGKA,2-5}}$ outputs $(\texttt{Propose}, \mathsf{id}, \mathsf{act})$ to $\mathcal{S}_{\text{2-5}}$. $\mathcal{S}_{\text{2-5}}$ then runs the simulated party id on input $(\texttt{Propose}, \mathsf{act})$, where it asks $\mathcal{A}$ to provide the randomness to run party id if $\mathsf{Rand}[\mathsf{id}] = \text{'bad'}$ and $\mathsf{act} = \text{'upd'-svk}$. Here, recall randomness is only used to generate a new key package $(\mathsf{kp}, \mathsf{dk})$.[14] If party id returns $\perp$, then $\mathcal{S}_{\text{2-5}}$ returns $(ack := \texttt{false}, \perp, \perp)$ to $\mathcal{F}_{\text{CGKA,2-5}}$. Otherwise, if id returns p, then $\mathcal{S}_{\text{2-5}}$ returns $(ack := \texttt{true}, p, \mathsf{svk}_t)$, where a long-term key $\mathsf{svk}_t \neq \perp$ is extracted from p only when $\mathsf{act} = \text{'add'-id}_t$.

(3) Commit query from id**.** This concerns the case when $\mathcal{Z}$ queries $(\texttt{Commit}, \vec{p}, \mathsf{svk})$ to $\mathcal{F}_{\text{CGKA,2-5}}$. If $\mathsf{Ptr}[\mathsf{id}] \neq \perp$, then $\mathcal{F}_{\text{CGKA,2-5}}$ outputs $(\texttt{Commit}, \mathsf{id}, \vec{p}, \mathsf{svk})$ to $\mathcal{S}_{\text{2-5}}$. $\mathcal{S}_{\text{2-5}}$ then runs the simulated party id on input $(\texttt{Commit}, \vec{p}, \mathsf{svk})$, where it asks $\mathcal{A}$ to provide the randomness to run party id if $\mathsf{Rand}[\mathsf{id}] = \text{'bad'}$. If party id returns $\perp$, then $\mathcal{S}_{\text{2-5}}$ returns $(ack := \texttt{false}, \perp, \perp, \perp, \perp, \perp)$ to $\mathcal{F}_{\text{CGKA,2-5}}$. Otherwise, if party id returns $(c_0, \vec{c}, w_0, \vec{w})$, then it checks if $\mathsf{Node}[c_0] = \perp$, $w_0 \neq \perp$, and if there exists some $rt' \in \mathbb{N}$ and w_0' such that $\mathsf{Wel}[w_0'] = \mathsf{root}_{rt'}$ and w_0' includes the same confTag as w_0. If so, $\mathcal{S}_{\text{2-5}}$ chooses any such (w_0', rt') and returns $(ack := \texttt{true}, rt := rt', c_0, \vec{c}, w_0, \vec{w})$ to $\mathcal{F}_{\text{CGKA,2-5}}$. As we show in Proposition 4.4.7 below, such for any such pair, the value of rt' is unique. Otherwise, if either $\mathsf{Node}[c_0] \neq \perp$; or $w_0 = \perp$; or there does not exist w_0' such that $\mathsf{Wel}[w_0'] = \mathsf{root}_{rt'}$ for some $rt' \in \mathbb{N}$ and w_0' includes the same confTag as w_0, then $\mathcal{S}_{\text{2-5}}$ returns $(ack := \texttt{true}, rt := \perp, c_0, \vec{c}, w_0, \vec{w})$ to $\mathcal{F}_{\text{CGKA,2-5}}$. Finally, when $\mathcal{F}_{\text{CGKA,2-5}}$ queries $(\texttt{Propose}, p)$ to $\mathcal{S}_{\text{2-5}}$ during the `*fill-prop` check, $\mathcal{S}_{\text{2-5}}$ extracts the unique $\mathsf{orig} = \mathsf{id}$ and act included in p (which are guaranteed to exist when commit succeeds in the real protocol) and returns them to $\mathcal{F}_{\text{CGKA,2-5}}$.

(4) Process query from id**.** This concerns the case when $\mathcal{Z}$ queries $(\texttt{Process}, c_0, \widehat{c}, \vec{p})$ to $\mathcal{F}_{\text{CGKA,2-5}}$. If $\mathsf{Ptr}[\mathsf{id}] \neq \perp$, then $\mathcal{F}_{\text{CGKA,2-5}}$ outputs $(\texttt{Process}, \mathsf{id}, c_0, \widehat{c}, \vec{p})$ to $\mathcal{S}_{\text{2-5}}$. $\mathcal{S}_{\text{2-5}}$ then (deterministically) runs the simulated party id on input $(\texttt{Process}, c_0, \widehat{c}, \vec{p})$. If party id returns $\perp$, then $\mathcal{S}_{\text{2-5}}$ returns $(ack := \texttt{false}, \perp, \perp, \perp)$ to $\mathcal{F}_{\text{CGKA,2-5}}$. Otherwise, if party id returns $(\mathsf{id}_c, upd \| rem \| add)$, then $\mathcal{S}_{\text{2-5}}$ checks if $\mathsf{Node}[c_0] = \perp$ and if there exists w_0 that includes the same confTag as c_0 such that $\mathsf{Wel}[w_0] = \mathsf{root}_{rt'}$ for some $rt' \in \mathbb{N}$. If so, $\mathcal{S}_{\text{2-5}}$ chooses any such (w_0, rt') and returns $(ack := \texttt{true}, rt := rt', \perp, \perp)$ to $\mathcal{F}_{\text{CGKA,2-5}}$. As we show in Proposition 4.4.7 below, such for any such pair, the value of rt' is unique. If $\mathsf{Node}[c_0] = \perp$ and no such w_0 exists, then $\mathcal{S}_{\text{2-5}}$ retrieves the associating long-term public key svk_c of id_c (which is guaranteed to exist when process succeeds in the real protocol) and returns $(ack := \texttt{true}, \perp, \mathsf{orig}' := \mathsf{id}_c, \mathsf{svk}' := \mathsf{svk}_c)$. Finally, if $\mathsf{Node}[c_0] \neq \perp$, then $\mathcal{S}_{\text{2-5}}$ simply returns $(ack := \texttt{true}, \perp, \perp, \perp)$. `*fill-prop` queries from $\mathcal{F}_{\text{CGKA,2-5}}$ to $\mathcal{S}_{\text{2-5}}$ are answered exactly as in commit queries described above.

(5) Join query from id**.** This concerns the case when $\mathcal{Z}$ queries $(\texttt{Join}, w_0, \widehat{w})$ to $\mathcal{F}_{\text{CGKA,2-5}}$. If $\mathsf{Ptr}[\mathsf{id}] = \perp$, then $\mathcal{F}_{\text{CGKA,2-5}}$ outputs $(\texttt{Join}, \mathsf{id}, w_0, \widehat{w})$ to $\mathcal{S}_{\text{2-5}}$. $\mathcal{S}_{\text{2-5}}$ then (deterministically) runs the simulated party id on input $(\texttt{Join}, w_0, \widehat{w})$. If party id returns $\perp$, then $\mathcal{S}_{\text{2-5}}$ returns $(ack := \texttt{false}, \perp, \perp, \perp)$ to $\mathcal{F}_{\text{CGKA,2-5}}$. Otherwise, if party id returns $(\mathsf{id}_c, \mathsf{mem})$, where mem is a list of $(\mathsf{id}, \mathsf{svk})$-tuples, then $\mathcal{S}_{\text{2-5}}$ checks if $\mathsf{Wel}[w_0] \neq \perp$. If so, $\mathcal{S}_{\text{2-5}}$ returns $(ack := \texttt{true}, \perp, \perp, \perp)$ to $\mathcal{F}_{\text{CGKA,2-5}}$. Otherwise, it checks if there exists a non-root c_0 such that $\mathsf{Node}[c_0] \neq \perp$ and c_0 includes the same confTag as the one included in w_0. Due to the modification we made in Hybrid 2-2 [*Unique* confTag *in* L_{com}] and by how $\mathcal{S}_{\text{2-5}}$ simulates the commit and process query (see above **(4)** and **(5)**), such c_0 is unique if it exists. Now, if such c_0 exists, then $\mathcal{S}_{\text{2-5}}$ returns $(ack := \texttt{true}, c_0' := c_0, \perp, \perp)$. Otherwise, if no such c_0 exists, then $\mathcal{S}_{\text{2-5}}$ further checks if there exists w_0' such that $\mathsf{Wel}[w_0'] \neq \perp$ that includes the same confTag as the one included in w_0. If so, $\mathcal{S}_{\text{2-5}}$ chooses any such w_0' and returns $(ack := \texttt{true}, c_0' := \mathsf{Wel}[w_0'], \perp, \perp)$. As we show in Proposition 4.4.7 below, the value of $\mathsf{Wel}[w_0']$ (which can either be a non-root or a detached root) is the same for all such w_0'. Finally, if no such

[14] As in prior works, note that $\mathcal{A}$ is only allowed to control the output of party id via randomness corruption. This is to capture *post-compromise security* in a meaningful way. We impose the same restriction when id is invoked on a commit query. See [Alw+20b; AJM22] for more details.

c_0 or w_0' exist, then $\mathcal{S}_{2\text{-}5}$ returns $(ack := \texttt{true}, \bot, \mathsf{orig}' := \mathsf{id}_c, \mathsf{mem}' := \mathsf{mem})$. This corresponds to the case $\mathsf{Wel}[w_0]$ is initialized by root_{rt} for a new $rt \in \mathbb{N}$.

(6) Key query from id**.** This concerns the case when $\mathcal{Z}$ queries Key to $\mathcal{F}_{\text{CGKA,2-5}}$. If $\mathsf{Ptr}[\mathsf{id}] = c_0 \neq \bot$, $\mathsf{HasKey}[\mathsf{id}] = \texttt{true}$, and $\mathsf{Node}[\mathsf{Ptr}[\mathsf{id}]].\mathsf{key} = \bot$, then $\mathcal{F}_{\text{CGKA,2-5}}$ outputs $(\texttt{Key}, \mathsf{id})$ to $\mathcal{S}_{2\text{-}5}$. (Recall that **safe** is always set to `false` in this hybrid.) If $\mathsf{HasKey}[\mathsf{id}] = \texttt{true}$, $\mathcal{S}_{2\text{-}5}$ must have invoked party id on input either a valid $(c_0, \widehat{c}, \vec{p})$ corresponding to a process query or $(w_0, \widehat{w})$ corresponding to a join query. In either case, the simulated party id is guaranteed to have computed a valid appSecret which is stored in its protocol state G (i.e., $G.\mathsf{appSecret}$). Thus, $\mathcal{S}_{2\text{-}5}$ returns $\mathsf{key} := G.\mathsf{appSecret}$ to $\mathcal{F}_{\text{CGKA,2-5}}$.

With the simulator $\mathcal{S}_{2\text{-}5}$ formally defined, we are now ready to prove the following lemma.

Lemma 4.4.6. *Hybrid 2-4 and Hybrid 2-5 are identical.*

Proof. *Part 2. Supporting Propositions.* Directly proving that the simulation provided by $\mathcal{S}_{2\text{-}5}$ creates an identical view to $\mathcal{Z}$ as in the previous hybrid is quite complex and possibly unreadable. To this end, we provide several supporting propositions that check the consistency within and between the history graph and protocol states maintained by $\mathcal{S}_{2\text{-}5}$ (and the ideal functionality $\mathcal{F}_{\text{CGKA,2-5}}$). The interested readers may first skim through Part 3 to check how the supporting propositions are used. Looking ahead, we are able to prove that `cons-invariant` in Figure 4.14 as a simple corollary of the propositions we prove in *Part 2*.
Part 2-1. Basic checks within/between history graphs and protocol states. The following shows that informally, if two w_0 and w_0' share the same confTag, then their corresponding nodes $\mathsf{Wel}[w_0]$ and $\mathsf{Wel}[w_0']$ must be assigned to the same non-root or a detached root. This shows that the confTag included in the welcome message w_0 commits the added users to be in the same group state (or equivalently to the same node in the history graph). Roughly, if this does not hold, then the environment can distinguish between the previous hybrid by causing an inconsistency in the history graph between parties that should belong to the same node $\mathsf{Node}[c_0]$.

Proposition 4.4.7 (Uniqueness of confTag **in welcome message).** *If two distinct w_0 and w_0' that include the same* confTag *satisfy $\mathsf{Wel}[w_0] \neq \bot$ and $\mathsf{Wel}[w_0'] \neq \bot$, then we must have $\mathsf{Wel}[w_0] = \mathsf{Wel}[w_0']$.*

Proof. Let us prove by contradiction. Assume we have two distinct w_0 and w_0' that include the same confTag but either of the following four cases hold:

1. $(\mathsf{Wel}[w_0], \mathsf{Wel}[w_0']) = (\mathsf{root}_{rt}, \mathsf{root}_{rt'})$ for some distinct rt and $rt' \in \mathbb{N}$;
2. $(\mathsf{Wel}[w_0], \mathsf{Wel}[w_0']) = (c_0, \mathsf{root}_{rt'})$ for some $rt' \in \mathbb{N}$ and non-root c_0;
3. $(\mathsf{Wel}[w_0], \mathsf{Wel}[w_0']) = (\mathsf{root}_{rt}, c_0')$ for some $rt \in \mathbb{N}$ and non-root c_0';
4. $(\mathsf{Wel}[w_0], \mathsf{Wel}[w_0']) = (c_0, c_0')$ for some distinct non-roots c_0 and c_0'.

Note that by construction $\mathsf{Wel}[w_0]$ or $\mathsf{Wel}[w_0']$ is never attached to the main root root_0 so we can safely discard this case. Below we assume $\mathsf{Wel}[w_0']$ is set before $\mathsf{Wel}[w_0]$ and further assume that all other w_0'' with the same confTag as w_0' satisfies $\mathsf{Wel}[w_0'] = \mathsf{Wel}[w_0'']$ and are set *after* $\mathsf{Wel}[w_0']$ is set. Namely, we assume without loss of generality that $\mathsf{Wel}[w_0']$ is the first to be created and w_0 to be the first welcome message that forms the contradiction.
Case (1) and *(2)*: $\mathsf{Wel}[w_0'] = \mathsf{root}_{rt'}$. Observe $\mathsf{Wel}[w_0']$ is only set to a detached root $\mathsf{root}_{rt'}$ during a join query. Moreover, by how $\mathcal{S}_{2\text{-}5}$ simulates the join query, at the point when $\mathsf{Wel}[w_0']$ is set, there does not exist a

non-root c_0 such that $\mathsf{Node}[c_0] \neq \bot$ and c_0 includes the same $\mathsf{confTag}$ as w'_0. Below we consider the timing that $\mathsf{Wel}[w_0]$ is set, which can be either during a commit or a join query.

Let us consider the former case. Notice $\mathsf{Wel}[w_0]$ cannot be set to a detached root during a commit query due to the `*attach` function in the ideal commit protocol. Hence, since *Case (1)* cannot occur, we only consider *Case (2)*, that is, $\mathsf{Wel}[w_0] = c_0$. We have two cases, $\mathsf{Node}[c_0] = \bot$ or not right before $\mathsf{Wel}[w_0] = c_0$ is set. In the former case, due to how $\mathcal{S}_{2\text{-}5}$ simulates the commit query, $\mathsf{Wel}[w_0]$ and $\mathsf{Wel}[w'_0]$ are assigned to the same node c_0. Hence, *Case (2)* cannot occur. In the latter case, if $\mathsf{Node}[c_0]$ was already set, then due to the modification we made in Hybrid 2-2 [*Unique* $\mathsf{confTag}$ *in* L_{com}], $\mathsf{Node}[c_0]$ must have been set during a process query. However, due to `*attach` function in the ideal process protocol, if this happends, then $\mathsf{Wel}[w'_0]$ will be reattached to c_0. Hence, *Case (2)* cannot occur either. Summarizing so far, $\mathsf{Wel}[w_0]$ cannot be set during a commit query.

Let us consider the latter case. We first consider *Case (1)*, where $\mathsf{Wel}[w_0]$ is set to root_{rt}. By observing how $\mathcal{S}_{2\text{-}5}$ simulates the join query and by our assumption, $\mathsf{Wel}[w_0]$ must be set to $\mathsf{Wel}[w'_0]$. Hence, *Case (1)* cannot occur. Next, consider *Case (2)*, where $\mathsf{Wel}[w_0]$ is set to c_0. By how $\mathcal{S}_{2\text{-}5}$ simulates the join query, c_0 must contain the same $\mathsf{confTag}$ as w'_0 and satisfy $\mathsf{Node}[c_0] \neq \bot$. However, considering that $\mathsf{Node}[c_0]$ is set only during a commit or a process query, it is clear that `*attach` function in the ideal commit or process protocols assigns $\mathsf{Wel}[w'_0]$ to c_0, thus contradicting $\mathsf{Wel}[w'_0] = \mathsf{root}_{rt'}$. Hence, *Case (2)* cannot occur either.

Case (3). Observe $\mathsf{Wel}[w_0]$ is only set to a detached root root_{rt} during a join query. Due to how $\mathcal{S}_{2\text{-}5}$ simulates the join query and considering that $\mathsf{Wel}[w_0]$ is not set to a non-root, we must have $\mathsf{Wel}[w_0] = \mathsf{Wel}[w'_0]$. However, this is a contradiction. Hence, *Case (3)* cannot occur.

Case (4). Due to how $\mathcal{S}_{2\text{-}5}$ simulates the commit, process, and join queries and by the definition of the `*attach` function in the ideal commit or process protocols, the $\mathsf{confTag}$ included in w_0, c_0, w'_0, c'_0 are identical, where we also use the fact that w_0 and w'_0 include the same $\mathsf{confTag}$. Moreover, due to the modification we made in Hybrid 2-2 [*Unique* $\mathsf{confTag}$ *in* L_{com}], we must have $c_0 = c'_0$ if they include the same $\mathsf{confTag}$ (and if $\mathcal{S}_{2\text{-}5}$ does not abort). However, this is a contradiction. Hence, *Case (4)* cannot occur.

This completes the proof. □

Remark 4.4.8 (Different welcome messages for the same group). Ideally, we might want a commit message c_0 to be uniquely bound to a single welcome message w_0 (i.e., if $\mathsf{Wel}[w_0] \neq \bot$, then no other w'_0 with the same $\mathsf{confTag}$ satisfies $\mathsf{Wel}[w'_0] \neq \bot$). However, due to the following concrete attack, Proposition 4.4.7 is the best we can hope for. Namely, users can be added to the same group by different welcome messages. However, Proposition 4.4.7 does guarantee that any different welcome messages provide a consistent view of the group to the invited users (i.e., $\mathsf{Wel}[w_0] = \mathsf{Wel}[w'_0]$).

1. The adversary $\mathcal{A}$ corrupts party id to obtain all secret information and state. It then runs id "in the head" using randomness rand it generated to obtain (c'_0, w'_0).
2. $\mathcal{A}$ modifies the signature sig' attached to w'_0 and creates another valid signature sig' on the same message, and creates a modified but valid welcome message w'_0. (Note that unforgeability does not say anything when the secret signing key ssk is leaked).
3. $\mathcal{A}$ queries $(\texttt{Join}, w_0, *)$ on some valid party id'. This sets $\mathsf{Wel}[w_0] = \mathsf{root}_{rt}$ for some $rt \in \mathbb{N}$.
4. $\mathcal{A}$ queries $(\texttt{Process}, c'_0, *)$ on some valid party id''. This sets attaches $\mathsf{Wel}[w_0]$ to c'_0. That is, we now have $\mathsf{Node}[c'_0] \neq \bot$ and $\mathsf{Wel}[w_0] = c'_0$.
5. Finally, $\mathcal{A}$ queries party id by setting $\mathsf{Rand}[\mathsf{id}] =$ 'bad' and using randomness rand. Since $\mathsf{Node}[c'_0] \neq \bot$, $\mathsf{Wel}[w'_0]$ is newly created and set to c'_0. Namely, we now have $\mathsf{Wel}[w'_0] = c'_0$.

Next, we provide a proposition that informally states that if some party id used w_0 to join a group and $\mathsf{Wel}[w_0]$ is assigned to a detached root, then a $\mathsf{Node}[c_0]$ with the same confTag as w_0 cannot yet exist in the history graph. In other words, if such $\mathsf{Node}[c_0]$ exists, then any w_0 with the same confTag should be assigned to c_0, i.e., $\mathsf{Wel}[w_0] = c_0$.

Proposition 4.4.9 (Consistency of confTag in commit and welcome messages). *If* $\mathsf{Wel}[w_0] = \mathsf{root}_{rt}$ *for some* $rt \in \mathbb{N}$*, then any non-root* c_0 *such that* $\mathsf{Node}[c_0] \neq \bot$ *does not include the same* confTag *as* w_0*.*

Proof. The statement can be equally stated as, if $\mathsf{Node}[c_0] \neq \bot$, then any w_0 such that $\mathsf{Wel}[w_0] = \mathsf{root}_{rt}$ for some $rt \in \mathbb{N}$ does not include the same confTag as c_0. Observe that the only place $\mathsf{Node}[c_0]$ for a non-root c_0 is set is either during a commit or process query. We provide the proof considering these two cases individually.

First, assume $\mathsf{Node}[c_0]$ for a non-root c_0 is set during a commit query. This means that some party id was invoked by $\mathcal{S}_{2\text{-}5}$ and output some $(c_0, \vec{c}, w_0', \vec{w}')$. There are two cases to consider: $w_0' = \bot$ or $w_0' \neq \bot$. In the former case, due to the modification we made in Hybrid 2-5 [*Consistency of no-join*], there cannot exist any w_0 that includes the same confTag as c_0 but $\mathsf{Wel}[w_0] \neq \bot$. Therefore, the statement holds as desired. In the latter case, we consider two more cases: $\mathsf{Wel}[w_0] = \mathsf{root}_{rt}$ was set before or after $\mathcal{S}_{2\text{-}5}$ invoked party id. If $\mathsf{Wel}[w_0] = \mathsf{root}_{rt}$ was set before $\mathcal{S}_{2\text{-}5}$ invoked party id, then due to how $\mathcal{S}_{2\text{-}5}$ simulates the commit query, any $\mathsf{Wel}[w_0]$ attached to a detached root must have been reattached to c_0 via the `*attach` function in the ideal commit protocol. Namely, there will not exist a $\mathsf{Wel}[w_0]$ that is attached to a detached root so the statement holds as desired. On the other hand, $\mathsf{Wel}[w_0]$ cannot be set to a detached root after $\mathcal{S}_{2\text{-}5}$ invoked party id either. This is because due to how $\mathcal{S}_{2\text{-}5}$ simulates the join query (which is the only place $\mathsf{Wel}[w_0]$ can potentially be set to a detached root), if w_0 and c_0 include the same confTag, then $\mathsf{Wel}[w_0]$ is attached to c_0. Therefore, the statement holds in case $\mathsf{Node}[c_0]$ is set during a commit query.

Next, assume $\mathsf{Node}[c_0]$ for a non-root c_0 is set during a process query. This means that some party id was invoked by $\mathcal{S}_{2\text{-}5}$ on input $(\texttt{Process}, c_0, \widehat{c}, \vec{p})$. Again, we consider two cases: $\mathsf{Wel}[w_0] = \mathsf{root}_{rt}$ was set before or after $\mathcal{S}_{2\text{-}5}$ invoked party id. If $\mathsf{Wel}[w_0] = \mathsf{root}_{rt}$ was already set before $\mathcal{S}_{2\text{-}5}$ invoked party id, then due to how $\mathcal{S}_{2\text{-}5}$ simulates the process query, any $\mathsf{Wel}[w_0]$ attached to a detached root must have been reattached to c_0 via the `*attach` function in the ideal process protocol. Hence, the statement holds as desired. The other case when $\mathsf{Wel}[w_0]$ was not set to a detached root before $\mathcal{S}_{2\text{-}5}$ invoked party id is also identical to the above case. Therefore, the statement holds in case $\mathsf{Node}[c_0]$ is set during a process query as well.

This concludes the proof. □

The following provides some useful equivalence relationships between the protocol states and nodes maintained by the history graph. Most of the relations are a simple consequence of the correctness of the real protocol and we provide them mainly for reference. Note that some relationships are not included in the following since we either do not require them or because we need to prove them. Specifically, *Case C* is missing many desirable consistency relation checks such as $\mathsf{Node}[c_0].\mathsf{orig} = \mathsf{id}_c$ and $\mathsf{Node}[c_0].\mathsf{mem} = G.\mathsf{memberIDsvks}()$. These relations are not simple consequence of the real protocol and will be handled separately below.

Fact 1 (Existence of id in history graph). Let $G \neq \bot$ and G^{prev} (possibly $\bot$) be the current and previous protocol states[15] of party id that is internally simulated by $\mathcal{S}_{2\text{-}5}$, respectively. Then, if $\mathsf{Ptr}[\mathsf{id}] = c_0$, then one of the following three cases hold:

Case A: [c_0 is the main root root_0]

[15]We assume the state is incremented (i.e., move from G^{prev} to G) when processing either a commitment or a welcome message. Therefore, even though the state is updated after a commit in a strict sense, we view them as the same "current" state for simplicity.

- $\mathsf{id} = \mathsf{id}_{\mathsf{creator}}$;
- $G^{\mathsf{prev}} = \perp$;
- $\mathsf{Node}[\mathsf{root}_0].\mathsf{orig} = \mathsf{id}_{\mathsf{creator}}$;
- $\mathsf{Node}[\mathsf{root}_0].\mathsf{par} = \perp$;
- $\mathsf{Node}[\mathsf{root}_0].\mathsf{prop} = \perp$;
- $\mathsf{Node}[\mathsf{root}_0].\mathsf{mem} = G.\mathsf{memberIDsvks}()$
- $G.\mathsf{epoch} = 0$;
- $G.\mathsf{confTransHash\text{-}w.o\text{-}`id_c'} = \perp$;
- $G.\mathsf{confTransHash} = \perp$;
- $G.\mathsf{groupCont}() = (G.\mathsf{gid}, G.\mathsf{epoch}, G.\mathsf{memberHash}, G.\mathsf{confTransHash})$;
- $G.\mathsf{confKey} = \mathsf{H}(G.\mathsf{joinerSecret}, G.\mathsf{groupCont}(), \text{`conf'})$;
- $G.\mathsf{membKey} = \mathsf{H}(G.\mathsf{joinerSecret}, G.\mathsf{groupCont}(), \text{`memb'})$;
- $G.\mathsf{appSecret} = \mathsf{H}(G.\mathsf{joinerSecret}, G.\mathsf{groupCont}(), \text{`app'})$;
- $G.\mathsf{interimTransHash} = \mathsf{H}(G.\mathsf{confTransHash}, \mathsf{confTag})$.

Case B: [c_0 is a detached root (i.e., $c_0 = \mathsf{root}_{rt}$ for some $rt \in \mathbb{N}$)] There exists a w_0 of the form $((\mathsf{gid}, \mathsf{epoch}, \mathsf{memberPublicInfo}, \mathsf{memberHash}, \mathsf{confTransHash\text{-}w.o\text{-}`id_c'}, \mathsf{confTransHash}, \mathsf{interimTransHash}, \mathsf{confTag}, \mathsf{id}_c), \mathsf{T}, \mathsf{sig})$ such that

- $\mathsf{Wel}[w_0] = \mathsf{root}_{rt}$;
- $G^{\mathsf{prev}} = \perp$;
- $\mathsf{Node}[\mathsf{root}_{rt}].\mathsf{orig} = \mathsf{id}_c$;
- $\mathsf{Node}[\mathsf{root}_{rt}].\mathsf{par} = \perp$;
- $\mathsf{Node}[\mathsf{root}_{rt}].\mathsf{prop} = \perp$;
- $\mathsf{Node}[\mathsf{root}_{rt}].\mathsf{mem} = G.\mathsf{memberIDsvks}()$;
- $G.\mathsf{gid} = \mathsf{gid}$;
- $G.\mathsf{epoch} = \mathsf{epoch}$;
- $G.\mathsf{memberPublicInfo}() = \mathsf{memberPublicInfo}$;
- $G.\mathsf{memberHash} = \texttt{*derive-member-hash}(G)$;
- $G.\mathsf{memberHash} = \mathsf{memberHash}$;
- $G.\mathsf{confTransHash\text{-}w.o\text{-}`id_c'} = \mathsf{confTransHash\text{-}w.o\text{-}`id_c'}$;

- G.confTransHash = confTransHash;
- G.confTransHash = $\mathsf{H}(G.\text{confTransHash-w.o-`id}_c\text{'}, \mathsf{id}_c)$;
- G.interimTransHash = interimTransHash;
- G.groupCont() = $(G.\mathsf{gid}, G.\mathsf{epoch}, G.\mathsf{memberHash}, G.\mathsf{confTransHash})$;
- G.confKey = $\mathsf{H}(G.\mathsf{joinerSecret}, G.\mathsf{groupCont}(), \text{`conf'})$;
- G.membKey = $\mathsf{H}(G.\mathsf{joinerSecret}, G.\mathsf{groupCont}(), \text{`memb'})$;
- G.appSecret = $\mathsf{H}(G.\mathsf{joinerSecret}, G.\mathsf{groupCont}(), \text{`app'})$;
- confTag = $\mathsf{MAC.Gen}(G.\mathsf{confKey}, G.\mathsf{confTransHash})$;
- G.confTag = confTag;
- G.interimTransHash = $\mathsf{H}(G.\mathsf{confTransHash}, \mathsf{confTag})$.

Moreover, all such w_0 agrees on every entry expect for $(\mathsf{T}, \mathsf{sig})$. In particular, all such w_0 include the same confTag.

Case C: [c_0 is a non-root (i.e., $c_0 = (\mathsf{gid}, \mathsf{epoch}, \mathsf{id}_c, \text{`commit'}, C_0 = (\mathsf{propIDs}, \mathsf{kp}, \mathsf{T}), \mathsf{sig}, \mathsf{confTag})$)]

- for all $p \in \vec{p} = \mathsf{Node}[c_0].\mathsf{prop}$, p is of the form $(\mathsf{gid}, \mathsf{epoch}, \mathsf{id}_s, \text{`proposal'}, P, \mathsf{sig}', \mathsf{membTag})$ and satisfies
  - G.gid = gid;
  - G^{prev}.epoch = epoch;
  - membTag = $\mathsf{MAC.Gen}(G^{\mathsf{prev}}.\mathsf{membKey}, (G^{\mathsf{prev}}.\mathsf{groupCont}(), \mathsf{id}_s, \text{`proposal'}, P, \mathsf{sig}'))$;
  - G.membTags = $(\mathsf{membTag})_{\mathsf{membTag\ included\ in}\ p \in \vec{p}}$;
  - propIDs = $(\mathsf{H}(p))_{p \in \vec{p}}$.
- G.gid = G^{prev}.gid;
- G.epoch = G^{prev}.epoch + 1 = epoch + 1;
- G.memberHash = `*derive-member-hash`(G);
- G.confTransHash-w.o-`id$_c$' = $\mathsf{H}(G^{\mathsf{prev}}.\mathsf{interimTransHash}, (\mathsf{gid}, \mathsf{epoch}, \text{`commit'}, C_0, \mathsf{sig}))$;
- G.confTransHash = $\mathsf{H}(G.\text{confTransHash-w.o-`id}_c\text{'}, \mathsf{id}_c)$;
- G.joinerSecret = $\mathsf{H}(G^{\mathsf{prev}}.\mathsf{initSecret}, G.\mathsf{comSecret})$;
- G.groupCont() = $(G.\mathsf{gid}, G.\mathsf{epoch}, G.\mathsf{memberHash}, G.\mathsf{confTransHash})$;
- G.confKey = $\mathsf{H}(G.\mathsf{joinerSecret}, G.\mathsf{groupCont}(), \text{`conf'})$;
- G.membKey = $\mathsf{H}(G.\mathsf{joinerSecret}, G.\mathsf{groupCont}(), \text{`memb'})$;
- G.appSecret = $\mathsf{H}(G.\mathsf{joinerSecret}, G.\mathsf{groupCont}(), \text{`app'})$;
- confTag = $\mathsf{MAC.Gen}(G.\mathsf{confKey}, G.\mathsf{confTransHash})$;

- $G.\mathsf{confTag} = \mathsf{confTag}$;
- $G.\mathsf{interimTransHash} = \mathsf{H}(G.\mathsf{confTransHash}, \mathsf{confTag})$.

Proof. All the relations in *Case A* and *Case C* are a consequence of the correctness of the real protocol. For the latter case, observe that $\mathsf{Node}[c_0].\mathsf{prop}$ is only set during a commit or a process query. All the relations in *Case B* that do not concern $\mathsf{Node}[*]$ is also a consequence of the correctness of the real protocol.

We check the remaining conditions for *Case B*. First, observe that the only place $\mathsf{Node}[\mathsf{root}_{rt}]$ for some $rt \in \mathbb{N}$ is set is during a join query when $\mathcal{S}_{2\text{-}5}$ returns $(ack := \mathtt{true}, \bot, \mathsf{orig}' := \mathsf{id}_c, \mathsf{mem}' := \mathsf{mem})$ to $\mathcal{F}_{\mathrm{CGKA},2\text{-}5}$. Here, id_c is those included in w_0. Then by the `*create-root` function in the ideal join protocol, we have $\mathsf{Node}[\mathsf{root}_{rt}] = \mathsf{id}_c$ as desired. Moreover, observing that every entry expect for $(\mathsf{T}, \mathsf{sig})$ in the welcome message w_0 is used to derive $\mathsf{interimTransHash}$, the uniqueness of the remaining entries are guaranteed due to the modification we made in Hybrid 2-1 [*No collision in RO*]. □

The following is the main proposition of *Part 2-1*. It shows that two parties are assigned to the same node in the history graph if and only if they agree on the same group secrets. This allows us to relate the consistency of history graph and protocol states.

Proposition 4.4.10 (Consistency of protocol secrets and history graph). *Let* id *and* id' *be two parties such that* $\mathsf{Ptr}[\mathsf{id}] \neq \bot$ *and* $\mathsf{Ptr}[\mathsf{id}'] \neq \bot$*, and let* G_{id} *and* $G_{\mathsf{id}'}$ *be their protocol states. Here,* id *and* id' *may be the same party from different epochs. Then, we have* $\mathsf{Ptr}[\mathsf{id}] = \mathsf{Ptr}[\mathsf{id}']$ *if and only if either one of the following conditions hold:*

- $G_{\mathsf{id}}.\mathsf{confKey} = G_{\mathsf{id}'}.\mathsf{confKey}$;
- $G_{\mathsf{id}}.\mathsf{membKey} = G_{\mathsf{id}'}.\mathsf{membKey}$;
- $G_{\mathsf{id}}.\mathsf{appSecret} = G_{\mathsf{id}'}.\mathsf{appSecret}$.

Proof. Let us first show the "if" direction of the statement. We only show the case $G_{\mathsf{id}}.\mathsf{confKey} = G_{\mathsf{id}'}.\mathsf{confKey}$ as the other cases can be proven identically. The proof heavily relies on the equality relations provided in Fact 1. First, since we can assume there is no collision in the hash function H due to the modification we made in Hybrid 2-1 [*No collision in RO*], we have $G_{\mathsf{id}}.\mathsf{confTransHash} = G_{\mathsf{id}'}.\mathsf{confTransHash}$ (which are included in $\mathsf{groupCont}()$). Then, this implies that $G_{\mathsf{id}}.\mathsf{confTag} = G_{\mathsf{id}'}.\mathsf{confTag}$. In case $\mathsf{Ptr}[\mathsf{id}]$ and $\mathsf{Ptr}[\mathsf{id}']$ are both non-roots, then this implies that $\mathsf{Ptr}[\mathsf{id}]$ and $\mathsf{Ptr}[\mathsf{id}']$ both include the same $\mathsf{confTag}$. Hence, by the modification we made in Hybrid 2-2 [*Unique* $\mathsf{confTag}$ *in* L_{com} *and* L_{wel}], we have $\mathsf{Ptr}[\mathsf{id}] = \mathsf{Ptr}[\mathsf{id}']$.

Now, let us consider the case $\mathsf{Ptr}[\mathsf{id}] = \mathsf{root}_{rt}$ for some $rt \in \mathbb{N}$. Then, since $\mathsf{Ptr}[\mathsf{id}]$ is assigned to a detached root only during a join query, there must exist w_0 such that $\mathsf{Wel}[w_0] = \mathsf{root}_{rt}$, where w_0 includes $G_{\mathsf{id}}.\mathsf{confTag}$ by the correctness of the protocol. Due to Proposition 4.4.9, there does not exist a non-root c_0 such that $\mathsf{Node}[c_0] \neq \bot$ but c_0 includes $G_{\mathsf{id}}.\mathsf{confTag} = G_{\mathsf{id}'}.\mathsf{confTag}$. This implies that we must have $\mathsf{Ptr}[\mathsf{id}'] = \mathsf{root}_{rt'}$ for some $rt' \in \mathbb{N}$. Then, by Proposition 4.4.7, we have $rt = rt'$ since there cannot exist two w_0 and w_0' such that $\mathsf{Wel}[w_0] = \mathsf{root}_{rt}$, $\mathsf{Wel}[w_0'] = \mathsf{root}_{rt'}$, and $rt \neq rt'$ that include the same $\mathsf{confTag}$. Therefore, we also have $\mathsf{Ptr}[\mathsf{id}] = \mathsf{Ptr}[\mathsf{id}']$ when they are assigned to detached roots.

It remains to show the "only if" direction of the statement. Again, we only show the case $G_{\mathsf{id}}.\mathsf{confKey} = G_{\mathsf{id}'}.\mathsf{confKey}$ as the other cases can be proven identically. Assume $\mathsf{Ptr}[\mathsf{id}] = \mathsf{Ptr}[\mathsf{id}']$. In case $\mathsf{Ptr}[\mathsf{id}] = \mathsf{root}_0$, then $\mathsf{id} = \mathsf{id}' = \mathsf{id}_{\mathsf{creator}}$. Therefore, the statement holds trivially. The case $\mathsf{Ptr}[\mathsf{id}] = c_0$ for some non-root c_0 holds as a direct consequence of Fact 1. Finally, in case $\mathsf{Ptr}[\mathsf{id}] = \mathsf{root}_{rt}$ for some $rt \in \mathbb{N}$, then by Fact 1, any w_0 that satisfy the relations provide in *Case B* produce the same $\mathsf{confTag}$. Then due to the modification we made in Hybrid 2-2 [*Unique* $\mathsf{confTag}$ *in* L_{wel}], we must have $G_{\mathsf{id}}.\mathsf{confKey} = G_{\mathsf{id}'}.\mathsf{confKey}$. The case $\mathsf{Ptr}[\mathsf{id}] = c_0$ for some non-root c_0 can be checked similarly to the case $\mathsf{Ptr}[\mathsf{id}] = \mathsf{root}_{rt}$.

This completes the proof. □

Remark 4.4.11 (Implication of $\mathsf{Ptr}[\mathsf{id}] = \mathsf{Ptr}[\mathsf{id}']$). Proposition 4.4.10 only focuses on the group secrets since we wanted an "if and only if" statement. However, if we only cared about the "only if" direction, there is much more we can deduce from $\mathsf{Ptr}[\mathsf{id}] = \mathsf{Ptr}[\mathsf{id}']$. Namely, following the "only if" direction of the proof of Proposition 4.4.10 and the proof of Lemma 4.4.4 to move from Hybrid 2-1 to 2-2, we can conclude that if two parties id and id′ satisfy $\mathsf{Ptr}[\mathsf{id}] = \mathsf{Ptr}[\mathsf{id}']$, then they agree on the same view of the group such as $G_{\mathsf{id}}.\mathsf{gid} = G_{\mathsf{id}'}.\mathsf{gid}$ and $G_{\mathsf{id}}.\mathsf{memberIDsvks}() = G_{\mathsf{id}'}.\mathsf{memberIDsvks}()$ as expected. We use this implication in Proposition 4.4.14.

Part 2-2. Consistency of proposal messages. The following proposition establishes that if a party outputs or receives a proposal p that already exists in the history graph (i.e., $\mathsf{Prop}[p] \neq \bot$), then it satisfies all the intuitive consistency checks.

Proposition 4.4.12 (Consistency of existing proposal node). *Assume party* id *such that* $\mathsf{Ptr}[\mathsf{id}] \neq \bot$ *outputs* p *on input* $(\texttt{Propose}, \mathsf{act})$ *from* $\mathcal{S}_{2\text{-}5}$*, where* p *is of the form* $(\mathsf{gid}, \mathsf{epoch}, \mathsf{id}, \text{'proposal'}, P, \mathsf{sig}, \mathsf{membTag})$*. Then, if* $\mathsf{Prop}[p] \neq \bot$*, we have the following (after* $\mathcal{S}_{2\text{-}5}$ *receives an output from* id *but before it provides input to* $\mathcal{F}_{CGKA,2\text{-}5}$*):*

- $\mathsf{Prop}[p].\mathsf{par} = \mathsf{Ptr}[\mathsf{id}]$*;*
- $\mathsf{Prop}[p].\mathsf{orig} = \mathsf{id}$*;*
- $\mathsf{Prop}[p].\mathsf{act} =$ 'upd'-svk *if* $P = (\text{'upd'}, \mathsf{kp})$*;* $\mathsf{Prop}[p].\mathsf{act} =$ 'add'-id_t-svk_t *if* $P = (\text{'add'}, \mathsf{kp}_t)$*; or* $\mathsf{Prop}[p].\mathsf{act} =$ 'rem'-id_t *if* $P = (\text{'rem'}, \mathsf{id}_t)$*, where* svk *and* svk_t *are included in* kp *and* kp_t*, respectively.*

Additionally, consider the following two cases:

- id *outputs* $(c_0, \vec{c}, w_0, \vec{w})$ *on input* $(\texttt{Commit}, \vec{p}, \mathsf{svk})$ *from* $\mathcal{S}_{2\text{-}5}$*; or*
- id *outputs* $(\mathsf{id}_c, upd \| rem \| add)$ *on input* $(\texttt{Process}, c_0, \widehat{c}, \vec{p})$ *from* $\mathcal{S}_{2\text{-}5}$*.*

For these two cases, we have the following (after $\mathcal{S}_{2\text{-}5}$ *receives an output from* id *but before it provides input to* $\mathcal{F}_{CGKA,2\text{-}5}$*):*

- *for all* $p \in \vec{p}$*,* p *is of the form* $(\mathsf{gid}, \mathsf{epoch}, \mathsf{id}_s, \text{'proposal'}, P, \mathsf{sig}, \mathsf{membTag})$ *and we have the following if* $\mathsf{Prop}[p] \neq \bot$*:*
  - $\mathsf{Prop}[p].\mathsf{par} = \mathsf{Ptr}[\mathsf{id}]$*;*
  - $\mathsf{Prop}[p].\mathsf{orig} = \mathsf{id}_s$*;*
  - $\mathsf{Prop}[p].\mathsf{act} =$ 'upd'-svk *if* $P = (\text{'upd'}, \mathsf{kp})$*;* $\mathsf{Prop}[p].\mathsf{act} =$ 'add'-id_t-svk_t *if* $P = (\text{'add'}, \mathsf{kp}_t)$*; or* $\mathsf{Prop}[p].\mathsf{act} =$ 'rem'-id_t *if* $P = (\text{'rem'}, \mathsf{id}_t)$*, where* svk *and* svk_t *are included in* kp *and* kp_t*, respectively.*

Proof. We first consider the former case where id executes a propose protocol. Let G_{id} be the protocol state of id. There are two places where $\mathsf{Prop}[p]$ can be set. One is during a propose query and the other is during `*fill-prop` which is invoked during a commit or process query. Assume $\mathsf{Prop}[p]$ is set during a propose query. Then, there exists some party id′ that was invoked by $\mathcal{S}_{2\text{-}5}$ for a propose query that output p. Since p includes id, we have $\mathsf{id}' = \mathsf{id}$ and the condition regarding $\mathsf{Prop}[p].\mathsf{act}$ holds by the correctness of the real propose protocol. Moreover, due to modification we made in Hybrid 2-2 [*Unique* membTag *in* L_{prop}], the protocol state G'_{id} of id when it outputs p the first time must satisfy $G_{\mathsf{id}}.\mathsf{membKey} = G'_{\mathsf{id}}.\mathsf{membKey}$. Then due to Proposition 4.4.10, we have $\mathsf{Prop}[p].\mathsf{par} = \mathsf{Ptr}[\mathsf{id}]$ as desired. On the other hand, assume $\mathsf{Prop}[p]$ is set during `*fill-prop` which is invoked during a commit or process query. Let the party being invoked be id′. By how $\mathcal{S}_{2\text{-}5}$ responds to `*fill-prop` it is clear that $\mathsf{Prop}[p].\mathsf{orig} = \mathsf{id}$ and the condition regarding $\mathsf{Prop}[p].\mathsf{act}$

hold. Moreover, due to modification we made in Hybrid 2-2 [*Unique* membTag *in* L_{prop}], the protocol state $G_{\mathsf{id}'}$ of party id' that accepts p must satisfy $G_{\mathsf{id}}.\mathsf{membKey} = G_{\mathsf{id}'}.\mathsf{membKey}$. Then due to Proposition 4.4.10, we have $\mathsf{Ptr}[\mathsf{id}] = \mathsf{Ptr}[\mathsf{id}']$. Therefore, by the definition of `*create-prop` run within `*fill-prop`, we have $\mathsf{Prop}[p].\mathsf{par} = \mathsf{Ptr}[\mathsf{id}]$ as desired.

The latter cases where id executes either a commit of process query consist of the same argument as above. Therefore, this completes the proof. □

Part 2-3. Consistency of commit messages. This is the final and most important part of the basic consistency checks. Unlike proposal messages, consistency of commit messages is proven by induction. Informally, this is because to conclude the current commit node is consistent, we must rely on the fact that the previous commit node is also consistent. We first show that if the current commit node id is assigned to agree with the group member as those included in the protocol state (i.e., $G.\mathsf{memberIDsvks}() = \mathsf{Node}[c_0].\mathsf{mem}$), then the next commit node and the updated protocol state agrees on the group member of the next epoch.

Proposition 4.4.13 (Consistency of commit and process protocol). *Assume party* id *and* c_0' *such that* $\mathsf{Ptr}[\mathsf{id}] = c_0'$ *and* $\mathsf{Node}[c_0'] \neq \bot$. *Let G be* id*'s protocol state and assume we have* $G.\mathsf{memberIDsvks}() = \mathsf{Node}[c_0'].\mathsf{mem}$. *Consider the following two cases:*

- id *outputs* $(c_0, \vec{c}, w_0, \vec{w})$ *on input* $(\texttt{Commit}, \vec{p}, \mathsf{svk})$ *from* $\mathcal{S}_{2\text{-}5}$*; or*
- id *outputs* $(\mathsf{id}_c, upd \| rem \| add)$ *on input* $(\texttt{Process}, c_0, \widehat{c}, \vec{p})$ *from* $\mathcal{S}_{2\text{-}5}$.

After receiving the output from id, $\mathcal{S}_{2\text{-}5}$ *continues the simulation by providing input to* $\mathcal{F}_{CGKA,2\text{-}5}$. *If the ideal commit or process protocols terminate without halting or outputting* $\bot$, *then we have* $G'.\mathsf{memberIDsvks}() = \mathsf{Node}[c_0].\mathsf{mem}$ *in both cases, where* G' *is the new group state included in* $G.\mathsf{pendCom}[c_0]$ *in case of a commit query or the updated group state in case of a process query.*

Proof. Let us first consider the case id is invoked on a commit query. Condition on the ideal functionality not halting or outputting $\bot$, we are guaranteed that the function `*next-members` on line 7 of the ideal commit protocol terminates as expected. In particular, since `*next-members` runs syntactically the same procedure as `*apply-props` on line 3 of the real commit protocol, we have $G'.\mathsf{memberIDsvks}() = \mathsf{mem}$ if $G.\mathsf{memberIDsvks}() = \mathsf{Node}[c_0'].\mathsf{mem}$, where mem is the outputs of `*next-members`. Now, if $\mathsf{Node}[c_0]$ is created via `*create-child` (i.e., $\mathsf{Node}[c_0] = \bot \wedge rt = \bot$), then we have $\mathsf{Node}[c_0].\mathsf{mem} = \mathsf{mem}$ as desired. Otherwise, if `*consistent-com` and `*attach` do not halt nor output $\bot$, we have $\mathsf{Node}[c_0].\mathsf{mem} = \mathsf{mem}$ as desired in case $\mathsf{Node}[c_0] \neq \bot$ or $\mathsf{Node}[c_0] = \bot \wedge rt \neq \bot$. This completes the proof in case of a commit query.

Let us now consider the case id is invoked on a process query. Following the same argument as above, in case $\mathsf{Node}[c_0]$ is created via `*create-child` (i.e., $\mathsf{Node}[c_0] = \bot \wedge rt = \bot$), we have $\mathsf{Node}[c_0].\mathsf{mem} = \mathsf{mem}$ as desired conditioned on the ideal functionality not halting or outputting $\bot$. Otherwise, if `*valid-successor` and `*attach` do not halt nor output $\bot$, we have $\mathsf{Node}[c_0].\mathsf{mem} = \mathsf{mem}$ as desired in case $\mathsf{Node}[c_0] \neq \bot$ or $\mathsf{Node}[c_0] = \bot \wedge rt \neq \bot$. This completes the proof in case of a join query. □

We next show that if a party id is assigned to some commit node in the history graph, then the group member stored on that commit node should be consistent with the members stored in the protocol state. The proof is by induction where the base case is guaranteed by Fact 1 and we use the previous Proposition 4.4.13 to move up the epoch. Specifically, any party is first assigned to a root in the beginning (*Case A* or *B* in Fact 1), and in this case, the commit node and protocol state are guaranteed to store the same group members.

Proposition 4.4.14 (Consistency of current commit node). *Assume party* id *and a non-root* c_0 *of the form* $(\mathsf{gid}, \mathsf{epoch}, \mathsf{id}_c, \text{'commit'}, C_0 = (\mathsf{propIDs}, \mathsf{kp}, \mathsf{T}), \mathsf{sig}, \mathsf{confTag})$ *such that* $\mathsf{Ptr}[\mathsf{id}] = c_0$ *and* $\mathsf{Node}[c_0] \neq \bot$. *Let* G *be the protocol state of* id. *Then we have the following:*

- $\mathsf{Node}[c_0].\mathsf{orig} = \mathsf{id}_c$;
- $\mathsf{Node}[c_0].\mathsf{mem} = G.\mathsf{memberIDsvks}()$.

Proof. We first prove the relation $\mathsf{Node}[c_0].\mathsf{orig} = \mathsf{id}_c$. Observe $\mathsf{Node}[c_0]$ is set either during a commit or a process query. Below, we only consider the case $\mathsf{Node}[c_0]$ is set during a commit query since the case for a process query is almost identical. There are further two cases to consider: $\mathsf{Node}[c_0]$ was initially $\bot$ and $\mathcal{S}_{2\text{-}5}$ outputs $rt = \bot$ or $\mathsf{Node}[c_0]$ was initially $\bot$ and $\mathcal{S}_{2\text{-}5}$ outputs $rt \in \mathbb{N}$. In the former case, due to the `*create-child` function in the ideal commit protocol, we have $\mathsf{Node}[c_0].\mathsf{orig} = \mathsf{id}_c$ as desired. In the latter case, there exists a detached root root_{rt} and a welcome message w_0 that includes the same $\mathsf{confTag}$ as c_0 such that $\mathsf{Wel}[w_0] = \mathsf{root}_{rt}$. First, by how $\mathcal{S}_{2\text{-}5}$ simulates the join query, we have $\mathsf{Node}[\mathsf{root}_{rt}].\mathsf{orig} = \mathsf{id}'_c$, where id'_c is included in $\mathsf{groupInfo}$ of the welcome message w_0. Next, due to the `*attach` function in the ideal commit protocol, we have $\mathsf{Node}[c_0].\mathsf{orig} = \mathsf{Node}[\mathsf{root}_{rt}].\mathsf{orig} = \mathsf{id}'_c$. Finally, due to Hybrid 2-2 [*Unique* $\mathsf{confTag}$ *in* L_{com} *and* L_{wel}], we have $\mathsf{id}'_c = \mathsf{id}_c$. Therefore, if $\mathsf{Node}[c_0]$ is set during a commit query, then we have $\mathsf{Node}[c_0].\mathsf{orig} = \mathsf{id}_c$ as desired.

So far we established the first part of the statement: $\mathsf{Node}[c_0].\mathsf{orig} = \mathsf{id}_c$. It remains to prove the second part of the statement: $\mathsf{Node}[c_0].\mathsf{mem} = G.\mathsf{memberIDsvks}()$. Below, we prove this by contradiction. Assume we have $\mathsf{Ptr}[\mathsf{id}] = c_0$ and $\mathsf{Node}[c_0] \neq \bot$ for a non-root c_0 but $\mathsf{Node}[c_0].\mathsf{mem} \neq G.\mathsf{memberIDsvks}()$. Observe that $\mathsf{Ptr}[\mathsf{id}]$ is assigned to a new value only during a process or a join query; $\mathsf{Ptr}[\mathsf{id}]$ remains the same during a process or a commit query. Let us consider the latter case where $\mathsf{Ptr}[\mathsf{id}] = c_0$ is assigned during join query, that is, id is invoked by $\mathcal{S}_{2\text{-}5}$ on input $(w_0, \widehat{w})$. We show that this case boils down to checking the former case. Since c_0 is a non-detached root by the assumption in the statement and by how $\mathcal{S}_{2\text{-}5}$ simulates the join query, $\mathsf{Node}[c_0] \neq \bot$ and c_0 includes the same $\mathsf{confTag}$ as w_0. Since $\mathsf{Node}[c_0]$ is set only during a commit or a process query, this implies that there is some party id' that outputs (resp. inputs) c_0 during a commit (resp. process) query. In case id' is invoked on a process query, then $\mathsf{Ptr}[\mathsf{id}'] = c_0$ due to the ideal process function. Then, due to Remark 4.4.11, if $G.\mathsf{memberIDsvks}() = G'.\mathsf{memberIDsvks}()$, where G and G' are the protocol states of id and id', respectively. Specifically, it suffices to check that $\mathsf{Node}[c_0].\mathsf{mem} \neq G'.\mathsf{memberIDsvks}()$ cannot happen during a process query. The case id' is invoked on a commit query is handled in the same way.

It remains to consider the former case where $\mathsf{Ptr}[\mathsf{id}] = c_0$ is assigned during process query. Then, taking the contrapositive of Proposition 4.4.13, since $\mathsf{Node}[c_0].\mathsf{mem} \neq G.\mathsf{memberIDsvks}()$, we must have $\mathsf{Node}[c'_0].\mathsf{mem} \neq G'.\mathsf{memberIDsvks}()$. We can iteratively apply this argument till we reach a point that $\mathsf{Ptr}[\mathsf{id}] = \mathsf{root}_{rt}$ for some $rt \in \{0\} \cup \mathbb{N}$. This is because any party is initially assigned to either a root or non-root via the join query (or to root_0 by default if $\mathsf{id} = \mathsf{id}_{\mathsf{creator}}$), and in case we arrive at a non-root, then by the above argument, we can focus on the commit or process query that generated the non-root and repeat the same argument till we reach a root. Finally, if G'' is the protocol state of id when $\mathsf{Ptr}[\mathsf{id}] = \mathsf{root}_{rt}$, then we have $\mathsf{Node}[\mathsf{root}_{rt}].\mathsf{mem} \neq G''.\mathsf{memberIDsvks}()$. However, by Fact 1, we must have $\mathsf{Node}[\mathsf{root}_{rt}].\mathsf{mem} = G''.\mathsf{memberIDsvks}()$. Therefore, this is a contradiction. This establishes the second part of the statement.

This completes the proof. □

Finally, the following proposition is an analog of Proposition 4.4.12 regarding the consistency check of existing proposal nodes. Specifically, the following establishes that if a party outputs or receives a commit c_0 that already exists in the history graph (i.e., $\mathsf{Node}[c_0] \neq \bot$), then it satisfies intuitive consistency checks.

Proposition 4.4.15 (Consistency of existing commit node). *Let party* id *satisfy* $\mathsf{Ptr}[\mathsf{id}] \neq \bot$ *and consider the following two cases:*

- id *outputs* $(c_0, \vec{c}, w_0, \vec{w})$ *on input* $(\texttt{Commit}, \vec{p}, \mathsf{svk})$ *from* $\mathcal{S}_{2\text{-}5}$*; or*
- id *outputs* $(\mathsf{id}_c, upd \| rem \| add)$ *on input* $(\texttt{Process}, c_0, \widehat{c}, \vec{p})$ *from* $\mathcal{S}_{2\text{-}5}$.

Let G' *be either the protocol state included in* $G.\mathsf{pendCom}[c_0]$ *of* id *after executing a commit protocol or the updated protocol state of* id *after executing the process protocol Then, we have the following (after* $\mathcal{S}_{2\text{-}5}$ *receives an output from* id *but before it provides input to* $\mathcal{F}_{CGKA,2\text{-}5}$*):*

- *if* $\mathsf{Node}[c_0] \neq \bot$*, then*
  - $\mathsf{Node}[c_0].\mathsf{orig} = \mathsf{id}$ *(resp.* id_c*) if* id *executed a commit (resp. process) protocol;*
  - $\mathsf{Node}[c_0].\mathsf{par} = \mathsf{Ptr}[\mathsf{id}]$*;*
  - $\mathsf{Node}[c_0].\mathsf{prop} = \vec{p}$*;*
  - $\mathsf{Node}[c_0].\mathsf{mem} = G'.\mathsf{memberIDsvks}()$.
- *if* c_0 *is attached to a detached root* root_{rt} *(i.e., either* $\mathsf{Node}[c_0] = \bot$ *and* $\mathcal{S}_{2\text{-}5}$ *outputs* $(ack := \texttt{true}, rt \neq \bot, c_0, \vec{c}, w_0, \vec{w})$ *if* id *executed a commit protocol; or* $\mathsf{Node}[c_0] = \bot$ *and* $\mathcal{S}_{2\text{-}5}$ *outputs* $(ack := \texttt{true}, rt \neq \bot, \bot, \bot)$ *if* id *executed a process protocol), then*
  - $\mathsf{Node}[\mathsf{root}_{rt}].\mathsf{orig} = \mathsf{id}$ *(resp.* id_c*) if* id *executed a commit (resp. process) protocol;*
  - $\mathsf{Node}[\mathsf{root}_{rt}].\mathsf{par} = \bot$*;*
  - $\mathsf{Node}[\mathsf{root}_{rt}].\mathsf{prop} = \bot$*;*
  - $\mathsf{Node}[\mathsf{root}_{rt}].\mathsf{mem} = G'.\mathsf{memberIDsvks}()$.

Proof. We first prove the simpler second case where $\mathsf{Node}[c_0] = \bot$ and c_0 is assigned to a detached root. Notice that $\mathsf{Node}[\mathsf{root}_{rt}]$ is only created during a join query. Let w'_0 be the associating welcome message that is used to create $\mathsf{Node}[\mathsf{root}_{rt}]$ and assume party id' was invoked by this join query. Let $G_{\mathsf{id}'}$ be the protocol state after id' processes the welcome message. Then, by *Case (B)* of Fact 1, we have $\mathsf{Node}[\mathsf{root}_{rt}].\mathsf{orig} = \mathsf{id}'_c$ and $\mathsf{Node}[\mathsf{root}_{rt}].\mathsf{mem} = G_{\mathsf{id}'}.\mathsf{memberIDsvks}()$, where id'_c is those included in w'_0. By how $\mathcal{S}_{2\text{-}5}$ simulates the commit and process queries, if id was invoked on a commit or a process query, then c_0 includes the same $\mathsf{confTag}$ as w'_0. Then, due to the modification we made in Hybrid 2-2 [*Unique* $\mathsf{confTag}$ *in* L_{com} *and* L_{wel}], we have $G'.\mathsf{confKey} = G_{\mathsf{id}'}.\mathsf{confKey}$ and $G'.\mathsf{confTransHash} = G_{\mathsf{id}'}.\mathsf{confTransHash}$. Then, by Fact 1 and due to the modification we made in Hybrid 2-1 [*No collision in RO*], we also have $G'.\mathsf{memberHash} = G_{\mathsf{id}'}.\mathsf{memberHash}$ and $\mathsf{id} = \mathsf{id}'_c$ (resp. $\mathsf{id}_c = \mathsf{id}'_c$) if id is invoked on a commit (resp. process) query. Finally, by the definition of `*derive-member-hash` (see Figure 4.23), we have $G'.\mathsf{memberIDsvks}() = G_{\mathsf{id}'}.\mathsf{memberIDsvks}()$. Since we have $\mathsf{Node}[\mathsf{root}_{rt}].\mathsf{par} = \mathsf{Node}[\mathsf{root}_{rt}].\mathsf{prop} = \bot$ by definition, this concludes the proof for the second case where c_0 is assigned to a detached root.

It remains to prove the first case where $\mathsf{Node}[c_0] \neq \bot$. There are four cases where $\mathsf{Node}[c_0]$ can be created when c_0 is a non-root.

(1) Some party id' output $(c_0, \vec{c}'_0, w'_0, \vec{w}'_0)$ on input $(\texttt{Commit}, \vec{p}', \mathsf{svk}')$ from $\mathcal{S}_{2\text{-}5}$ and there does not exist any w_0 such that $\mathsf{Wel}[w_0] = \mathsf{root}_{rt'}$ for any $rt' \in \mathbb{N}$ that includes the same $\mathsf{confTag}$ as c_0 (before $\mathcal{S}_{2\text{-}5}$ provides input to $\mathcal{F}_{\text{CGKA},2\text{-}5}$);

(2) Some party id' output $(c_0, \vec{c}'_0, w'_0, \vec{w}'_0)$ on input $(\texttt{Commit}, \vec{p}', \mathsf{svk}')$ from $\mathcal{S}_{2\text{-}5}$ and there exists a w_0 such that $\mathsf{Wel}[w_0] = \mathsf{root}_{rt'}$ for some $rt' \in \mathbb{N}$ that includes the same $\mathsf{confTag}$ as c_0 (before $\mathcal{S}_{2\text{-}5}$ provides input to $\mathcal{F}_{\text{CGKA,2-5}}$);

(3) Some party id' output $(\mathsf{id}'_c, upd' \| rem' \| add')$ on input $(\texttt{Process}, c_0, \widehat{c}', \vec{p}')$ from $\mathcal{S}_{2\text{-}5}$ and there does not exist any w_0 such that $\mathsf{Wel}[w_0] = \mathsf{root}_{rt'}$ for any $rt' \in \mathbb{N}$ that includes the same $\mathsf{confTag}$ as c_0 (before $\mathcal{S}_{2\text{-}5}$ provides input to $\mathcal{F}_{\text{CGKA,2-5}}$);

(4) Some party id' output $(\mathsf{id}'_c, upd' \| rem' \| add')$ on input $(\texttt{Process}, c_0, \widehat{c}', \vec{p}')$ from $\mathcal{S}_{2\text{-}5}$ and there exists a w_0 such that $\mathsf{Wel}[w_0] = \mathsf{root}_{rt'}$ for some $rt' \in \mathbb{N}$ that includes the same $\mathsf{confTag}$ as c_0 (before $\mathcal{S}_{2\text{-}5}$ provides input to $\mathcal{F}_{\text{CGKA,2-5}}$).

Since the proof for the latter two cases are almost identical to the former two cases we only prove Cases (1) and (2).

For both cases, let $G'_{\mathsf{id}'}$ be the pending protocol state included in $G_{\mathsf{id}'}.\mathsf{pendCom}[c_0]$ of the protocol state $G_{\mathsf{id}'}$ of id' after executing the commit query. Then, by the correctness of the ideal commit protocol, we have $\mathsf{Node}[c_0].\mathsf{orig} = \mathsf{id}'$, $\mathsf{Node}[c_0].\mathsf{par} = \mathsf{Ptr}[\mathsf{id}']$, $\mathsf{Node}[c_0].\mathsf{prop} = \vec{p}'$, and $\mathsf{Node}[c_0].\mathsf{mem} = G_{\mathsf{id}'}.\mathsf{memberIDsvks}()$, where the last condition holds due to Proposition 4.4.13. Note that we have $\mathsf{Node}[c_0].\mathsf{par}$ and $\mathsf{Node}[c_0].\mathsf{prop}$ in Case (2) due to the `*consistent-com` and `*attach` functions in the ideal commit protocol.

Now, since id' and id output the same c_0, and c_0 includes id, we have $\mathsf{id} = \mathsf{id}'$. Hence, $\mathsf{Node}[c_0].\mathsf{orig} = \mathsf{id}$. Moreover, by the modification in Hybrid 2-2 [*Unique* $\mathsf{confTag}$ *in* L_{com}], we have $(G'_{\mathsf{id}'}.\mathsf{confKey}, G'_{\mathsf{id}'}.\mathsf{confTransHash}) = (G'.\mathsf{confKey}, G'.\mathsf{confTransHash})$. Then, by Proposition 4.4.10, since both $\mathsf{confKey}$ are the same we have $\mathsf{Ptr}[\mathsf{id}] = \mathsf{Ptr}[\mathsf{id}']$. Hence, $\mathsf{Node}[c_0].\mathsf{par} = \mathsf{Ptr}[\mathsf{id}]$. Also, due to the modification we made in Hybrid 2-1 [*No collision in RO*] and by the definition of `*derive-member-hash` (see Figure 4.23), we also have $\vec{p}' = \vec{p}$ and $G'.\mathsf{memberIDsvks}() = G'_{\mathsf{id}'}.\mathsf{memberIDsvks}()$. Therefore, $\mathsf{Node}[c_0].\mathsf{prop} = \vec{p}$ and $\mathsf{Node}[c_0].\mathsf{mem} = G_{\mathsf{id}}.\mathsf{memberIDsvks}()$. This completes the proof for Cases (1) and (2).

This concludes the proof. □

Combining the propositions in *Part 2*, we obtain the following corollary.

Corollary 4.4.16 (Invariant `cons-invariant`). *`cons-invariant` in Figure 4.14 always outputs* `true` *(condition on $\mathcal{S}_{2\text{-}5}$ not aborting).*

Proof. The first two conditions (a) and (b) hold due to Fact 1 and Propositions 4.4.12, 4.4.14 and 4.4.15. Moreover, Condition (c) holds due to the modification we made in Hybrid 2-2 [*Unique* $\mathsf{confTag}$ *in* L_{com}] and the fact that attaching detached roots to an existing non-root can not cause a cycle in the history graph. □

Part 3. Analysis of the simulation. We are finally ready to analyze that $\mathcal{S}_{2\text{-}5}$ provides the same view to $\mathcal{Z}$ as in the previous Hybrid 2-4. Below, we only focus on the case $\mathcal{S}_{2\text{-}5}$ receives a non-$\perp$ from the simulated parties. Otherwise, $\mathcal{S}_{2\text{-}5}$ can perfectly simulate the previous hybrid by simply setting $ack = \texttt{false}$.

(1) Analysis of Create. It is clear that $\mathcal{F}_{\text{CGKA,2-5}}$ outputs $\perp$ to $\mathcal{Z}$ if and only if $\mathcal{S}_{2\text{-}4}$ returned $\perp$ to $\mathcal{Z}$ (or to $\mathcal{F}_{\text{CGKA,2-5}}$ to be more precise) in Hybrid 2-4. Therefore, the view of $\mathcal{Z}$ remains identical in both hybrids.

(2) Analysis of Proposal. If party id returns p to $\mathcal{S}_{2\text{-}5}$, we need to check that $\mathcal{F}_{\text{CGKA,2-5}}$ also returns p to $\mathcal{Z}$ as in the previous hybrid. We only focus on $\mathsf{act} =$ 'upd'-svk since the other cases are just a simplification of this check. We first check that the `*valid-svk` check made by $\mathcal{F}_{\text{CGKA,2-5}}$ on line 4 of $(\texttt{Process}, \mathsf{act})$ in Figure 4.10 succeeds. For id to have output p, we need $\texttt{*fetch-ssk-if-nec}(G, \mathsf{svk}) = \mathsf{ssk} \neq \perp$ in the real protocol (see Figure 4.22). Within $\texttt{*fetch-ssk-if-nec}(G, \mathsf{svk})$, if $G.\mathsf{member}[G.\mathsf{id}].\mathsf{svk} \neq \mathsf{svk}$, then we must have $\mathsf{SSK}[\mathsf{id}, \mathsf{svk}] \neq \perp$ due to the check made by $\mathcal{F}_{\text{AS}}$. This implies that $\mathcal{F}_{\text{AS}}$ run within `*valid-svk`

in $\mathcal{F}_{\text{CGKA,2-5}}$ outputs `true` on input $(\texttt{has-ssk}, \mathsf{id}, \mathsf{svk})$, and hence, $\texttt{*valid-svk}(\mathsf{id}, \mathsf{svk})$ also outputs `true`. On the other hand, since we have $\mathsf{Node}[c_0].\mathsf{mem} = G.\mathsf{memberIDsvks}()$ due to Proposition 4.4.14, if $G.\mathsf{member}[G.\mathsf{id}].\mathsf{svk} = \mathsf{svk}$, then we have $\mathsf{Node}[\mathsf{Ptr}[\mathsf{id}]].\mathsf{mem}[\mathsf{id}] = \mathsf{svk}$. Therefore, $\texttt{*valid-svk}(\mathsf{id}, \mathsf{svk})$ also outputs `true` in this case as well. Therefore, $\texttt{*valid-svk}(\mathsf{id}, \mathsf{svk})$ outputs `true` if id did not return $\perp$ in the real protocol.

Finally, due to Proposition 4.4.12, the **assert** condition checked within `*consistent-prop` on line 9 of $(\texttt{Propose}, \mathsf{act})$ is not triggered when $\mathsf{Prop}[p] \neq \perp$. Therefore, if party id returns p to $\mathcal{S}_{\text{2-5}}$, then $\mathcal{F}_{\text{CGKA,2-5}}$ also returns p to $\mathcal{Z}$ as specified.

(3) Analysis of Commit. Assume $\mathsf{Ptr}[\mathsf{id}] = c_0'$ and let $G \neq \perp$ be the protocol state of the simulated party id before it executes the commit. The check made by `*valid-svk` is the same check we covered in the analysis of proposal (see above *(2)*). We therefore first check the **assert** condition on line 8 is never triggered. To do so, we first establish that the set mem output by `*next-members` is identical to $G'.\mathsf{memberIDs}$ (but possibly ordered differently), where G' is the protocol state generated on line 3 in the real commit protocol. This consists of three checks. First, by Proposition 4.4.14 we have $\mathsf{Node}[c_0'].\mathsf{mem} = G.\mathsf{memberIDsvks}()$. Therefore, if commit succeeds in the real protocol, then we have $(\mathsf{id}, *) \in \mathsf{Node}[c_0'].\mathsf{mem}$. Next, due to how $\mathcal{S}_{\text{2-5}}$ answers to `*fill-prop` and by Proposition 4.4.12, we have $\mathsf{Prop}[p] \neq \perp$ and $\mathsf{Prop}[p].\mathsf{par} = c_0'$ for all $p \in \vec{p}$ and the contents of $\mathsf{Prop}[p]$ (i.e., $\mathsf{Prop}[p].\mathsf{orig}$ and $\mathsf{Prop}[p].\mathsf{act}$) are consistent with p. Combining the three checks, we are guaranteed that `*next-members` outputs mem is identical to those created in `*apply-props` in the real protocol. Therefore, this establishes that the **assert** condition $\mathsf{mem} \neq \perp$ and $(\mathsf{id}, \mathsf{svk}) \in \mathsf{mem}$ are satisfied.

To finish the remaining analysis, we consider three cases: $\mathsf{Node}[c_0] = \perp \wedge rt = \perp$, $\mathsf{Node}[c_0] = \perp \wedge rt \in \mathbb{N}$, and $\mathsf{Node}[c_0] \neq \perp$. Here, recall that the **assert** condition `cons-invariant` $\wedge$ `auth-invariant` in line 28 of commit is never triggered as they are always set to `true`. This follows from the fact that **sig-inj-allowed** and **mac-inj-allowed** are always set to `true` in this hybrid and `cons-invariant` is always set to `true` due to Corollary 4.4.16.

[*Case 1:* $\mathsf{Node}[c_0] = \perp$ and $rt = \perp$] It suffices to show that $\mathsf{Wel}[w_0] = \perp$ when $w_0 \neq \perp$ (see line 18 in ideal commit protocol). Let us prove by contradiction and assume $\mathsf{Wel}[w_0] \neq \perp$. First, by how $\mathcal{S}_{\text{2-5}}$ simulates the commit query (see **(3)** of *Part 1*), there does not exist w_0' such that $\mathsf{Wel}[w_0'] = \mathsf{root}_{rt'}$ for some $rt' \in \mathbb{N}$ and w_0' includes the same confTag as w_0. This means, $\mathsf{Wel}[w_0] = c_0''$ for some non-root $c_0'' \neq c_0$ such that $\mathsf{Node}[c_0''] \neq \perp$. Since $\mathsf{Wel}[w_0]$ is assigned a non-root value only during a commit query, we must have that some id′ (possibly id) was invoked by $\mathcal{S}_{\text{2-5}}$ and output c_0'' and w_0. Due to correctness of the protocol, c_0'' and w_0 must include the same confTag. Similarly, c_0 and w_0 must also include the same confTag. However, due to the modification we made in Hybrid 2-2 [*Unique* confTag *in* L_{com}], the simulator $\mathcal{S}_{\text{2-5}}$ aborts the simulation as in the prior hybrid. Therefore, $\mathcal{S}_{\text{2-5}}$ provides the same view to $\mathcal{Z}$ as in the prior hybrid.

[*Case 2:* $\mathsf{Node}[c_0] = \perp$ and $rt \neq \perp$] It suffices to verify that the checks run within `*consistent-com` are satisfied and $\mathsf{Wel}[w_0] \in \{\perp, c_0\}$ when $w_0 \neq \perp$ (see line 18 in ideal commit protocol). Let us consider the former check. Due to Proposition 4.4.15, the only check within `*consistent-com` that we need to verify is whether we have $\mathsf{Rand}[\mathsf{id}] = \text{'bad'}$. Here, note that the condition $\mathsf{Node}[c_0].\mathsf{mem} = \mathsf{mem}$ is satisfied since we established $\mathsf{mem} = G'.\mathsf{memberIDsvks}()$ above. Now, due to the modification we made in Hybrid 2-4 [*Unique* c_0 *with good randomness*] , unless $\mathcal{S}_{\text{2-5}}$ runs party id on the same randomness, we must have $\mathsf{Node}[c_0] = \perp$ since every c_0 output by the parties include a unique confTag. Hence, we must have $\mathsf{Rand}[\mathsf{id}] = \text{'bad'}$ as desired and all the checks run within `*consistent-com` are satisfied. Finally, it is clear that `*attach` on line 24 of the commit procedure assigns c_0 to $\mathsf{Wel}[w_0]$. Therefore, the latter check on $\mathsf{Wel}[w_0] = c_0$ is also satisfied.

[*Case 3:* $\mathsf{Node}[c_0] \neq \perp$] It suffices to verify that the checks run within `*consistent-com` are satisfied and $\mathsf{Wel}[w_0] \in \{\perp, c_0\}$ when $w_0 \neq \perp$ (see line 18 in ideal commit protocol). Since the check regarding

`*consistent-com` is identical to the above *Case 2*, we only consider the latter check. Assume for the sake of contradiction that $\mathsf{Wel}[w_0] = c_0'' \neq c_0$ for $c_0'' \neq \bot$ and $\mathsf{Node}[c_0''] \neq \bot$. Observe the only situation the value of $\mathsf{Wel}[w_0]$ is set is either during a commit query or a join query. We first consider the case $\mathsf{Wel}[w_0]$ is set during a commit query. If this case occurs, then this implies that some id' output (c_0'', w_0) as otherwise the **assert** condition regarding $\mathsf{Wel}[w_0]$ in the commit procedure is triggered. Due to the correctness of the real protocol, all c_0, c_0'', and w_0 include the same $\mathsf{confTag}$. However, due to the modification we made in Hybrid 2-2 [*Unique* $\mathsf{confTag}$ *in* L_{com}], we must have $c_0'' = c_0$ or otherwise the simulator $\mathcal{S}_{2\text{-}5}$ aborts the simulation as in the previous hybrid. Hence, we have $\mathsf{Wel}[w_0] = c_0$ as desired.

Let us consider the other case where $\mathsf{Wel}[w_0]$ is set during a join query, which implies that some id' output c_0''. We have two cases to consider: c_0'' is a non-root or a detached root. If c_0'' is a non-root, then due to how $\mathcal{S}_{2\text{-}5}$ simulates the join query (see **(5)** of *Part 1*), this implies that c_0'' includes the same $\mathsf{confTag}$ as the one included in w_0 and we have $\mathsf{Node}[c_0''] \neq \bot$ when answering the join query. Recall that when c_0'' is a non-root, $\mathsf{Node}[c_0'']$ is set only during a commit or process query. Then, due to the modification we made in Hybrid 2-2 [*Unique* $\mathsf{confTag}$ *in* L_{com}], since c_0'' and c_0 include the same $\mathsf{confTag}$, we must have $c_0'' = c_0$ as desired or otherwise the simulator $\mathcal{S}_{2\text{-}5}$ aborts the simulation as in the previous hybrid. On the other hand, if c_0'' is a detached root, then `*attach` on line 24 of the commit procedure assigns c_0 to $\mathsf{Wel}[w_0]$.

Collecting all the checks, we have either $\mathsf{Wel}[w_0] = \bot$ or $\mathsf{Wel}[w_0] = c_0$ as desired. Therefore, $\mathcal{S}_{2\text{-}5}$ provides the same view to $\mathcal{Z}$ as in the prior hybrid.

(4) Analysis of Process. Let $G \neq \bot$ be the protocol state of the simulated party id after it executes the process protocol. Moreover, assume id outputs $(\mathsf{id}_c, upd \| rem \| add)$ on input $(\mathtt{Process}, c_0, \widehat{c}, \vec{p})$. There are three cases that can occur while $\mathcal{S}_{2\text{-}5}$ answers the process query (see **(4)** of *Part 1*): *Case 1:* $\mathsf{Node}[c_0] = \bot$ and $rt = \bot$; *Case 2:* $\mathsf{Node}[c_0] = \bot$ and $rt \neq \bot$; and *Case 3:* $\mathsf{Node}[c_0] \neq \bot$. We analyze each cases separately.

[*Case 1:* $\mathsf{Node}[c_0] = \bot$ and $rt = \bot$] Following the same argument we made for analyzing the commit query (see *(3)* above), `*next-members` on line 13 of the ideal process protocol outputs $(\mathsf{mem}, \mathsf{propSem})$, where mem is identical to those created in `*apply-props` in the real protocol and $\mathsf{propSem}$ is identical to $upd \| rem \| add$ output by id. This implies that the **assert** conditions on line 14 and line 27 of the ideal process protocol are never triggered. Finally, in *Case 1*, $\mathcal{S}_{2\text{-}5}$ sets $\mathsf{orig}' = \mathsf{id}_c$, where id_c is those output by id, so we conclude that `*output-proc`(c_0) outputs $(\mathsf{id}_c, \mathsf{propSem})$ as in the previous hybrid.

[*Case 2:* $\mathsf{Node}[c_0] = \bot$ and $rt \neq \bot$] Identically to *Case 1*, `*next-members` on line 13 of the ideal process protocol outputs $(\mathsf{mem}, \mathsf{propSem})$, where mem is identical to those created in `*apply-props` in the real protocol and $\mathsf{propSem}$ is identical to $upd \| rem \| add$ output by id. Moreover, by Proposition 4.4.15, we have $\mathsf{Node}[\mathsf{root}_{rt}].\mathsf{orig} = \mathsf{id}_c$, $\mathsf{Node}[\mathsf{root}_{rt}].\mathsf{par} = \bot$, $\mathsf{Node}[\mathsf{root}_{rt}].\mathsf{prop} = \bot$, and $\mathsf{Node}[\mathsf{root}_{rt}].\mathsf{mem} = G.\mathsf{memberIDsvks}()$. Therefore, the check within the `*valid-successor` and `*attach` functions on line 22 and line 23, respectively, all passes. Hence, `*output-proc`(c_0) outputs $(\mathsf{id}_c, \mathsf{propSem})$ as in the previous hybrid.

[*Case 3:* $\mathsf{Node}[c_0] \neq \bot$] This is almost identical to *Case 2*. The only difference is that by Proposition 4.4.15, we have $\mathsf{Node}[c_0].\mathsf{orig} = \mathsf{id}_c$, $\mathsf{Node}[c_0].\mathsf{par} = \mathsf{Ptr}[\mathsf{id}]$, $\mathsf{Node}[c_0].\mathsf{prop} = \vec{p}$, and $\mathsf{Node}[c_0].\mathsf{mem} = G.\mathsf{memberIDsvks}()$. Observe the check within the `*valid-successor` function on line 22 all passes. Hence, `*output-proc`(c_0) outputs $(\mathsf{id}_c, \mathsf{propSem})$ as in the previous hybrid.

(5) Analysis of Join. Let $G \neq \bot$ be the protocol state of the simulated party id after it executes the join protocol. Assume id outputs $(\mathsf{id}_c, G.\mathsf{memberIDsvks}())$ on input $(\mathtt{Join}, w_0, \widehat{w})$. There are four cases that can occur while $\mathcal{S}_{2\text{-}5}$ answers the join query (see **(5)** of *Part 1*): *Case 1:* $\mathsf{Wel}[w_0] \neq \bot$; *Case 2:* $\mathsf{Wel}[w_0] = \bot$ but there exists a unique c_0 including the same $\mathsf{confTag}$ as w_0 satisfying $\mathsf{Node}[c_0] \neq \bot$; *Case 3:* $\mathsf{Wel}[w_0] = \bot$ and no such c_0 exists but there exists a (possibly non-unique) w_0' including the same $\mathsf{confTag}$ as w_0 satisfying $\mathsf{Wel}[w_0] \neq \bot$; and *Case 4:* $\mathsf{Wel}[w_0] = \bot$ and no such c_0 or w_0' exist. We analyze each cases separately.

[*Case 1:*] In case $\mathsf{Wel}[w_0] = c_0$ already exists, it suffices to consider the case it was initially set. Namely, it suffices to check that $\mathcal{S}_{2\text{-}5}$ simulates the previous hybrid in the below *Cases 2, 3*, and *4*.

[*Case 2:*] $\mathsf{Node}[c_0]$ can be set only during a commit or process query. Due to Propositions 4.4.14 and 4.4.15 and correctness of the real protocol, we have $(\mathsf{id}, *) \in \mathsf{Node}[c_0].\mathsf{mem}$. Therefore, the **assert** condition on line 13 if the ideal join protocol is never triggered. Moreover, due to the same reason, the output of the ideal join protocol $(\mathsf{Node}[c_0].\mathsf{orig}, \mathsf{Node}[c_0].\mathsf{mem})$ is identical to those from the previous hybrid (i.e., those output by id).

[*Case 3:*] Since w'_0 includes the same $\mathsf{confTag}$ as w_0, we have $\mathsf{Wel}[w_0]$ is assigned to $\mathsf{Wel}[w'_0]$ due to Proposition 4.4.7. Note that there may exist many w'_0 but all $\mathsf{Wel}[w'_0]$ are identical, so this is well-defined. Moreover, since there is no c_0 that includes the same $\mathsf{confTag}$ as w_0 and w'_0, we must have $\mathsf{Wel}[w'_0] = \mathsf{root}_{rt}$ for some $rt \in \mathbb{N}$. Hence, it suffices to check that $\mathcal{S}_{2\text{-}5}$ simulates the previous hybrid in case $\mathsf{Wel}[w'_0]$ was initially set to root_{rt}, which we provide in the final *Cases 4*.

[*Case 4:*] Since $\mathcal{S}_{2\text{-}5}$ sets $\mathsf{orig}' := \mathsf{id}_c$ and $\mathsf{mem}' := G.\mathsf{memberIDsvks}()$, it is clear that the **assert** condition on line 13 of the ideal join protocol is not triggered. Moreover, since the ideal join protocol outputs $(\mathsf{Node}[c_0].\mathsf{orig} = \mathsf{orig}', \mathsf{Node}[c_0].\mathsf{mem} = \mathsf{mem}')$, $\mathcal{S}_{2\text{-}5}$ simulates the previous hybrid perfectly.

(6) Analysis of Key Query. Due to Proposition 4.4.10, every id and id' such that $\mathsf{Ptr}[\mathsf{id}] = \mathsf{Ptr}[\mathsf{id}]$ contain the same $\mathsf{appSecret}$. Therefore, $\mathcal{S}_{2\text{-}5}$ provides an identical view to $\mathcal{Z}$ of the previous hybrid. □

From Hybrid 2-5 to 2-6: Lemma 4.4.17.

Lemma 4.4.17. *Hybrid 2-5 and Hybrid 2-6 are indistinguishable assuming* CmPKE *and* SIG *are correct with overwhelming probability.*

Proof. The only difference between the previous hybrid occurs when id outputs $\perp$ when invoked on a commit, process, or join query by $\mathcal{S}_{2\text{-}5}$ but `*succeed-com`, `*succeed-proc`, or `*succeed-wel` output `true`, respectively. Notice the ideal `succeed-`$*$ functionalities only care for commit and proposal messages that are not adversarially generated (i.e., $\mathsf{Node}[c_0].\mathsf{stat} \neq$ 'adv' and $\mathsf{Prop}[p].\mathsf{stat} \neq$ 'adv'). Therefore, as long as CmPKE and SIG are do not produce ciphertexts or signatures that do not correctly decrypt or verify, then the statement holds. Here, note that this must hold even the ciphertexts and signatures are created with maliciously generated randomness since the ideal functionality allows for $\mathsf{Rand}[\mathsf{id}] =$ 'bad'. However, since we use the global random oracle to expand the randomness, no adversary can find an input that maps to a bad randomness assuming CmPKE and SIG are correct with overwhelming probability (on honestly generated randomness). This completes the proof. □

From Hybrid 2-6 to 2-7: Lemma 4.4.18.

Lemma 4.4.18. *Hybrid 2-6 and Hybrid 2-7 are indistinguishable assuming* CmPKE *has commitment-binding property.*

Proof. The only difference from the previous hybrid occurs when id outputs non-$\perp$ when invoked on a process query with $(c_0, \widehat{c}', \vec{p})$ such that $\mathsf{Node}[c_0].\mathsf{stat} =$ 'good' and $\widehat{c}' \neq \mathsf{Node}[c_0].\mathsf{vcom}[\mathsf{index}_{\mathsf{id}}]$, where $\mathsf{index}_{\mathsf{id}} \leftarrow \mathsf{Node}[c_0].\mathsf{indexOf}(\mathsf{id})$. This implies there exist two CmPKE ciphertexts $(\mathsf{T}, \widehat{\mathsf{ct}})$ and $(\mathsf{T}, \widehat{\mathsf{ct}}')$ that satisfy

$$\widehat{\mathsf{ct}} \neq \widehat{\mathsf{ct}}' \wedge \mathsf{CmDec}(\mathsf{dk}_{\mathsf{id}}, \mathsf{T}, \widehat{\mathsf{ct}}) \neq \perp \wedge \mathsf{CmDec}(\mathsf{dk}_{\mathsf{id}}, \mathsf{T}, \widehat{\mathsf{ct}}') \neq \perp,$$

where $\mathsf{dk}_{\mathsf{id}}$ is id's current CmPKE decryption key. It is clear that this contradicts the commitment-binding property of CmPKE. Thus, Hybrid 2-6 and Hybrid 2-7 are indistinguishable assuming CmPKE has commitment-binding property. □

From Hybrid 3 to 4: Lemma 4.4.19. Hybrid 4 concerns the authenticity of the signature scheme. The functionality $\mathcal{F}_{\text{CGKA},4}$ halts if the **sig-inj-allowed** predicate returns false, i.e., $\mathcal{Z}$ succeeds to forge a signature without knowing signing keys. To prove Lemma 4.4.19 (i.e., the probability $\mathcal{F}_{\text{CGKA},4}$ halts is negligible), we show that, if $\mathcal{Z}$ can inject a message for which **sig-inj-allowed** predicate returns false, it can be used to break the sEUF-CMA security of SIG. In other words, if SIG is sEUF-CMA secure, the simulator receives only messages which the **sig-inj-allowed** predicate returns true. Thus, $\mathcal{F}_{\text{CGKA},4}$ never halts, and we conclude that Hybrid 3 and Hybrid 4 are indistinguishable. We below provide formal proof of the above overview.

Lemma 4.4.19. *Hybrid 3 and Hybrid 4 are indistinguishable assuming* SIG *is* sEUF-CMA *secure.*

Proof. To show Lemma 4.4.19, we consider the following sub-hybrids between Hybrid 3 and Hybrid 4.

Hybrid 3-0 := **Hybrid 3.** This is identical to Hybrid 3. We use the functionality $\mathcal{F}_{\text{CGKA},3}$ and the simulator $\mathcal{S}_{3\text{-}0} := \mathcal{S}_3$. In this hybrid, the **sig-inj-allowed** predicate always returns true.

Hybrid 3-1. This concerns injections of commit messages. The simulator $\mathcal{S}_{3\text{-}1}$ is defined exactly as $\mathcal{S}_{3\text{-}0}$ except that it aborts if the following condition holds (the simulator checks the condition whenever it updates the history graph).

Condition (A): There exists a non-root node c_0 such that $\mathsf{Node}[c_0].\mathsf{stat} = \text{'adv'}$ and $\mathsf{Node}[c_p].\mathsf{mem}[\mathsf{id}_c] \notin \mathsf{ExposedSvk}$, where $c_p := \mathsf{Node}[c_0].\mathsf{par}$ is the parent of c_0 and $\mathsf{id}_c := \mathsf{Node}[c_0].\mathsf{orig}$ is the committer of c_0.

Condition (A) relates to Condition (a) of `auth-invariant`: If a non-root node c_0 satisfying Condition (A) exists, Condition (a) of `auth-invariant` returns false. We show in Lemma 4.4.20 that Hybrid 3-0 and Hybrid 3-1 are indistinguishable.

Hybrid 3-2. This concerns injections of proposal messages. The simulator $\mathcal{S}_{3\text{-}2}$ is defined exactly as $\mathcal{S}_{3\text{-}1}$ except that it aborts if the following condition holds (the simulator checks the condition whenever it updates the history graph).

Condition (B): There exists a proposal node p such that $\mathsf{Prop}[p].\mathsf{stat} = \text{'adv'}$ and $\mathsf{Node}[c_p].\mathsf{mem}[\mathsf{id}_s] \notin \mathsf{ExposedSvk}$, where $c_p := \mathsf{Prop}[p].\mathsf{par}$ is the parent commit node of p and $\mathsf{id}_s := \mathsf{Prop}[p].\mathsf{orig}$ is the sender of p.

Condition (B) relates to Condition (b) of `auth-invariant`: If a node p satisfying Condition (B) exists, Condition (b) of `auth-invariant` returns false. We show in Lemma 4.4.21 that Hybrid 3-1 and Hybrid 3-2 are indistinguishable.

Hybrid 3-3. This concerns injections of welcome messages. The simulator $\mathcal{S}_{3\text{-}3}$ is defined exactly as $\mathcal{S}_{3\text{-}2}$ except that it aborts if the following condition holds (the simulator checks the condition whenever it updates the history graph).

Condition (C): There exists a detached root root_{rt} such that $\mathsf{Node}[\mathsf{root}_{rt}].\mathsf{mem}[\mathsf{id}_c] \notin \mathsf{ExposedSvk}$, where $\mathsf{id}_c := \mathsf{Node}[\mathsf{root}_{rt}].\mathsf{orig}$ is the committer of the corresponding welcome message.

Condition (C) relates to Condition (c) of `auth-invariant`: If a root node root_{rt} satisfying Condition (C) exists, Condition (c) of `auth-invariant` returns false. We show in Lemma 4.4.22 that Hybrid 3-2 and Hybrid 3-3 are indistinguishable.

Hybrid 3-4 := **Hybrid 4.** This is identical to Hybrid 4. We replace the functionality $\mathcal{F}_{\text{CGKA},3}$ with $\mathcal{F}_{\text{CGKA},4}$, that is, we use the original **sig-inj-allowed** predicate. The simulator $\mathcal{S}_{3\text{-}4}$ is defined exactly as $\mathcal{S}_{3\text{-}3}$ except that it no longer abort. Since $\mathcal{S}_{3\text{-}3}$ aborts if and only if $\mathcal{F}_{\text{CGKA},4}$ halts, Hybrid 3-3 and Hybrid 3-4 are identical.

From Lemmata 4.4.20 to 4.4.22 provided below, Hybrid 3-0 and Hybrid 3-3 are indistinguishable. Moreover, Hybrid 3-3 and Hybrid 3-4 are identical. Therefore, we conclude that Hybrid 3 and Hybrid 4 are indistinguishable. □

From Hybrid 3-0 to 3-1: Lemma 4.4.20.

Lemma 4.4.20. *Hybrid 3-0 and Hybrid 3-1 are indistinguishable assuming* SIG *is* sEUF-CMA *secure.*

Proof. The only difference between $\mathcal{S}_{3\text{-}0}$ and $\mathcal{S}_{3\text{-}1}$ is that $\mathcal{S}_{3\text{-}1}$ aborts if Condition (A) holds. We show that, if $\mathcal{Z}$ can distinguish the two hybrids, then there exists an adversary $\mathcal{B}$ that breaks the sEUF-CMA security of SIG. We first explain the description of $\mathcal{B}$ and how $\mathcal{B}$ extracts a valid signature forgery using $\mathcal{Z}$; we then show the validity of the forged signature, and finally evaluate $\mathcal{B}$'s advantage.

$\mathcal{B}$ simulates for $\mathcal{Z}$ the interaction with $\mathcal{F}_{\text{AS}}^{\text{IW}}$ and $\mathcal{F}_{\text{KS}}^{\text{IW}}$, and the protocol execution of each party as in $\mathcal{S}_{3\text{-}1}$ except for signature generation. Observer that honest signing keys are generated through `register-svk` queries to $\mathcal{F}_{\text{AS}}^{\text{IW}}$. Let svk^* be the challenge signing key received from the sEUF-CMA game. At the beginning of the game, $\mathcal{B}$ chooses an index $i \in [Q]$ at random, where Q is the largest total number of `register-svk` queries from $\mathcal{Z}$. $\mathcal{B}$ embeds the challenge key svk^* in the i-th `register-svk` query (if the i-th signature key is generated with bad randomness, $\mathcal{B}$ aborts). For the other `register-svk` queries, $\mathcal{B}$ generates signature keys as in the previous hybrid. We assume svk^* is embedded in id^*'s signature key. Whenever id^* creates key packages, proposals, or commit/welcome messages using svk^*, $\mathcal{B}$ uses the signing oracle to generate signatures. If id^* is corrupted or it generates a signature using bad randomness while it holds svk^*, $\mathcal{B}$ aborts.

$\mathcal{B}$ extracts a forgery as follows: Whenever $\mathcal{B}$ creates a node c_0, $\mathcal{B}$ checks whether c_0 satisfies Condition (A). If a node c_0 satisfies the condition, $\mathcal{B}$ retrieves the signature sig and the signed message $m := \mathsf{comCont}$ from c_0. If the sender of c_0 is id^*, and the corresponding signature key is svk^*, $\mathcal{B}$ submits (m, sig) to the challenger. Note that (m, sig) is a valid message-signature pair because the node is created only if (m, sig) is valid.

We argue (m, sig) is a valid forgery. Since c_0 satisfies Condition (A) and c_0 is sent from id^* using svk^*, the following holds:

Fact (1): $\mathsf{Node}[c_0].\mathsf{stat} = \text{'adv'}$;

Fact (2): $\mathsf{Node}[c_0].\mathsf{orig} = \mathsf{id}_c = \mathsf{id}^*$;

Fact (3): $\mathsf{Node}[c_p].\mathsf{mem}[\mathsf{id}_c] = \mathsf{svk}^*$, where $c_p := \mathsf{Node}[c_0].\mathsf{par}$; and

Fact (4): $\mathsf{Node}[c_p].\mathsf{mem}[\mathsf{id}_c] \notin \mathsf{ExposedSvk}$.

Fact (1) implies (m, sig) has not been generated by $\mathcal{B}$. Therefore, the signing oracle has not output (m, sig). Facts (2)-(4) imply svk^* has not been exposed. Therefore, (m, sig) is a valid forgery on svk^*, and $\mathcal{B}$ wins the sEUF-CMA game.

We finally evaluate the success probability of $\mathcal{B}$. The probability that $\mathcal{B}$ correctly guesses the signature key used to forge is $1/Q$. Therefore, if $\mathcal{Z}$ distinguishes the two hybrids with non-negligible probability ϵ, $\mathcal{B}$

wins the game with probability at least ϵ/Q, which is also non-negligible. This contradicts the assumption that SIG is sEUF-CMA secure. Therefore, ϵ must be negligible, and we conclude that Hybrid 3-0 and Hybrid 3-1 are indistinguishable for $\mathcal{Z}$. □

From Hybrid 3-1 to 3-2: Lemma 4.4.21.

Lemma 4.4.21. *Hybrid 3-1 and Hybrid 3-2 are indistinguishable assuming* SIG *is* sEUF-CMA *secure.*

Proof. The only difference between $\mathcal{S}_{3\text{-}1}$ and $\mathcal{S}_{3\text{-}2}$ is $\mathcal{S}_{3\text{-}2}$ aborts if Condition (B) holds. We show that, if $\mathcal{Z}$ can distinguish the two hybrids, then there exists an adversary $\mathcal{B}$ that breaks the sEUF-CMA security of SIG. We first explain the description of $\mathcal{B}$ and how $\mathcal{B}$ extracts a valid signature forgery using $\mathcal{Z}$; we then show the validity of the forged signature, and finally evaluate $\mathcal{B}$'s advantage.

$\mathcal{B}$ simulates for $\mathcal{Z}$ the interaction with $\mathcal{F}_{\text{AS}}^{\text{IW}}$ and $\mathcal{F}_{\text{KS}}^{\text{IW}}$, and the protocol execution of each party as shown in Lemma 4.4.20, and extracts the forgery as follows: Whenever $\mathcal{B}$ creates a node p, $\mathcal{B}$ checks whether p satisfies Condition (B). If some node p satisfies the condition, $\mathcal{B}$ retrieves the signature sig and the signed message $m :=$ propCont from p. If the sender of p is id^* and the corresponding signature key is svk^*, $\mathcal{B}$ submits (m, sig) as the forgery.

We argue (m, sig) is a valid forgery. Since p satisfies Condition (B) and p is sent from id^* with svk^*, the following holds:

Fact (1): $\mathsf{Prop}[p].\mathsf{stat} = \text{'adv'}$;

Fact (2): $\mathsf{Prop}[p].\mathsf{orig} = \mathsf{id}_c = \mathsf{id}^*$;

Fact (3): $\mathsf{Node}[c_p].\mathsf{mem}[\mathsf{id}_c] = \mathsf{svk}^*$, where $c_p := \mathsf{Node}[c_0].\mathsf{par}$; and

Fact (4): $\mathsf{Node}[c_p].\mathsf{mem}[\mathsf{id}_c] \notin \mathsf{ExposedSvk}$.

Note that $c_p := \mathsf{Prop}[p].\mathsf{par}$ is the parent of p. Fact (1) implies (m, sig) has not been generated by $\mathcal{B}$. Therefore, the signing oracle has not output (m, sig). Facts (2)-(4) imply svk^* has not been exposed. Therefore, (m, sig) is a valid forgery on svk^*, and $\mathcal{B}$ wins the sEUF-CMA game.

We evaluate the success probability of $\mathcal{B}$. The probability that $\mathcal{B}$ correctly guesses the signature key used to forge is $1/Q$. Therefore, if $\mathcal{Z}$ distinguishes the two hybrids with non-negligible probability ϵ, $\mathcal{B}$ wins the game with probability at least ϵ/Q, which is also non-negligible. This contradicts the assumption that SIG is sEUF-CMA secure. Therefore, ϵ must be negligible, and we conclude that Hybrid 3-1 and Hybrid 3-2 are indistinguishable for $\mathcal{Z}$. □

From Hybrid 3-2 to 3-3: Lemma 4.4.22.

Lemma 4.4.22. *Hybrid 3-2 and Hybrid 3-3 are indistinguishable assuming* SIG *is* sEUF-CMA *secure.*

Proof. The only difference between $\mathcal{S}_{3\text{-}2}$ and $\mathcal{S}_{3\text{-}3}$ is $\mathcal{S}_{3\text{-}3}$ aborts if Condition (C) holds. We show that, if $\mathcal{Z}$ can distinguish the two hybrids, then there exists an adversary $\mathcal{B}$ that breaks the sEUF-CMA security of SIG. We first explain the description of $\mathcal{B}$ and how $\mathcal{B}$ extracts a valid signature forgery using $\mathcal{Z}$; we then show the validity of the forged signature, and finally evaluate $\mathcal{B}$'s advantage.

$\mathcal{B}$ simulates for $\mathcal{Z}$ the interaction with $\mathcal{F}_{\text{AS}}^{\text{IW}}$ and $\mathcal{F}_{\text{KS}}^{\text{IW}}$, and the protocol execution of each party as shown in Lemma 4.4.20, and extracts the forgery as follows: Whenever $\mathcal{B}$ creates a node root_{rt}, $\mathcal{B}$ checks whether root_{rt} satisfies Condition (C). If some node root_{rt} satisfies the condition, $\mathcal{B}$ retrieves the signature sig and

the signed message $m := (\mathsf{groupInfo}, \mathsf{T})$ from w_0. If the sender of w_0 is id^* and the corresponding signature key is svk^*, $\mathcal{B}$ submits (m, sig) as the forgery.

We argue (m, sig) is a valid forgery. Since root_{rt} satisfies Condition (C) and w_0 is valid on $(\mathsf{id}^*, \mathsf{svk}^*)$, the following facts hold:

Fact (1): $\mathsf{Node}[\mathsf{root}_{rt}].\mathsf{stat} = \text{'adv'}$;

Fact (2): $\mathsf{Node}[\mathsf{root}_{rt}].\mathsf{orig} = \mathsf{id}_c = \mathsf{id}^*$;

Fact (3): $\mathsf{Node}[\mathsf{root}_{rt}].\mathsf{mem}[\mathsf{id}_c] = \mathsf{svk}^*$; and

Fact (4): $\mathsf{Node}[\mathsf{root}_{rt}].\mathsf{mem}[\mathsf{id}_c] \notin \mathsf{ExposedSvk}$.

Fact (1) implies (m, sig) has not been generated by $\mathcal{B}$. Therefore, the signing oracle has not output (m, sig). Facts (2)-(4) imply svk^* has not been exposed. Therefore, (m, sig) is a valid forgery on svk^*, and $\mathcal{B}$ wins the sEUF-CMA game.

We evaluate the success probability of $\mathcal{B}$. The probability that $\mathcal{B}$ correctly guesses the signature key used to forge is $1/Q$. Therefore, if $\mathcal{Z}$ distinguishes the two hybrids with non-negligible probability ϵ, $\mathcal{B}$ wins the game with probability at least ϵ/Q, which is also non-negligible. This contradicts the assumption that SIG is sEUF-CMA secure. Therefore, ϵ must be negligible, and we conclude that Hybrid 3-2 and Hybrid 3-3 are indistinguishable for $\mathcal{Z}$. □

From Hybrid 4 to 5: Lemma 4.4.23. Hybrid 5 concerns the authenticity of MAC. The functionality $\mathcal{F}_{\text{CGKA},5}$ halts if the **mac-inj-allowed** predicate returns false, i.e., $\mathcal{Z}$ succeeds to forge a MAC tag without knowing the MAC key. To show Lemma 4.4.23 (i.e., the probability $\mathcal{F}_{\text{CGKA},5}$ halts is negligible), we show that, if $\mathcal{Z}$ can distinguish the two hybrids, we can break the Chained CmPKE conforming GSD security of CmPKE. To this end, we consider that the simulator creates the GSD graph based on epoch secrets and MAC tags. The GSD graph represents the relationship of epoch secrets and MAC tags and indicates which MAC key is exposed (Note that to discuss which MAC key is exposed, we will use the sEUF-CMA security of SIG.). We show that, if $\mathcal{Z}$ injects a message for which the **mac-inj-allowed** predicate returns false, it can be used to break the Chained CmPKE conforming GSD security. In other words, if CmPKE is Chained CmPKE conforming GSD secure, the simulator receives only messages for which the **mac-inj-allowed** predicate returns true. Thus, $\mathcal{F}_{\text{CGKA},5}$ never halts, and we conclude that Hybrid 4 and Hybrid 5 are indistinguishable. We below provide formal proof of the above overview.

Lemma 4.4.23. *Hybrid 4 and Hybrid 5 are indistinguishable assuming* SIG *is* sEUF-CMA *secure and* CmPKE *is* Chained CmPKE *conforming GSD secure.*

Proof. To prove the lemma, we consider the following sub-hybrids between Hybrids 4 and 5:

Hybrid 4-0 := **Hybrid 4.** This is identical to Hybrid 4. We use the functionality $\mathcal{F}_{\text{CGKA},4}$ and the simulator $\mathcal{S}_{4\text{-}0} := \mathcal{S}_4$. In this hybrid, the **mac-inj-allowed** predicate always returns true.[16]

Hybrid 4-1. This concerns injections of key packages. The simulator $\mathcal{S}_{4\text{-}1}$ is defined exactly as $\mathcal{S}_{4\text{-}0}$ except that it aborts if the following condition holds.

[16]Since the **sig-inj-allowed** predicate always returns true (cf. Lemma 4.4.19), the truth value of `auth-invariant` and the truth value of **mac-inj-allowed** are the same.

Condition (KP): There exists a proposal node p such that $\mathsf{Prop}[p].\mathsf{act} = \text{'add'-}\mathsf{id}_t\text{-}\mathsf{svk}_t, \mathsf{svk}_t \notin \mathsf{ExposedSvk}$ and $\mathsf{DK}[\mathsf{id}_t, \mathsf{kp}_t] = \bot$, where kp_t is the key package in p.

We show in Lemma 4.4.24 that Hybrid 4-0 and Hybrid 4-1 are indistinguishable assuming SIG is sEUF-CMA secure. In the following hybrids, if a valid key package is injected, the corresponding signing key is always exposed. We will use this fact to discuss which secret is exposed in the GSD graph.

Hybrid 4-2. We modify the simulator $\mathcal{S}_{4\text{-}1}$ so that it creates the GSD graph based on CmPKE keys, epoch secrets, and MAC tags. Roughly speaking, the GSD graph represents the relationship of CmPKE keys, epoch secrets, and MAC tags, and it indicates which secret is exposed. The simulator $\mathcal{S}_{4\text{-}2}$ internally runs two simulators $\mathcal{S}_{\mathsf{GSD}}$ and $\mathcal{S}'_{4\text{-}2}$: $\mathcal{S}_{\mathsf{GSD}}$ simulates the GSD oracles and creates the GSD graph as shown in Figure 4.30; $\mathcal{S}'_{4\text{-}2}$ simulates the interaction for the environment $\mathcal{Z}$, the functionality, and the adversary $\mathcal{A}$ by using the GSD oracles provided by $\mathcal{S}_{\mathsf{GSD}}$. In addition, $\mathcal{S}'_{4\text{-}2}$ always rejects injected commit/proposal messages if the corresponding MAC key is not exposed, and aborts if a commit node is attached to a detached root although the corresponding initial secret is not exposed.[17] So as not to interrupt the proof, we formally explain how $\mathcal{S}_{\mathsf{GSD}}$ and $\mathcal{S}'_{4\text{-}2}$ are defined. We show in Lemma 4.4.25 that Hybrid 4-1 and Hybrid 4-2 are indistinguishable assuming CmPKE is Chained CmPKE conforming GSD secure.

Hybrid 4-3. We undo the changes made between Hybrid 4-0 and Hybrid 4-2. That is, the simulator $\mathcal{S}_{4\text{-}3}$ no longer aborts the simulation. Using the same arguments to move through Hybrid 4-0 to Hybrid 4-2, Hybrid 4-2 and Hybrid 4-3 remain indistinguishable.

Hybrid 4-4 := **Hybrid 5.** This is identical to Hybrid 5. We replace the functionality $\mathcal{F}_{\text{CGKA},4}$ with $\mathcal{F}_{\text{CGKA},5}$, that is, we use the original **mac-inj-allowed** predicate. The simulator $\mathcal{S}_{4\text{-}4}$ is defined exactly as $\mathcal{S}_{4\text{-}3}$. We show in Lemma 4.4.32 that Hybrid 4-3 and Hybrid 4-4 are identical.

From Lemmata 4.4.24, 4.4.25 and 4.4.32 provided below, Hybrid 4-0 and Hybrid 4-4 are indistinguishable. Therefore, we conclude that Hybrid 4 and Hybrid 5 are indistinguishable.

□

From Hybrid 4-0 to 4-1: Lemma 4.4.24.

Lemma 4.4.24. *Hybrid 4-0 and Hybrid 4-1 are indistinguishable assuming* SIG *is* sEUF-CMA *secure.*

Proof. The only difference between $\mathcal{S}_{4\text{-}0}$ and $\mathcal{S}_{4\text{-}1}$ is $\mathcal{S}_{4\text{-}1}$ aborts Condition (KP) holds. We show that, if $\mathcal{Z}$ can distinguish the two hybrids, then there exists an adversary $\mathcal{B}$ that breaks the sEUF-CMA security of SIG. We first explain the description of $\mathcal{B}$ and how $\mathcal{B}$ extracts a valid signature forgery using $\mathcal{Z}$; we then show the validity of the forged signature and finally evaluate $\mathcal{B}$'s advantage.

$\mathcal{B}$ simulates for $\mathcal{Z}$ the interaction with $\mathcal{F}^{\text{IW}}_{\text{AS}}$ and $\mathcal{F}^{\text{IW}}_{\text{KS}}$, and the protocol execution of each party as in $\mathcal{S}_{4\text{-}1}$ except for the signature generation. Let svk^* be the challenge signature key provided by the sEUF-CMA game. Observer that honest signing keys are generated on `register-svk` queries to $\mathcal{F}^{\text{IW}}_{\text{AS}}$. We assume $\mathcal{Z}$ issues at most Q `register-svk` queries. At the beginning of the game, $\mathcal{B}$ chooses an index $i \in [Q]$ at random, and embeds the challenge key svk^* in the i-th `register-svk` query (if the i-th signature key is generated with bad randomness, $\mathcal{B}$ aborts). For other `register-svk` queries, $\mathcal{B}$ generates signature keys

[17] We define $\mathcal{S}'_{4\text{-}2}$ so that it never creates history graph nodes for which **mac-inj-allowed** returns false.

following the description of $\mathcal{F}_{\mathrm{AS}}^{\mathrm{IW}}$. Assume svk^* is embedded in id^*'s signing key. Whenever id^* creates key packages, proposals, or commit/welcome messages using svk^*, $\mathcal{B}$ uses the signing oracle to generate signatures. If id^* is corrupted or it generates a signature using bad randomness while it holds svk^*, $\mathcal{B}$ aborts.

$\mathcal{B}$ extracts the forgery as follows: Whenever $\mathcal{B}$ creates a node p, $\mathcal{B}$ checks whether p satisfies Condition (KP). If a node p satisfies the condition, $\mathcal{B}$ retrieves the signature sig and the signed message $m := (\mathsf{id}, \mathsf{ek}, \mathsf{svk})$ from p. If $(\mathsf{id}, \mathsf{svk}) = (\mathsf{id}^*, \mathsf{svk}^*)$, $\mathcal{B}$ submits (m, sig) as the forgery. (Note that the proposal node is created only if (m, sig) is valid.)

We argue (m, sig) is a valid forgery. Since p satisfies Condition (KP) and p contains the key package on $(\mathsf{id}^*, \mathsf{svk}^*)$, the following facts hold:

Fact (1): $\mathsf{Prop}[p].\mathsf{act} = \text{'add'-}\mathsf{id}^*\text{-}\mathsf{svk}^*$;

Fact (2): $\mathsf{DK}[\mathsf{id}^*, \mathsf{kp}^*] = \bot$, where kp^* is the key package in p; and

Fact (3): $\mathsf{svk}^* \notin \mathsf{ExposedSvk}$.

Fact (1) implies the adversary outputs the valid signature on svk^* because history graph nodes are created when messages are valid. Fact (2) implies (m, sig) has not been generated by Key Service via `register-kp` query; Specifically, the signing oracle has not output (m, sig). Fact (3) implies svk^* has not been exposed when $\mathcal{B}$ obtains (m, sig). Therefore, (m, sig) is a valid forgery on svk^*, and $\mathcal{B}$ wins the game.

We evaluate the success probability of $\mathcal{B}$. The probability that $\mathcal{B}$ correctly guesses the signature key used to forge is $1/Q$. If $\mathcal{Z}$ can distinguish the hybrids with probability ϵ, $\mathcal{B}$ wins the game within probability ϵ/Q. If ϵ is non-negligible, $\mathcal{B}$ wins sEUF-CMA game with non-negligible probability. This contradicts the assumption that SIG is sEUF-CMA secure. Therefore, ϵ must be negligible, and Hybrid 4-0 and Hybrid 4-1 are indistinguishable for $\mathcal{Z}$. □

From Hybrid 4-1 to 4-2: Lemma 4.4.25. The proof consists of two parts: We first explain how simulator $\mathcal{S}_{4\text{-}2}$ simulates Hybrid 4-2 while creating the GSD graph based on secrets and MAC tags. (see *Part 1*); we then show $\mathcal{S}_{4\text{-}2}$ provides an indistinguishable view to $\mathcal{Z}$ (see *Part 2*).

Part 1: Description of the Simulator $\mathcal{S}_{4\text{-}2}$. We consider that $\mathcal{S}_{4\text{-}2}$ internally runs two simulators $\mathcal{S}_{\mathsf{GSD}}$ and $\mathcal{S}'_{4\text{-}2}$: $\mathcal{S}_{\mathsf{GSD}}$ simulates the GSD oracles and creates the GSD graph following the procedures shown in Figure 4.30, and $\mathcal{S}'_{4\text{-}2}$ simulates the interaction for the environment $\mathcal{Z}$, the functionality, and the adversary $\mathcal{A}$ by using the GSD oracles provided by $\mathcal{S}_{\mathsf{GSD}}$. Looking ahead, we use the GSD game to show several hybrids remain indistinguishable. We allow the reduction to simulate $\mathcal{S}_{4\text{-}2}$ by running $\mathcal{S}'_{4\text{-}2}$ on its own while using the challenger provided by the GSD game as a replacement of $\mathcal{S}_{\mathsf{GSD}}$.

$\mathcal{S}'_{4\text{-}2}$ creates the GSD graph based on secrets (CmPKE decryption keys and epoch secrets) and MAC tags created by simulated parties. $\mathcal{S}'_{4\text{-}2}$ keeps a counter ctr (it is initialized with 1), denoting the smallest unused GSD node. Whenever deriving a new encryption key, epoch secret, or MAC tag, $\mathcal{S}'_{4\text{-}2}$ assigns a GSD node to the secret/tag. For example, when $\mathcal{S}'_{4\text{-}2}$ generates a random secret, it sends an unused GSD node to the GSD oracle and the oracle chooses the value. When $\mathcal{S}'_{4\text{-}2}$ uses a specific value as the secret, it assigns the value to an unused GSD node by using `Set-Secret` oracle. When $\mathcal{S}'_{4\text{-}2}$ computes an epoch secret or MAC tag using the random oracle, it calls `Join-Hash/Hash` oracle to assign the derived secret/tag to an unused GSD node. Throughout the proof, we denote the secret (decryption keys and epoch secrets) by (s, u), the pair of the secret s and the assigned GSD node u. When the node is assigned to the secret, if $\mathcal{S}'_{4\text{-}2}$ knows the secret $s \neq \bot$, the secret is set to (s, u), otherwise set to $(\bot, u)$. The value of each secret will be updated adaptively during the simulation: As soon as $\mathcal{S}'_{4\text{-}2}$ corrupts some GSD node v (i.e., query $\mathtt{Corr}(u)$), for each node u

such that **gsd-exp**(u) becomes true, $\mathcal{S}'_{4\text{-}2}$ computes the secret s_u from previously obtained ciphertexts and corrupted secrets, and replaces all occurrence of $(\perp, u)$ with (s_u, u). Hence, if **gsd-exp**$(u) = \texttt{true}$, then $\mathcal{S}'_{4\text{-}2}$ knows the secret s assigned to the node u. The special case is that the encryption key is generated by $\mathcal{A}$ (this case occurs when an injected add/update proposal or commit message is received). In this case, since $\mathcal{S}'_{4\text{-}2}$ does not know the corresponding decryption key, it sets the decryption key to $(\perp, \perp)$.

$\mathcal{S}'_{4\text{-}2}$ maintains the following lists for the simulation:

- $L_{\text{'memb'}}$: It contains tuple $(u_{\text{'memb'}}, m, \mathsf{membTag})$ where $u_{\text{'memb'}}$ is a GSD node assigned to the membership key, m is a MACed message under the key $u_{\text{'memb'}}$ and membTag is a corresponding a membership tag generated by $\mathcal{S}_{\mathsf{GSD}}$.
- $L_{\text{'epoch'}}$: It contains tuple $(u_{\mathsf{par\text{-}init}}, \mathsf{comSecret}, \mathsf{joinerSecret}, \mathsf{confTransHash}, \mathsf{confTag})$ where $u_{\mathsf{par\text{-}init}}$ is a GSD node assigned to the parent initial secret, comSecret is a commit secret, joinerSecret is a joiner secret, confTransHash is a confirmation hash and confTag is a confirmation tag (i.e., epoch secrets and MAC tag of each epoch).
- $L_{\text{'enc'}}$: It contains tuple $((s, u), u_{\mathsf{id}}, (\mathsf{T}, \mathsf{ct}_{u_{\mathsf{id}}}))$ where (s, u) is a encrypted secret (comSecret or joinerSecret), u_{id} is a GSD node assigned to the encryption key of party id and $(\mathsf{T}, \mathsf{ct}_{u_{\mathsf{id}}})$ is a corresponding CmPKE ciphertext.

The first two lists are used when $\mathcal{S}'_{4\text{-}2}$ needs to recompute the same epoch secrets or MAC tags. In particular, due to the modification we made in Hybrid 2-2, confTag is unique for each joinerSecret and confTransHash (and joinerSecret is unique for each initSecret and comSecret (cf. Hybrid 2-1)). The third $L_{\text{'enc'}}$ is used to check whether the received ciphertext can be sent to `CmDec` oracle.

Now we explain how $\mathcal{S}'_{4\text{-}2}$ simulates the protocol using the GSD oracles. Other procedures not described are identical to the previous hybrid. Note that to make the proof of Proposition 4.4.31 easier to read, oracle calls corresponding to the corruption of a GSD node are highlighted with underlines.

Key package creation for id**.** When $\mathcal{S}'_{4\text{-}2}$ creates a key package, it generates a CmPKE's encryption key for the key package with the help of the GSD oracle.

- If $\mathsf{Rand}[\mathsf{id}] = \text{'good'}$, $\mathcal{S}'_{4\text{-}2}$ generates a CmPKE key pair as $(\mathsf{ek}, \mathsf{dk}) := (\texttt{*get-ek}(u_{\mathsf{ctr}}), (\perp, u_{\mathsf{ctr}}))$ (and ctr is incremented as $\mathsf{ctr} \leftarrow \mathsf{ctr} + 1$). Here, $\mathsf{ek} \leftarrow \texttt{*get-ek}(u)$ denotes that $\mathcal{S}'_{4\text{-}2}$ obtains the encryption key ek on a node u by calling the oracle $\texttt{CmEnc}(\{u\}, 0)$ (the special node 0 is only used here). $\mathcal{S}'_{4\text{-}2}$ creates a key package based on the encryption key.
- Else, key packages are generated as in the previous hybrid. After generating the key package, $\mathcal{S}'_{4\text{-}2}$ assigns the seed s of the CmPKE key to an unused GSD node $u_{\mathsf{id}} := u_{\mathsf{ctr}}$ (ctr is incremented) by querying $\underline{\texttt{Set-Full-Secret}(s, u_{\mathsf{id}})}$. It sets $\mathsf{dk} := (\mathsf{dk}_{\mathsf{id}}, u_{\mathsf{id}})$.

Simulation of $\mathsf{id}_{\mathsf{creator}}$ **on input** $(\texttt{Create}, \mathsf{svk})$**.** $\mathcal{S}'_{4\text{-}2}$ creates a new group following the protocol, except for the initialization of epoch secrets and the confirmation tag. (The initial key package is created following the procedure in *Key package creation for* id above.)

$\mathcal{S}'_{4\text{-}2}$ generates the initial epoch secrets as follows: $\mathcal{S}'_{4\text{-}2}$ first prepares new GSD nodes $u_{\text{'joi'}}$, $u_{\text{'conf'}}$, $u_{\text{'app'}}$, $u_{\text{'memb'}}$, and $u_{\text{'init'}}$ (their values are set to $u_{\mathsf{ctr}}, u_{\mathsf{ctr}+1}, \ldots, u_{\mathsf{ctr}+4}$ and ctr is incremented as $\mathsf{ctr} \leftarrow \mathsf{ctr} + 5$).

- If $\mathsf{Rand}[\mathsf{id}_{\mathsf{creator}}] = \text{'good'}$, $\mathcal{S}'_{4\text{-}2}$ queries $\texttt{Hash}(u_{\text{'joi'}}, u_{\mathsf{lbl}}, (G.\mathsf{groupCont}(), \mathsf{lbl}))$ for each $\mathsf{lbl} \in \{\text{'conf'}, \text{'app'}, \text{'memb'}, \text{'init'}\}$.[18] It sets $G.\mathsf{joinerSecret} := (\perp, u_{\text{'joi'}})$, $G.\mathsf{confKey} := (\perp, u_{\text{'conf'}})$, $G.\mathsf{appSecret} :=$

[18] $G.\mathsf{groupCont}()$ returns the group information defined in Table 4.3. It is determined before the party computes the epoch secret.

$(\bot, u_{\text{'app'}})$, $G.\mathsf{membKey} := (\bot, u_{\text{'memb'}})$, $G.\mathsf{initSecret} := (\bot, u_{\text{'init'}})$. Note that the value assigned to $u_{\text{'joi'}}$ is set to random during the first call of Hash.

- Else, $\mathcal{S}'_{4\text{-}2}$ generates the joiner secret $s_{\text{'joi'}}$ using the randomness provided from $\mathcal{A}$, and computes $s_{\mathsf{lbl}} := \mathsf{RO}(s_{\text{'joi'}}, (G.\mathsf{groupCont}(), \mathsf{lbl}))$ for each $\mathsf{lbl} \in \{\text{'conf'}, \text{'app'}, \text{'memb'}, \text{'init'}\}$. Then, $\mathcal{S}'_{4\text{-}2}$ assigns the epoch secrets to GSD oracles: It queries $\texttt{Set-Secret}(u_{\text{'joi'}}, s_{\text{'joi'}})$ and $\texttt{Hash}(u_{\text{'joi'}}, u_{\mathsf{lbl}}, (G.\mathsf{groupCont}(), \mathsf{lbl}))$ for each $\mathsf{lbl} \in \{\text{'conf'}, \text{'app'}, \text{'memb'}, \text{'init'}\}$. $\mathcal{S}'_{4\text{-}2}$ sets $G.\mathsf{joinerSecret} := (s_{\text{'joi'}}, u_{\text{'joi'}})$, $G.\mathsf{confKey} := (s_{\text{'conf'}}, u_{\text{'conf'}})$, $G.\mathsf{appSecret} := (s_{\text{'app'}}, u_{\text{'app'}})$, $G.\mathsf{membKey} := (s_{\text{'memb'}}, u_{\text{'memb'}})$, $G.\mathsf{initSecret} := (s_{\text{'init'}}, u_{\text{'init'}})$.

$\mathcal{S}'_{4\text{-}2}$ generates the confirmation tag as follows:

- If $\mathsf{Rand}[\mathsf{id}_{\mathsf{creator}}] = \text{'good'}$, $\mathcal{S}'_{4\text{-}2}$ prepares a new GSD node $u_{\mathsf{ctag}} := u_{\mathsf{ctr}}$ (and sets $\mathsf{ctr} \leftarrow \mathsf{ctr} + 1$). Then, it queries $\texttt{Hash}(u_{\text{'conf'}}, u_{\mathsf{ctag}}, G.\mathsf{confTransHash})$ and $G.\mathsf{confTag} := \texttt{Corr}(u_{\mathsf{ctag}})$. (See Remark 4.4.2.)
- Else, $\mathcal{S}'_{4\text{-}2}$ computes $G.\mathsf{confTag} := \mathsf{RO}(s_{\text{'conf'}}, G.\mathsf{confTransHash})$ as in the previous hybrid.

Finally, $\mathcal{S}'_{4\text{-}2}$ stores $L_{\text{'epoch'}} \xleftarrow{+} (\bot, (\bot, \bot), G.\mathsf{joinerSecret}, G.\mathsf{confTransHash}, G.\mathsf{confTag})$.

Simulation of id **on input** $(\texttt{Propose}, \mathsf{act})$**.** $\mathcal{S}'_{4\text{-}2}$ generates a proposal message p as in the previous hybrid, except for the generation of key packages and membership tags.

When $\mathcal{S}'_{4\text{-}2}$ generates an update proposal, it creates a key package following the procedure in *Key package creation for* id above. Then, $\mathcal{S}'_{4\text{-}2}$ keeps the generated decryption key dk for p (dk will be used when $\mathcal{S}'_{4\text{-}2}$ receives p in the commit or process protocol).

$\mathcal{S}'_{4\text{-}2}$ generates the membership tag as follows, by using the membership key $G.\mathsf{membKey} = (s_{\text{'memb'}}, u_{\text{'memb'}})$ at epoch $\mathsf{Ptr}[\mathsf{id}]$.

- If $\textbf{gsd-exp}(u_{\text{'memb'}}) = \texttt{true}$ (i.e., $s_{\text{'memb'}} \neq \bot$)[19], $\mathcal{S}'_{4\text{-}2}$ computes $\mathsf{membTag} := \mathsf{RO}(s_{\text{'memb'}}, (\mathsf{propCont}, \mathsf{sig}))$ as in the previous hybrid.
- Else if $(u_{\text{'memb'}}, (\mathsf{propCont}, \mathsf{sig}), \mathsf{membTag}) \in L_{\text{'memb'}}$ exists for some membTag, it is used as the membership tag. Note that this case occurs when the same p is generated multiple times.
- Else, $\mathcal{S}'_{4\text{-}2}$ prepares a new GSD node $u_{\mathsf{mtag}} := u_{\mathsf{ctr}}$ (and sets $\mathsf{ctr} \leftarrow \mathsf{ctr} + 1$) and queries $\texttt{Hash}(u_{\text{'memb'}}, u_{\mathsf{mtag}}, (\mathsf{propCont}, \mathsf{sig}))$ and $\mathsf{membTag} := \texttt{Corr}(u_{\mathsf{mtag}})$. $\mathcal{S}'_{4\text{-}2}$ stores $L_{\text{'memb'}} \xleftarrow{+} (u_{\text{'memb'}}, (\mathsf{propCont}, \mathsf{sig}), \mathsf{membTag})$.

Simulation of id **on input** $(\texttt{Commit}, \vec{p}, \mathsf{svk})$**.** $\mathcal{S}'_{4\text{-}2}$ generates a commit message as in the previous hybrid, except for the differences shown below:

In *unframe-prop, $\mathcal{S}'_{4\text{-}2}$ verifies the membership tag membTag in p as follows, by using the membership key $G.\mathsf{membKey} = (s_{\text{'memb'}}, u_{\text{'memb'}})$ at epoch $\mathsf{Ptr}[\mathsf{id}]$:

- If $\textbf{gsd-exp}(u_{\text{'memb'}}) = \texttt{true}$ (i.e., $s_{\text{'memb'}} \neq \bot$), $\mathcal{S}'_{4\text{-}2}$ computes $\mathsf{membTag}' = \mathsf{RO}(s_{\text{'memb'}}, (\mathsf{propCont}, \mathsf{sig}))$ and checks whether $\mathsf{membTag} = \mathsf{membTag}'$ holds.
- Else if $(u_{\text{'memb'}}, (\mathsf{propCont}, \mathsf{sig}), \mathsf{membTag}') \in L_{\text{'memb'}}$ exists for some $\mathsf{membTag}'$, $\mathcal{S}'_{4\text{-}2}$ checks whether $\mathsf{membTag} = \mathsf{membTag}'$ holds.
- Else, $\mathcal{S}'_{4\text{-}2}$ outputs $\bot$. We call this event $\mathsf{E}_{\mathsf{inj\text{-}p}}$ and in Proposition 4.4.26 we will prove that the simulator in the previous hybrid also outputs $\bot$ when $\mathsf{E}_{\mathsf{inj\text{-}p}}$ occurs. Thus, it does not alter the view of $\mathcal{Z}$

[19] Recall that if $\textbf{gsd-exp}(u) = \texttt{true}$ for the node u, then $\mathcal{S}'_{4\text{-}2}$ knows the secret $s \neq \bot$ assigned to u. This is because if $\textbf{gsd-exp}(u) = \texttt{true}$, $\mathcal{S}'_{4\text{-}2}$ can compute the secret s from the known information (e.g., previously generated ciphertexts).

In `*apply-props`, $\mathcal{S}'_{4\text{-}2}$ updates the membership list as follows:

- If p is an update proposal sent from $\mathsf{id}_s \neq \mathsf{id}$, $\mathcal{S}'_{4\text{-}2}$ stores the decryption key corresponding to p to $G'.\mathsf{member}[\mathsf{id}_s].\mathsf{dk}$. Namely, if p has been generated by $\mathcal{S}'_{4\text{-}2}$, it knows the decryption key. Otherwise (i.e., $\mathcal{S}'_{4\text{-}2}$ does not know the decryption key), $\mathcal{S}'_{4\text{-}2}$ sets $G'.\mathsf{member}[\mathsf{id}_s].\mathsf{dk} := (\bot, \bot)$.
- If p is an add proposal that contains id_t and kp_t, $\mathcal{S}'_{4\text{-}2}$ copies the decryption key stored in $\mathsf{DK}[\mathsf{id}_t, \mathsf{kp}_t]$ to $G'.\mathsf{member}[\mathsf{id}_t].\mathsf{dk}$. If kp_t is not registered to Key Service $\mathcal{F}_{\mathrm{KS}}^{\mathrm{IW}}$ (i.e.,$\mathsf{DK}[\mathsf{id}_t, \mathsf{kp}_t] = \bot$), $\mathcal{S}'_{4\text{-}2}$ sets $G'.\mathsf{member}[\mathsf{id}_t].\mathsf{dk} := (\bot, \bot)$.

After the execution of `*apply-props`, $\mathcal{S}'_{4\text{-}2}$ obtains the new membership list and the decryption key of each member. In other words, for all members id, $G.\mathsf{member}[\mathsf{id}].\mathsf{dk}$ is of the form $(*, u_{\mathsf{id}})$ or $(\bot, \bot)$, where $*$ is either $\bot$ or $s_{\mathsf{id}} \neq \bot$.

$\mathcal{S}'_{4\text{-}2}$ runs `*rekey` as follows: It first creates a key package following the procedure in *Key package creation for* id above. Then $\mathcal{S}'_{4\text{-}2}$ generates a new commit secret:

- If $\mathsf{Rand}[\mathsf{id}] =$ 'good', $\mathcal{S}'_{4\text{-}2}$ first prepares a GSD node $u_{\mathsf{com}} := u_{\mathsf{ctr}}$ (and sets $\mathsf{ctr} \leftarrow \mathsf{ctr} + 1$) and generates a random commit secret as follows. Let $G'.\mathsf{member}$ be the new membership list and receivers be the set of the identity of the existing parties.
  - If $G'.\mathsf{member}[\mathsf{id}].\mathsf{dk} \neq (\bot, \bot)$ for all $\mathsf{id} \in \mathsf{receivers}$, each $G'.\mathsf{member}[\mathsf{id}].\mathsf{dk}$ is of the form $(*, u_{\mathsf{id}})$. $\mathcal{S}'_{4\text{-}2}$ composes the set $S_{\mathsf{receivers}} := \{ u_{\mathsf{id}} \}_{\mathsf{id} \in \mathsf{receivers}}$ and queries $\mathtt{CmEnc}(S_{\mathsf{receivers}}, u_{\mathsf{com}})$ to compute the ciphertext $(\mathsf{T}, \vec{\mathsf{ct}} = (\mathsf{ct}_u)_{u \in S_{\mathsf{receivers}}})$. Note that the commit secret is chosen at random by the `CmEnc` oracle.
    - If $\mathbf{gsd\text{-}exp}(u_{\mathsf{id}}) = \mathtt{true}$ for some $u_{\mathsf{id}} \in S_{\mathsf{receivers}}$, $\mathcal{S}'_{4\text{-}2}$ decrypts $(\mathsf{T}, \mathsf{ct}_{u_{\mathsf{id}}})$ using the secret s_{id} of u_{id}, and obtains s_{com}. Then, $\mathcal{S}'_{4\text{-}2}$ sets $G'.\mathsf{comSecret} := (s_{\mathsf{com}}, u_{\mathsf{com}})$ and stores $L_{\text{'enc'}} \mathrel{+}\!\leftarrow ((s_{\mathsf{com}}, u_{\mathsf{com}}), u_{\mathsf{id}}, (\mathsf{T}, \mathsf{ct}_{u_{\mathsf{id}}}))$ for each $u_{\mathsf{id}} \in S_{\mathsf{receivers}}$.
    - Else, $\mathcal{S}'_{4\text{-}2}$ sets $G'.\mathsf{comSecret} := (\bot, u_{\mathsf{com}})$ and stores $L_{\text{'enc'}} \mathrel{+}\!\leftarrow ((\bot, u_{\mathsf{com}}), u_{\mathsf{id}}, (\mathsf{T}, \mathsf{ct}_{u_{\mathsf{id}}}))$ for each $u_{\mathsf{id}} \in S_{\mathsf{receivers}}$.
  - Else (i.e., $G'.\mathsf{member}[\mathsf{id}].\mathsf{dk} = (\bot, \bot)$ for some $\mathsf{id} \in \mathsf{receivers}$), $\mathcal{S}'_{4\text{-}2}$ generates and encrypts the commit secret s_{com} as in the previous hybrid. Then, $\mathcal{S}'_{4\text{-}2}$ queries $\mathtt{Set\text{-}Secret}(u_{\mathsf{com}}, s_{\mathsf{com}})$ and sets $G'.\mathsf{comSecret} := (s_{\mathsf{com}}, u_{\mathsf{com}})$.
- If $\mathsf{Rand}[\mathsf{id}] =$ 'bad', $\mathcal{S}'_{4\text{-}2}$ generates and encrypts the commit secret s_{com} as in the previous hybrid. In this case, $\mathcal{S}'_{4\text{-}2}$ may have generated or received the same commit message. Thus, $\mathcal{S}'_{4\text{-}2}$ checks if the same commit message has been generated or received earlier, and if not, it assigns a GSD node to the commit secret s_{com}. This check is done when $\mathcal{S}'_{4\text{-}2}$ derives the epoch secret in `*derive-keys` function (see below).

To compute the epoch secrets and confirmation tag, $\mathcal{S}'_{4\text{-}2}$ runs `*derive-keys` and `*gen-conf-tag` as follows: Let $(s_{\mathsf{par\text{-}init}}, u_{\mathsf{par\text{-}init}})$ be the initial secret at epoch ($s_{\mathsf{par\text{-}init}}$ can be $\bot$). Note that $G'.\mathsf{groupCont}()$ (including the confirmation hash $G'.\mathsf{confTransHash}$) for the new epoch is computed before running `*derive-keys` function.

- If $\mathsf{Rand}[\mathsf{id}] =$ 'good', $\mathcal{S}'_{4\text{-}2}$ prepares new GSD nodes $u_{\text{'joi'}}$, $u_{\text{'conf'}}$, $u_{\text{'app'}}$, $u_{\text{'memb'}}$, and $u_{\text{'init'}}$ (their values are set to $u_{\mathsf{ctr}}, u_{\mathsf{ctr}+1}, \ldots, u_{\mathsf{ctr}+4}$ and ctr is incremented as $\mathsf{ctr} \leftarrow \mathsf{ctr} + 5$) and queries $\mathtt{Join\text{-}Hash}(u_{\mathsf{par\text{-}init}}, u_{\mathsf{com}}, u_{\text{'joi'}}, \text{'joi'})$ and $\mathtt{Hash}(u_{\text{'joi'}}, u_{\mathsf{lbl}}, (G'.\mathsf{groupCont}(), \mathsf{lbl}))$ for each $\mathsf{lbl} \in \{\text{'conf'}, \text{'app'}, \text{'memb'}, \text{'init'}\}$.

- If $\textbf{gsd-exp}(u_{\mathsf{par\text{-}init}}) = \texttt{false} \vee \textbf{gsd-exp}(u_{\mathsf{com}}) = \texttt{false}$, $\mathcal{S}'_{4\text{-}2}$ sets $G'.\mathsf{joinerSecret} := (\bot, u_{\text{'joi'}})$, $G'.\mathsf{confKey} := (\bot, u_{\text{'conf'}})$, $G'.\mathsf{appSecret} := (\bot, u_{\text{'app'}})$, $G'.\mathsf{membKey} := (\bot, u_{\text{'memb'}})$ and $G'.\mathsf{initSecret} := (\bot, u_{\text{'init'}})$. Then, $\mathcal{S}'_{4\text{-}2}$ prepares a new GSD node $u_{\mathsf{ctag}} := u_{\mathsf{ctr}}$ (and sets $\mathsf{ctr} \leftarrow \mathsf{ctr}+1$), and queries $\texttt{Hash}(u_{\text{'conf'}}, u_{\mathsf{ctag}}, G'.\mathsf{confTransHash})$ and $G'.\mathsf{confTag} := \texttt{Corr}(u_{\mathsf{ctag}})$.
- Else, (i.e., $\textbf{gsd-exp}(u_{\mathsf{par\text{-}init}}) = \texttt{true} \wedge \textbf{gsd-exp}(u_{\mathsf{com}}) = \texttt{true}$), $\mathcal{S}'_{4\text{-}2}$ knows both $s_{\mathsf{par\text{-}init}}$ and s_{com}. It computes $s_{\text{'joi'}} := \mathsf{RO}(s_{\mathsf{par\text{-}init}}, s_{\mathsf{com}}, \text{'joi'})$ and $s_{\mathsf{lbl}} := \mathsf{RO}(s_{\text{'joi'}}, (G'.\mathsf{groupCont}(), \mathsf{lbl}))$ for each $\mathsf{lbl} \in \{\text{'conf'}, \text{'app'}, \text{'memb'}, \text{'init'}\}$, and sets $G'.\mathsf{joinerSecret} := (s_{\text{'joi'}}, u_{\text{'joi'}})$, $G'.\mathsf{confKey} := (s_{\text{'conf'}}, u_{\text{'conf'}})$, $G'.\mathsf{appSecret} := (s_{\text{'app'}}, u_{\text{'app'}})$, $G'.\mathsf{membKey} := (s_{\text{'memb'}}, u_{\text{'memb'}})$ and $G'.\mathsf{initSecret} := (s_{\text{'init'}}, u_{\text{'init'}})$. $\mathcal{S}'_{4\text{-}2}$ generates the confirmation tag as $G'.\mathsf{confTag} := \mathsf{RO}(s_{\text{'conf'}}, G'.\mathsf{confTransHash})$.

Finally, $\mathcal{S}'_{4\text{-}2}$ stores $L_{\text{'epoch'}} \xleftarrow{+} (u_{\mathsf{par\text{-}init}}, G'.\mathsf{comSecret}, G'.\mathsf{joinerSecret}, G'.\mathsf{confTransHash}, G'.\mathsf{confTag})$.

- If $\mathsf{Rand}[\mathsf{id}] = \text{'bad'}$, $\mathcal{S}'_{4\text{-}2}$ knows $s_{\mathsf{com}} \neq \bot$. $\mathcal{S}'_{4\text{-}2}$ does as follows.
  - If $(u_{\mathsf{par\text{-}init}}, (s_{\mathsf{com}}, *), *, G'.\mathsf{confTransHash}, \mathsf{confTag}') \in L_{\text{'epoch'}}$ exists for some unique $\mathsf{confTag}'$,[20] $\mathcal{S}'_{4\text{-}2}$ uses $\mathsf{confTag}'$ as the confirmation tag. Note that this case occurs if some party has derived the same epoch secrets in the commit or process protocol.
  - Else, $\mathcal{S}'_{4\text{-}2}$ prepares new GSD nodes u_{com}, $u_{\text{'joi'}}$, $u_{\text{'conf'}}$, $u_{\text{'app'}}$, $u_{\text{'memb'}}$, and $u_{\text{'init'}}$ (their values are set to $u_{\mathsf{ctr}}, u_{\mathsf{ctr}+1}, \ldots, u_{\mathsf{ctr}+5}$ and ctr is incremented as $\mathsf{ctr} \leftarrow \mathsf{ctr}+6$) and queries $\texttt{Set-Secret}(u_{\mathsf{com}}, s_{\mathsf{com}})$, $\texttt{Join-Hash}(u_{\mathsf{par\text{-}init}}, u_{\mathsf{com}}, u_{\text{'joi'}}, \text{'joi'})$ and $\texttt{Hash}(u_{\text{'joi'}}, u_{\mathsf{lbl}}, (G'.\mathsf{groupCont}(), \mathsf{lbl}))$ for each $\mathsf{lbl} \in \{\text{'conf'}, \text{'app'}, \text{'memb'}, \text{'init'}\}$.
    - If $\textbf{gsd-exp}(u_{\mathsf{par\text{-}init}}) = \texttt{false}$, $\mathcal{S}'_{4\text{-}2}$ prepares a new node $u_{\mathsf{ctag}} := u_{\mathsf{ctr}}$ (and sets $\mathsf{ctr} \leftarrow \mathsf{ctr}+1$) and computes the confirmation tag by querying $\texttt{Hash}(u_{\text{'conf'}}, u_{\mathsf{ctag}}, G'.\mathsf{confTransHash})$ and $G'.\mathsf{confTag} := \texttt{Corr}(u_{\mathsf{ctag}})$. Then it checks the following.
      - If $(\bot, (\bot, \bot), *, G'.\mathsf{confTransHash}, \mathsf{confTag}') \in L_{\text{'epoch'}}$ exists for some $\mathsf{confTag}'$ such that $G'.\mathsf{confTag} = \mathsf{confTag}'$, $\mathcal{S}'_{4\text{-}2}$ aborts. We call this event $\mathsf{abort}_{\mathsf{attach}}$, and in Proposition 4.4.30 we will prove that the probability the simulator aborts due to this event is negligible.[21]
      - Else, the simulator $\mathcal{S}'_{4\text{-}2}$ sets $G'.\mathsf{comSecret} := (s_{\mathsf{com}}, u_{\mathsf{com}})$, $G'.\mathsf{joinerSecret} := (\bot, u_{\text{'joi'}})$, $G'.\mathsf{confKey} := (\bot, u_{\text{'conf'}})$, $G'.\mathsf{appSecret} := (\bot, u_{\text{'app'}})$, $G'.\mathsf{membKey} := (\bot, u_{\text{'memb'}})$ and $G'.\mathsf{initSecret} := (\bot, u_{\text{'init'}})$. Finally, $\mathcal{S}'_{4\text{-}2}$ stores $L_{\text{'epoch'}} \xleftarrow{+} (u_{\mathsf{par\text{-}init}}, G'.\mathsf{comSecret}, G'.\mathsf{joinerSecret}, G'.\mathsf{confTransHash}, G'.\mathsf{confTag})$.
    - If $\textbf{gsd-exp}(u_{\mathsf{par\text{-}init}}) = \texttt{true}$ (i.e., $\mathcal{S}'_{4\text{-}2}$ knows $s_{\mathsf{par\text{-}init}} \neq \bot$), then $\mathcal{S}'_{4\text{-}2}$ computes the joiner secret $s_{\text{'joi'}} := \mathsf{RO}(s_{\mathsf{par\text{-}init}}, s_{\mathsf{com}}, \text{'joi'})$ and the epoch secrets $s_{\mathsf{lbl}} := \mathsf{RO}(s_{\text{'joi'}}, (G'.\mathsf{groupCont}(), \mathsf{lbl}))$ for each $\mathsf{lbl} \in \{\text{'conf'}, \text{'app'}, \text{'memb'}, \text{'init'}\}$. $\mathcal{S}'_{4\text{-}2}$ sets $G'.\mathsf{comSecret} := (s_{\mathsf{com}}, u_{\mathsf{com}})$, $G'.\mathsf{joinerSecret} := (s_{\text{'joi'}}, u_{\text{'joi'}})$, $G'.\mathsf{confKey} := (s_{\text{'conf'}}, u_{\text{'conf'}})$, $G'.\mathsf{appSecret} := (s_{\text{'app'}}, u_{\text{'app'}})$, $G'.\mathsf{membKey} := (s_{\text{'memb'}}, u_{\text{'memb'}})$ and $G'.\mathsf{initSecret} := (s_{\text{'init'}}, u_{\text{'init'}})$. It also computes the confirmation tag $G'.\mathsf{confTag} := \mathsf{RO}(s_{\text{'conf'}}, G'.\mathsf{confTransHash})$. Finally, $\mathcal{S}'_{4\text{-}2}$ stores $L_{\text{'epoch'}} \xleftarrow{+} (u_{\mathsf{par\text{-}init}}, G'.\mathsf{comSecret}, G'.\mathsf{joinerSecret}, G'.\mathsf{confTransHash}, G'.\mathsf{confTag})$.

If a new member exists, $\mathcal{S}'_{4\text{-}2}$ runs $\texttt{*welcome-msg}$ following the protocol description except that $\mathcal{S}'_{4\text{-}2}$ encrypts the joiner secret $G'.\mathsf{joinerSecret}$ as follows:

[20] Due to the argument we made in Hybrid 2-2, confTag is unique for each initSecret, comSecret and confTransHash.

[21] $\mathsf{abort}_{\mathsf{attach}}$ occurs if the created commit node by the commit protocol is attached to an existing detached root even if the corresponding initial secret is not leaked. If such a node is created, the **mac-inj-allowed** predicate returns false. Namely, we prove in Proposition 4.4.30 that the probability of such a node is created is negligible.

- If $\mathsf{Rand}[\mathsf{id}] = \text{'good'}$, $\mathcal{S}'_{4\text{-}2}$ does as follows. Let $G'.\mathsf{member}$ be the new membership list and addedMem be the set of new party's identities.
  - If $G'.\mathsf{member}[\mathsf{id}].\mathsf{dk} \neq (\perp, \perp)$ for all $\mathsf{id} \in \mathsf{addedMem}$, each $G'.\mathsf{member}[\mathsf{id}].\mathsf{dk}$ is of the form $(*, u_{\mathsf{id}})$. $\mathcal{S}'_{4\text{-}2}$ composes the set $S_{\mathsf{addedMem}} := \{ u_{\mathsf{id}} \}_{\mathsf{id} \in \mathsf{addedMem}}$, and queries $\mathtt{CmEnc}(S_{\mathsf{addedMem}}, u_{\text{'joi'}})$ to compute the ciphertext $(\mathsf{T}, \vec{\mathsf{ct}} = (\mathsf{ct}_u)_{u \in S_{\mathsf{addedMem}}})$.
    - If $\mathbf{gsd\text{-}exp}(u_{\mathsf{id}}) = \mathtt{true}$ for some $u_{\mathsf{id}} \in S_{\mathsf{addedMem}}$, it decrypts $(\mathsf{T}, \mathsf{ct}_{u_{\mathsf{id}}})$ using the secret s_{id} of u_{id}, and obtains $s_{\text{'joi'}}$. $\mathcal{S}'_{4\text{-}2}$ stores $L_{\text{'enc'}} \xleftarrow{+} ((s_{\text{'joi'}}, u_{\text{'joi'}}), u_{\mathsf{id}}, (\mathsf{T}, \mathsf{ct}_{u_{\mathsf{id}}}))$ for each $u_{\mathsf{id}} \in S_{\mathsf{addedMem}}$. Then, $\mathcal{S}'_{4\text{-}2}$ computes $s_{\mathsf{lbl}} := \mathsf{RO}(s_{\text{'joi'}}, (G'.\mathsf{groupCont}(), \mathsf{lbl}))$ for each $\mathsf{lbl} \in \{\text{'conf'}, \text{'app'}, \text{'memb'}, \text{'init'}\}$ and updates the epoch secrets as $G'.\mathsf{joinerSecret} := (s_{\text{'joi'}}, u_{\text{'joi'}})$, $G'.\mathsf{confKey} := (s_{\text{'conf'}}, u_{\text{'conf'}})$, $G'.\mathsf{appSecret} := (s_{\text{'app'}}, u_{\text{'app'}})$, $G'.\mathsf{membKey} := (s_{\text{'memb'}}, u_{\text{'memb'}})$ and $G'.\mathsf{initSecret} := (s_{\text{'init'}}, u_{\text{'init'}})$. ($L_{\text{'epoch'}}$ is also updated accordingly.)
    - Else, $\mathcal{S}'_{4\text{-}2}$ stores $L_{\text{'enc'}} \xleftarrow{+} (G'.\mathsf{joinerSecret}, u_{\mathsf{id}}, (\mathsf{T}, \mathsf{ct}_{u_{\mathsf{id}}}))$ for each $u_{\mathsf{id}} \in S_{\mathsf{addedMem}}$.
  - Else (i.e., $G'.\mathsf{member}[\mathsf{id}].\mathsf{dk} = (\perp, \perp)$ for some $\mathsf{id} \in \mathsf{addedMem}$), $\mathcal{S}'_{4\text{-}2}$ queries $s_{\text{'joi'}} := \mathtt{Corr}(u_{\text{'joi'}})$ and encrypts $s_{\text{'joi'}}$ as in the previous hybrid. Then, $\mathcal{S}'_{4\text{-}2}$ computes $s_{\mathsf{lbl}} := \mathsf{RO}(s_{\text{'joi'}}, (G'.\mathsf{groupCont}(), \mathsf{lbl}))$ for each $\mathsf{lbl} \in \{\text{'conf'}, \text{'app'}, \text{'memb'}, \text{'init'}\}$ and updates the epoch secrets as $G'.\mathsf{joinerSecret} := (s_{\text{'joi'}}, u_{\text{'joi'}})$, $G'.\mathsf{confKey} := (s_{\text{'conf'}}, u_{\text{'conf'}})$, $G'.\mathsf{appSecret} := (s_{\text{'app'}}, u_{\text{'app'}})$, $G'.\mathsf{membKey} := (s_{\text{'memb'}}, u_{\text{'memb'}})$ and $G'.\mathsf{initSecret} := (s_{\text{'init'}}, u_{\text{'init'}})$. ($L_{\text{'epoch'}}$ is also updated accordingly.)
- If $\mathsf{Rand}[\mathsf{id}] = \text{'bad'}$, $\mathcal{S}'_{4\text{-}2}$ queries $s_{\text{'joi'}} := \mathtt{Corr}(u_{\text{'joi'}})$ and encrypts $s_{\text{'joi'}}$ as in the previous hybrid. Then, $\mathcal{S}'_{4\text{-}2}$ computes $s_{\mathsf{lbl}} := \mathsf{RO}(s_{\text{'joi'}}, (G'.\mathsf{groupCont}(), \mathsf{lbl}))$ for each $\mathsf{lbl} \in \{\text{'conf'}, \text{'app'}, \text{'memb'}, \text{'init'}\}$ and updates the epoch secrets as $G'.\mathsf{joinerSecret} := (s_{\text{'joi'}}, u_{\text{'joi'}})$, $G'.\mathsf{confKey} := (s_{\text{'conf'}}, u_{\text{'conf'}})$, $G'.\mathsf{appSecret} := (s_{\text{'app'}}, u_{\text{'app'}})$, $G'.\mathsf{membKey} := (s_{\text{'memb'}}, u_{\text{'memb'}})$ and $G'.\mathsf{initSecret} := (s_{\text{'init'}}, u_{\text{'init'}})$. ($L_{\text{'epoch'}}$ is also updated accordingly.)

Simulation of id **on input** $(\mathtt{Process}, c_0, \widehat{c}, \vec{p})$**.** $\mathcal{S}'_{4\text{-}2}$ simulates the process protocol as in the previous hybrid, except for the differences shown below:

Firstly, $\mathcal{S}'_{4\text{-}2}$ simulates `*unframe-prop` and `*apply-props` identically when simulating the `Commit` protocol. After the execution of `*apply-props`, $\mathcal{S}'_{4\text{-}2}$ obtains the new membership list and the decryption key of each party. In other words, for all members id, $G.\mathsf{member}[\mathsf{id}].\mathsf{dk}$ is of the form $(*, u_{\mathsf{id}})$ or $(\perp, \perp)$.

During execution of `*apply-rekey`, $\mathcal{S}'_{4\text{-}2}$ decrypts the ciphertext $\mathsf{ct} = (\mathsf{T}, \widehat{\mathsf{ct}})$ in $(c_0, \widehat{c})$ as follows, by using id's current decryption key $(s_{\mathsf{id}}, u_{\mathsf{id}}) = G.\mathsf{member}[\mathsf{id}].\mathsf{dk}$[22]. Let $(*, u_{\mathsf{par\text{-}init}})$ be the initial secret at epoch $\mathsf{Ptr}[\mathsf{id}]$ and let confTag be the confirmation tag in c_0. Note that the confirmation hash $G'.\mathsf{confTransHash}$ for the new epoch is computed before running `*apply-rekey` function.

Case (1): If $s_{\mathsf{id}} \neq \perp$, $\mathcal{S}'_{4\text{-}2}$ simply decrypts ct by itself to obtain the commit secret s_{com}.

Case (2): Else, if $(*, u_{\mathsf{id}}, \mathsf{ct}) \notin L_{\text{'enc'}}$ (i.e., ct can be sent to `CmDec` oracle), $\mathcal{S}'_{4\text{-}2}$ sends $(u_{\mathsf{id}}, \mathsf{ct})$ to `CmDec` oracle to obtain the commit secret s_{com}.

Case (3): Else, if $((s', u'), u_{\mathsf{id}}, \mathsf{ct}) \in L_{\text{'enc'}}$ for some $(s' \neq \perp, u')$, s' is used as the commit secret s_{com} (i.e., $s_{\mathsf{com}} := s'$). $\mathcal{S}'_{4\text{-}2}$ retrieves such s_{com}.

[22] u_{id} is always non-$\perp$ because id uses the decryption key generated by itself: When it joins the group, it fetches the CmPKE key registered by itself (cf. `get-dks` queries to $\mathcal{F}_{\mathrm{KS}}$ (line 16 in the join protocol)); When it updates its state, it replaces the old CmPKE key with a new CmPKE key generated by itself which stored in pendUpd (cf. line 11 in `*apply-props` function) or pendCom array (cf. line 4 in the process protocol).

Case (4)[23]**:** Else, if $((\bot, u'), u_{\mathsf{id}}, \mathsf{ct}) \in L_{\text{'enc'}}$ and $(u_{\mathsf{par\text{-}init}}, (\bot, u'), *, G'.\mathsf{confTransHash}, \mathsf{confTag}') \in L_{\text{'epoch'}}$ exist for some u' and $\mathsf{confTag}'$, then $\mathcal{S}'_{4\text{-}2}$ skips running `*derive-keys`, and verifies the confirmation tag confTag in c_0 by checking $\mathsf{confTag} = \mathsf{confTag}'$.

Case (5)[24]**:** Else, if $((\bot, u'), u_{\mathsf{id}}, \mathsf{ct}) \in L_{\text{'enc'}}$ for some u', $\mathcal{S}'_{4\text{-}2}$ outputs $\bot$. We call this event $\mathsf{E}_{\mathsf{inj\text{-}c\text{-}1}}$. In Proposition 4.4.27, we will prove that the simulator in the previous hybrid also outputs $\bot$ when $\mathsf{E}_{\mathsf{inj\text{-}c\text{-}1}}$ occurs. Thus, $\mathcal{Z}$'s view is indistinguishable.

It remains to explain what $\mathcal{S}'_{4\text{-}2}$ does with the commit secret s_{com} for Cases (1)-(3). The simulator finishes by funning `*derive-keys` and `*vrf-conf-tag` as follows.

- If $(u_{\mathsf{par\text{-}init}}, (s_{\mathsf{com}}, *), *, G'.\mathsf{confTransHash}, \mathsf{confTag}') \in L_{\text{'epoch'}}$ exists for some $\mathsf{confTag}'$, $\mathcal{S}'_{4\text{-}2}$ verifies the confirmation tag by checking $\mathsf{confTag} = \mathsf{confTag}'$ and skips `*derive-keys` and `*vrf-conf-tag`. Note that this case occurs if $\mathcal{S}'_{4\text{-}2}$ derived the epoch secret corresponding to the received commit message c_0.
- Else if $\mathbf{gsd\text{-}exp}(u_{\mathsf{par\text{-}init}}) = \mathtt{false}$, $\mathcal{S}'_{4\text{-}2}$ outputs $\bot$. We call this event $\mathsf{E}_{\mathsf{inj\text{-}c\text{-}2}}$. In Proposition 4.4.28 below, we will prove that the simulator in the previous hybrid also outputs $\bot$ when $\mathsf{E}_{\mathsf{inj\text{-}c\text{-}2}}$ occurs, i.e., $\mathcal{Z}$'s view is not changed.
- Else (i.e., $\mathbf{gsd\text{-}exp}(u_{\mathsf{par\text{-}init}}) = \mathtt{true}$), $\mathcal{S}'_{4\text{-}2}$ knows both $s_{\mathsf{par\text{-}init}}$ and s_{com}. It computes the joiner secret $s_{\text{'joi'}} := \mathsf{RO}(s_{\mathsf{par\text{-}init}}, s_{\mathsf{com}}, \text{'joi'})$ and the epoch secrets $s_{\mathsf{lbl}} := \mathsf{RO}(s_{\text{'joi'}}, (G'.\mathsf{groupCont}(), \mathsf{lbl}))$ for each $\mathsf{lbl} \in \{\text{'conf'}, \text{'app'}, \text{'memb'}, \text{'init'}\}$. Then, $\mathcal{S}'_{4\text{-}2}$ recomputes the confirmation tag $\mathsf{confTag}' := \mathsf{RO}(s_{\text{'conf'}}, G'.\mathsf{confTransHash})$ and checks whether $\mathsf{confTag} = \mathsf{confTag}'$ holds. If true, $\mathcal{S}'_{4\text{-}2}$ prepares new GSD nodes u_{com}, $u_{\text{'joi'}}$, $u_{\text{'conf'}}$, $u_{\text{'app'}}$, $u_{\text{'memb'}}$, and $u_{\text{'init'}}$ (their values are set to $u_{\mathsf{ctr}}, u_{\mathsf{ctr}+1}, \ldots, u_{\mathsf{ctr}+5}$ and ctr is incremented as $\mathsf{ctr} \leftarrow \mathsf{ctr} + 6$) and queries $\mathtt{Set\text{-}Secret}(u_{\mathsf{com}}, s_{\mathsf{com}})$, $\mathtt{Join\text{-}Hash}(u_{\mathsf{par\text{-}init}}, u_{\mathsf{com}}, u_{\text{'joi'}}, \text{'joi'})$ and $\mathtt{Hash}(u_{\text{'joi'}}, u_{\mathsf{lbl}}, (G'.\mathsf{groupCont}(), \mathsf{lbl}))$ for each $\mathsf{lbl} \in \{\text{'conf'}, \text{'app'}, \text{'memb'}, \text{'init'}\}$. $\mathcal{S}'_{4\text{-}2}$ sets $G'.\mathsf{comSecret} := (s_{\mathsf{com}}, u_{\mathsf{com}})$, $G'.\mathsf{joinerSecret} := (s_{\text{'joi'}}, u_{\text{'joi'}})$, $G'.\mathsf{confKey} := (s_{\text{'conf'}}, u_{\text{'conf'}})$, $G'.\mathsf{appSecret} := (s_{\text{'app'}}, u_{\text{'app'}})$, $G'.\mathsf{membKey} := (s_{\text{'memb'}}, u_{\text{'memb'}})$ and $G'.\mathsf{initSecret} := (s_{\text{'init'}}, u_{\text{'init'}})$. Finally, $\mathcal{S}'_{4\text{-}2}$ stores $L_{\text{'epoch'}} \mathrel{+}\!\leftarrow (u_{\mathsf{par\text{-}init}}, G'.\mathsf{comSecret}, G'.\mathsf{joinerSecret}, G'.\mathsf{confTransHash}, \mathsf{confTag})$.

Simulation of id **on input** $(\mathtt{Join}, w_0, \widehat{w})$**.** $\mathcal{S}'_{4\text{-}2}$ simulates the join protocol as in the previous hybrid, except for the differences shown below:

When initializing the group membership list, for each encryption key in $G.\mathsf{member}$, if $\mathcal{S}'_{4\text{-}2}$ knows the corresponding decryption key, $\mathcal{S}'_{4\text{-}2}$ copies it to $G.\mathsf{member}$. Otherwise, the decryption key is set to $(\bot, \bot)$.

$\mathcal{S}'_{4\text{-}2}$ decrypts the ciphertext $\mathsf{ct} = (\mathsf{T}, \widehat{\mathsf{ct}})$ in $(w_0, \widehat{w})$ using id's decryption key $(s_{\mathsf{id}}, u_{\mathsf{id}}) = G.\mathsf{member}[\mathsf{id}].\mathsf{dk}$[25] as follows. Let $G.\mathsf{confTransHash}$ be the confirmation hash and confTag be the confirmation tag included in w_0.

Case (1): If $s_{\mathsf{id}} \neq \bot$, $\mathcal{S}'_{4\text{-}2}$ simply decrypts ct by itself to obtain the joiner secret $s_{\text{'joi'}}$.

Case (2): Else, if $(*, u_{\mathsf{id}}, \mathsf{ct}) \notin L_{\text{'enc'}}$ (i.e., ct can be sent to `CmDec` oracle), $\mathcal{S}'_{4\text{-}2}$ sends $(u_{\mathsf{id}}, \mathsf{ct})$ to obtain the joiner secret $s_{\text{'joi'}}$.

[23]This case occurs if the received commit message is generated by the commit protocol with good randomness, and the encrypted commit secret has not been leaked.

[24]This case occurs when the ciphertext was generated by `CmEnc` oracle, but the simulator did not derive the corresponding epoch secret. Put differently, this occurs when $\mathcal{Z}$ generates a malicious welcome message using an honestly generated ciphertext.

[25]u_{id} is always non-$\bot$ because when a party id joins the group, id uses the decryption key generated by itself (cf. `get-dks` queries to $\mathcal{F}_{\mathrm{KS}}$ (line 16 in the join protocol)).

Case (3): Else, if $((s', u'), u_{\mathsf{id}}, \mathsf{ct}) \in L_{\text{`enc'}}$ for some $(s' \neq \bot, u')$, $s'$ is used as the joiner secret $s_{\text{`joi'}}$ (i.e., $s_{\text{`joi'}} := s'$). $\mathcal{S}'_{4\text{-}2}$ retrieves such $s_{\text{`joi'}}$.

Case (4)[26]**:** Else, if $((\bot, u'), u_{\mathsf{id}}, \mathsf{ct}) \in L_{\text{`enc'}}$ and $(*, *, (\bot, u'), G.\mathsf{confTransHash}, \mathsf{confTag}') \in L_{\text{`epoch'}}$ exist for some u' and $\mathsf{confTag}'$, then $\mathcal{S}'_{4\text{-}2}$ skips the derivation of the epoch secrets, and verifies the confirmation tag confTag in w_0 by checking $\mathsf{confTag} = \mathsf{confTag}'$.

Case (5)[27]**:** If $((\bot, u'), u_{\mathsf{id}}, \mathsf{ct}) \in L_{\text{`enc'}}$ for some u', $\mathcal{S}'_{4\text{-}2}$ outputs $\bot$. We call this event $\mathsf{E}_{\mathsf{inj\text{-}w}}$. In Proposition 4.4.29, we will prove that the simulator in the previous hybrid also outputs $\bot$ when $\mathsf{E}_{\mathsf{inj\text{-}w}}$ occurs, i.e., $\mathcal{Z}$'s view is indistinguishable.

It remains to explain what $\mathcal{S}'_{4\text{-}2}$ does with the commit secret $s_{\text{`joi'}}$ for Cases (1)-(3). The simulator verifies confTag in w_0 as follows.

- If $(*, *, (s_{\text{`joi'}}, *), G.\mathsf{confTransHash}, \mathsf{confTag}') \in L_{\text{`epoch'}}$ exists for some $\mathsf{confTag}'$, $\mathcal{S}'_{4\text{-}2}$ verifies the confirmation tag by checking $\mathsf{confTag} = \mathsf{confTag}'$.
- Else, $\mathcal{S}'_{4\text{-}2}$ computes $s_{\mathsf{lbl}} := \mathsf{RO}(s_{\text{`joi'}}, (G.\mathsf{groupCont}(), \mathsf{lbl}))$ for each $\mathsf{lbl} \in \{\text{'conf', 'app', 'memb', 'init'}\}$, and verifies the confirmation tag by checking $\mathsf{confTag} = \mathsf{RO}(s_{\text{`conf'}}, G.\mathsf{confTransHash})$. If the tag is valid, $\mathcal{S}'_{4\text{-}2}$ prepares new GSD nodes $u_{\text{`joi'}}$, $u_{\text{`conf'}}$, $u_{\text{`app'}}$, $u_{\text{`memb'}}$, and $u_{\text{`init'}}$ (their values are set to $u_{\mathsf{ctr}}$, $u_{\mathsf{ctr}+1}, \ldots, u_{\mathsf{ctr}+4}$ and ctr is incremented as $\mathsf{ctr} \leftarrow \mathsf{ctr} + 5$) and queries $\underline{\mathtt{Set\text{-}Secret}(u_{\text{`joi'}}, s_{\text{`joi'}})}$ and $\mathtt{Hash}(u_{\text{`joi'}}, u_{\mathsf{lbl}}, (G.\mathsf{groupCont}(), \mathsf{lbl}))$ for each $\mathsf{lbl} \in \{\text{'conf', 'app', 'memb', 'init'}\}$. Then, $\mathcal{S}'_{4\text{-}2}$ sets $G.\mathsf{joinerSecret} := (s_{\text{`joi'}}, u_{\text{`joi'}})$, $G.\mathsf{confKey} := (s_{\text{`conf'}}, u_{\text{`conf'}})$, $G.\mathsf{appSecret} := (s_{\text{`app'}}, u_{\text{`app'}})$, $G.\mathsf{membKey} := (s_{\text{`memb'}}, u_{\text{`memb'}})$ and $G.\mathsf{initSecret} := (s_{\text{`init'}}, u_{\text{`init'}})$. Finally, $\mathcal{S}'_{4\text{-}2}$ stores $L_{\text{`epoch'}} \xleftarrow{+} (\bot, (\bot, \bot), G.\mathsf{joinerSecret}, G.\mathsf{confTransHash}, \mathsf{confTag})$.

Simulation of id **on input** `Key`**.** Let $G.\mathsf{appSecret}$ be the application secret at epoch $\mathsf{Ptr}[\mathsf{id}]$. If it is of the from $(s_{\text{`app'}} \neq \bot, u_{\text{`app'}})$, $\mathcal{S}'_{4\text{-}2}$ returns $s_{\text{`app'}}$. Else, $\mathcal{S}'_{4\text{-}2}$ queries $s_{\text{`app'}} := \mathtt{Corr}(u_{\text{`app'}})$ and returns $s_{\text{`app'}}$; Then $\mathcal{S}'_{4\text{-}2}$ replaces $(\bot, u_{\text{`app'}})$ with $(s_{\text{`app'}}, u_{\text{`app'}})$

Corruption query for id**.** id's state at epoch $\mathsf{Ptr}[\mathsf{id}]$ contains the following secrets:

- The decryption key stored in $G.\mathsf{member}[\mathsf{id}].\mathsf{dk}$ and $G.\mathsf{pendUpd}$ array;
- The decryption keys of the key packages registered to Key Service. They are stored in $\mathsf{DK}[\mathsf{id}, *]$; and
- The current epoch secrets confKey, membKey, appSecret, and initSecret. Note that id holds appSecret only if `Key` query has not been queried to $\mathsf{Ptr}[\mathsf{id}]$.
- The epoch secrets stored in $G.\mathsf{pendCom}$ array;

For each of the above secrets, if the secret is of the form $(\bot, u)$, $\mathcal{S}'_{4\text{-}2}$ queries $\underline{s := \mathtt{Corr}(u)}$ and replaces $(\bot, u)$ with (s, u). Then, for each node v such that **gsd-exp**(v) becomes true due to the corruption, $\mathcal{S}'_{4\text{-}2}$ computes the secret s_v from previously obtained ciphertexts and corrupted secrets, and replaces all occurrence of $(\bot, v)$ with (s_v, v). Finally, $\mathcal{S}'_{4\text{-}2}$ returns id's secrets listed above to the adversary $\mathcal{A}$.

Part 2: Indistinguishability of the Two Hybrid. We next prove indistinguishability between Hybrid 4-1 and Hybrid 4-2.

[26]This case occurs if the received welcome message is generated by the commit protocol with good randomness, and the encrypted joiner secret has not been leaked.

[27]This case occurs when the ciphertext was generated by `CmEnc` oracle, but the simulator did not derive the corresponding epoch secret. Put differently, it occurs when $\mathcal{Z}$ generates a malicious commit message using an honestly generated ciphertext.

Lemma 4.4.25. *Hybrid 4-1 and Hybrid 4-2 are indistinguishable assuming* CmPKE *is* Chained CmPKE *conforming GSD secure.*

Proof. The difference from the previous hybrid is that the simulator $\mathcal{S}'_{4\text{-}2}$ always outputs $\perp$ when events $\mathsf{E}_{\text{inj-p}}$, $\mathsf{E}_{\text{inj-c-1}}$, $\mathsf{E}_{\text{inj-c-2}}$, and $\mathsf{E}_{\text{inj-w}}$ occur, and it aborts when event $\mathsf{abort}_{\mathsf{attach}}$ occurs. In the following, we show that, if $\mathcal{Z}$ can distinguish the two hybrids (i.e., $\mathcal{S}'_{4\text{-}2}$ provides a different view to $\mathcal{Z}$ when the events occur), then there exists an adversary $\mathcal{B}$ that can win the Chained CmPKE conforming GSD game.

We first show that $\mathcal{Z}$'s view is not changed when $\mathsf{E}_{\text{inj-p}}$ occurs. In other words, $\mathcal{Z}$ cannot inject a proposal message without knowing the membership key.

Proposition 4.4.26. *$\mathcal{Z}$'s view when $\mathsf{E}_{\text{inj-p}}$ occurs is indistinguishable between the two hybrids if* CmPKE *is* Chained CmPKE *conforming GSD secure.*

Proof. The difference from the previous hybrid is that $\mathcal{S}'_{4\text{-}2}$ outputs $\perp$ when event $\mathsf{E}_{\text{inj-p}}$ occurs. We show that, if $\mathcal{Z}$ can distinguish the two hybrids, then there exists an adversary $\mathcal{B}$ that can win the Chained CmPKE conforming GSD game. We first explain the description of $\mathcal{B}$ and how $\mathcal{B}$ embeds the GSD challenge. We then show the validity of the GSD challenge and finally evaluate $\mathcal{B}$'s advantage.

$\mathcal{B}$ simulates for $\mathcal{Z}$ the interaction with $\mathcal{F}_{\text{AS}}^{\text{IW}}$ and $\mathcal{F}_{\text{KS}}^{\text{IW}}$, and the protocol execution of each party as in $\mathcal{S}'_{4\text{-}2}$, except that $\mathcal{B}$ interacts with its GSD challenger instead of $\mathcal{S}_{\text{GSD}}$. We assume $\mathcal{B}$ receives at most N distinct proposal messages as input to the commit and process protocol. At the beginning of the game, $\mathcal{B}$ chooses $i \in [N]$ at random and hopes that the i-th proposal message triggers $\mathsf{E}_{\text{inj-p}}$, and the simulator in the previous hybrid outputs non-$\perp$ when it receives the message. ($\mathcal{B}$ succeeds to guess with probability $1/N$.) When $\mathcal{B}$ receives the i-th proposal message p, $\mathcal{B}$ embeds the GSD challenge and determines the challenge bit as follows. We assume p is processed by id without loss of generality, and let $u_{\mathsf{memb}'}$ be the GSD node assigned to the membership key at epoch $\mathsf{Ptr}[\mathsf{id}]$. Also, let $\mathsf{membTag}$ be the membership tag in p.

1. $\mathcal{B}$ prepares a new GSD node $u_{\mathsf{mtag}} := u_{\mathsf{ctr}}$ and queries $\mathtt{Hash}(u_{\mathsf{memb}'}, u_{\mathsf{mtag}}, (\mathsf{propCont}, \mathsf{sig}))$, where $(\mathsf{propCont}, \mathsf{sig})$ is taken from p.
2. $\mathcal{B}$ queries $\mathsf{membTag}' := \mathtt{Chall}(u_{\mathsf{mtag}})$
3. If $\mathsf{membTag} = \mathsf{membTag}'$, $\mathcal{B}$ submits 0 to the GSD challenger; otherwise submits 1.

Note that $\mathcal{B}$ can send $(\mathsf{propCont}, \mathsf{sig})$ to `Hash` oracle (that is, $\mathcal{B}$ has not queries $(\mathsf{propCont}, \mathsf{sig})$ to `Hash` oracle before) because $(u_{\mathsf{memb}'}, (\mathsf{propCont}, \mathsf{sig}), *) \notin L_{\mathsf{memb}'}$ when $\mathsf{E}_{\text{inj-p}}$ occurs. If $\mathcal{B}$ succeeds the guess, $\mathsf{membTag}$ in p is valid, i.e., it satisfies $\mathsf{membTag} = \mathsf{RO}(s_{\mathsf{memb}'}, (\mathsf{propCont}, \mathsf{sig}))$ for the membership key $s_{\mathsf{memb}'}$ assigned to $u_{\mathsf{memb}'}$. Moreover, $\mathcal{B}$ assigns $\mathsf{RO}(s_{\mathsf{memb}'}, (\mathsf{propCont}, \mathsf{sig}))$ to u_{mtag}. Thus, if the challenge oracle returns the real value (i.e., the challenge bit is 0), $\mathsf{membTag} = \mathsf{membTag}'$ holds with probability 1; otherwise with negligible probability. Therefore, $\mathcal{B}$ can output the correct challenge bit with overwhelming probability.

We then check the validity of the GSD challenge. Observe that the GSD graph is acyclic and u_{mtag} is a sink node. In addition, by the structure of the GSD graph, we have

$$\begin{aligned}\textbf{gsd-exp}(u_{\mathsf{mtag}}) &= \big(u_{\mathsf{mtag}} \in \mathsf{Corr}\big) \vee \textbf{gsd-exp}(u_{\mathsf{memb}'}) \\ &= \mathtt{false}.\end{aligned}$$

This is because u_{mtag} is not sent to `Corr` or `Set-Secret` oracle, and $\textbf{gsd-exp}(u_{\mathsf{memb}'}) = \mathtt{false}$ when $\mathsf{E}_{\text{inj-p}}$ occurs. Hence, u_{mtag} is a valid challenge node, and $\mathcal{B}$ wins the GSD game.

We finally evaluate the advantage of $\mathcal{B}$. $\mathcal{B}$ wins the GSD game if $\mathcal{Z}$ distinguishes the hybrids and $\mathcal{B}$ succeeds to guess the proposal message. If $\mathcal{Z}$ distinguishes the two hybrids with non-negligible probability

ϵ, $\mathcal{B}$ wins the game with probability ϵ/Q, which is also non-negligible. This contradicts the assumption that CmPKE is Chained CmPKE conforming GSD secure. Therefore, ϵ must be negligible, and we conclude that $\mathcal{Z}$'s view when $\mathsf{E}_{\text{inj-p}}$ occurs is indistinguishable between Hybrid 4-1 and Hybrid 4-2. □

We then prove that $\mathcal{Z}$'s view is not changed when $\mathsf{E}_{\text{inj-c-1}}$ occurs. In other words, $\mathcal{Z}$ cannot inject a commit message without knowing the encrypted commit secret.

Proposition 4.4.27. *$\mathcal{Z}$'s view when $\mathsf{E}_{\text{inj-c-1}}$ occurs is indistinguishable between the two hybrids if* CmPKE *is* Chained CmPKE *conforming GSD secure.*

Proof. The difference from the previous hybrid is that $\mathcal{S}'_{4\text{-}2}$ outputs $\perp$ when event $\mathsf{E}_{\text{inj-c-1}}$ occurs. We show that, if $\mathcal{Z}$ can distinguish the two hybrids, then there exists an adversary $\mathcal{B}$ that can win the Chained CmPKE conforming GSD game. We first explain the description of $\mathcal{B}$ and how $\mathcal{B}$ embeds the GSD challenge. We then show the validity of the GSD challenge and finally evaluate $\mathcal{B}$'s advantage.

$\mathcal{B}$ simulates for $\mathcal{Z}$ the interaction with $\mathcal{F}_{\text{AS}}^{\text{IW}}$ and $\mathcal{F}_{\text{KS}}^{\text{IW}}$, and the protocol execution of each party as in $\mathcal{S}'_{4\text{-}2}$, except that $\mathcal{B}$ interacts with its GSD challenger instead of $\mathcal{S}_{\text{GSD}}$. We assume $\mathcal{B}$ receives at most Q distinct commit messages as input to the process protocol. At the beginning of the game, $\mathcal{B}$ chooses $i \in [Q]$ at random and hopes that the i-th commit message triggers $\mathsf{E}_{\text{inj-c-1}}$, and the simulator in the previous hybrid outputs non-$\perp$ when it receives the message. ($\mathcal{B}$ succeeds to guess with probability $1/Q$.) When $\mathcal{B}$ receives the i-th commit message c_0, $\mathcal{B}$ embeds the GSD challenge and determines the challenge bit as follows. We assume c_0 is processed by id without loss of generality and let $u_{\text{par-init}}$ be the GSD node assigned to the initial secret at epoch $\mathsf{Ptr}[\mathsf{id}]$. Let confTag be the confirmation tag in c_0. In addition, when $\mathsf{E}_{\text{inj-c-1}}$ occurs, there exists a node u' such that $((\perp, u'), u_{\mathsf{id}}, \mathsf{ct}) \in L_{\text{`enc'}}$, where u_{id} is the GSD node corresponding to id's current CmPKE key and ct is the ciphertext in c_0

1. $\mathcal{B}$ prepares new GSD nodes $u_{\text{`joi'}}$ and $u_{\text{`conf'}}$ (their values are set to u_{ctr} and $u_{\mathsf{ctr}+1}$, and ctr is incremented as $\mathsf{ctr} \leftarrow \mathsf{ctr} + 2$) and queries $\texttt{Join-Hash}(u_{\text{par-init}}, u', u_{\text{`joi'}}, \text{`joi'})$ and $\texttt{Hash}(u_{\text{`joi'}}, u_{\text{`conf'}}, (G'.\mathsf{groupCont}(), \text{`conf'}))$.
2. $\mathcal{B}$ prepares a new GSD node $u_{\mathsf{ctag}} := u_{\mathsf{ctr}}$ and queries $\texttt{Hash}(u_{\text{`conf'}}, u_{\mathsf{ctag}}, G'.\mathsf{confTransHash})$.
3. $\mathcal{B}$ queries $\mathsf{confTag}' := \texttt{Chall}(u_{\mathsf{ctag}})$.
4. If $\mathsf{confTag} = \mathsf{confTag}'$, $\mathcal{B}$ submits 0 to the GSD challenger; otherwise submits 1.

Note that $\mathcal{B}$ can issue the above queries to oracles Hash and Join-Hash because the inputs are new GSD nodes. Note also that the confirmation key derived from the current initial secret and c_0 is correctly computed to $u_{\text{`conf'}}$ because the commit secret encrypted in ct is assigned to $u'$. If $\mathcal{B}$ succeeds the guess, confTag in $c_0$ is valid, i.e., it satisfies $\mathsf{confTag} = \mathsf{RO}(s_{\text{`conf'}}, G'.\mathsf{confTransHash})$ for the confirmation key $s_{\text{`conf'}}$ assigned to $u_{\text{`conf'}}$. Moreover, $\mathcal{B}$ assigns $\mathsf{RO}(s_{\text{`conf'}}, G'.\mathsf{confTransHash})$ to u_{ctag}. Thus, if the challenge oracle returns the real value (i.e., the challenge bit is 0), $\mathsf{confTag} = \mathsf{confTag}'$ holds with probability 1; otherwise with negligible probability. Therefore, $\mathcal{B}$ can output the correct challenge bit with overwhelming probability.

We check the validity of the GSD challenge. Observe that the GSD graph is acyclic and u_{ctag} is a sink node. In addition, by the structure of the GSD graph, we have

$$\begin{aligned}\textbf{gsd-exp}(u_{\mathsf{ctag}}) &= \big(u_{\mathsf{ctag}} \in \mathsf{Corr}\big) \vee \textbf{gsd-exp}(u_{\text{`conf'}}) \\ &= \big(u_{\mathsf{ctag}} \in \mathsf{Corr}\big) \vee \big(u_{\text{`conf'}} \in \mathsf{Corr}\big) \vee \textbf{gsd-exp}(u_{\text{`joi'}}) \\ &= \big(u_{\mathsf{ctag}} \in \mathsf{Corr}\big) \vee \big(u_{\text{`conf'}} \in \mathsf{Corr}\big) \vee \big(u_{\text{`joi'}} \in \mathsf{Corr}\big) \vee\end{aligned}$$

$$(\textbf{gsd-exp}(u_{\mathsf{par\text{-}init}}) \wedge \textbf{gsd-exp}(u')) = \texttt{false}.$$

This is because u_{ctag}, $u_{\text{'conf'}}$, and $u_{\text{'joi'}}$ are not sent to `Corr` or `Set-Secret` oracle, and $\textbf{gsd-exp}(u') = \texttt{false}$ when $\mathsf{E}_{\mathsf{inj\text{-}c\text{-}1}}$ occurs. Hence, u_{ctag} is a valid challenge node, and $\mathcal{B}$ wins the GSD game.

We finally evaluate the advantage of $\mathcal{B}$. $\mathcal{B}$ can win the GSD game if $\mathcal{Z}$ distinguishes the hybrids and $\mathcal{B}$ succeeds to guess the message. If $\mathcal{Z}$ distinguishes the two hybrids with non-negligible probability ϵ, $\mathcal{B}$ wins the game with probability ϵ/Q, which is also non-negligible. This contradicts the assumption that CmPKE is Chained CmPKE conforming GSD secure. Therefore, ϵ must be negligible, and we conclude that $\mathcal{Z}$'s view when $\mathsf{E}_{\mathsf{inj\text{-}c\text{-}1}}$ occurs is indistinguishable between Hybrid 4-1 and Hybrid 4-2. □

We also prove that that $\mathcal{Z}$'s view is not changed when $\mathsf{E}_{\mathsf{inj\text{-}c\text{-}2}}$ occurs. In other words, $\mathcal{Z}$ cannot inject a commit message without knowing the initial secret of the parent node.

Proposition 4.4.28. *$\mathcal{Z}$'s view when $\mathsf{E}_{\mathsf{inj\text{-}c\text{-}2}}$ occurs is indistinguishable between the two hybrids if* CmPKE *is* Chained CmPKE *conforming GSD secure.*

Proof. The difference from the previous hybrid is that $\mathcal{S}'_{4\text{-}2}$ outputs $\perp$ when event $\mathsf{E}_{\mathsf{inj\text{-}c\text{-}2}}$ occurs. We show that, if $\mathcal{Z}$ can distinguish the two hybrids, then there exists an adversary $\mathcal{B}$ that can win the Chained CmPKE conforming GSD game. We first explain the description of $\mathcal{B}$ and how $\mathcal{B}$ embeds the GSD challenge. We then show the validity of the GSD challenge and finally evaluate $\mathcal{B}$'s advantage.

$\mathcal{B}$ simulates for $\mathcal{Z}$ the interaction with $\mathcal{F}^{\mathrm{IW}}_{\mathrm{AS}}$ and $\mathcal{F}^{\mathrm{IW}}_{\mathrm{KS}}$, and the protocol execution of each party as in $\mathcal{S}'_{4\text{-}2}$, except that $\mathcal{B}$ interacts with its GSD challenger instead of $\mathcal{S}_{\mathsf{GSD}}$. We assume $\mathcal{B}$ receives at most Q distinct commit messages as input to the process protocol. At the beginning of the game, $\mathcal{B}$ chooses $i \in [Q]$ at random and hopes that the i-th commit message triggers $\mathsf{E}_{\mathsf{inj\text{-}c\text{-}2}}$, and the simulator in the previous hybrid outputs non-$\perp$ when it receives the message. ($\mathcal{B}$ succeeds to guess with probability $1/Q$.) When $\mathcal{B}$ receives the i-th commit message c_0, $\mathcal{B}$ embeds the GSD challenge and determines the challenge bit as follows. We assume c_0 is processed by id without loss of generality and let $u_{\mathsf{par\text{-}init}}$ be the GSD node assigned to the initial secret at epoch $\mathsf{Ptr}[\mathsf{id}]$. Let confTag be the confirmation tag in c_0. Note that when $\mathsf{E}_{\mathsf{inj\text{-}c\text{-}2}}$ occurs, $\mathcal{B}$ succeeds to decrypt the commit secret s_{com}.

1. $\mathcal{B}$ prepares new GSD nodes u_{com}, $u_{\text{'joi'}}$ and $u_{\text{'conf'}}$ (their values are set to u_{ctr}, $u_{\mathsf{ctr}+1}$ and $u_{\mathsf{ctr}+2}$. ctr is incremented as $\mathsf{ctr} \leftarrow \mathsf{ctr}+3$) and queries $\texttt{Set-Secret}(u_{\mathsf{com}}, s_{\mathsf{com}})$, $\texttt{Join-Hash}(u_{\mathsf{par\text{-}init}}, u_{\mathsf{com}}, u_{\text{'joi'}}, \text{'joi'})$ and $\texttt{Hash}(u_{\text{'joi'}}, u_{\text{'conf'}}, (G'.\mathsf{groupCont}(), \text{'conf'}))$.
2. $\mathcal{B}$ prepares a new GSD node $u_{\mathsf{ctag}} := u_{\mathsf{ctr}}$ and queries $\texttt{Hash}(u_{\text{'conf'}}, u_{\mathsf{ctag}}, G'.\mathsf{confTransHash})$.
3. $\mathcal{B}$ queries $\mathsf{confTag}' := \texttt{Chall}(u_{\mathsf{ctag}})$.
4. If $\mathsf{confTag} = \mathsf{confTag}'$, $\mathcal{B}$ submits 0 to the GSD challenger; otherwise submits 1.

Note that $\mathcal{B}$ can issue the above queries because $\mathcal{B}$ prepares new GSD nodes. Note also that the confirmation key derived from the current initial secret and c_0 is correctly computed to $u_{\text{'conf'}}$. If $\mathcal{B}$ succeeds to guess, confTag in c_0 is valid, i.e., it satisfies $\mathsf{confTag} = \mathsf{RO}(s_{\text{'conf'}}, G'.\mathsf{confTransHash})$ for the confirmation key $s_{\text{'conf'}}$ assigned to $u_{\text{'conf'}}$. Moreover, $\mathcal{B}$ assigns $\mathsf{RO}(s_{\text{'conf'}}, G'.\mathsf{confTransHash})$ to u_{ctag}. Thus, if the challenge oracle returns the real value (i.e., the challenge bit is 0), $\mathsf{confTag} = \mathsf{confTag}'$ holds with probability 1; otherwise with negligible probability. Therefore, $\mathcal{B}$ can output the correct challenge bit with overwhelming probability.

We check the validity of the GSD challenge. Observe that the GSD graph is acyclic and u_{ctag} is a sink node. In addition, by the structure of the GSD graph, we have

$$\begin{aligned}\mathbf{gsd\text{-}exp}(u_{\mathsf{ctag}}) &= \big(u_{\mathsf{ctag}} \in \mathsf{Corr}\big) \vee \mathbf{gsd\text{-}exp}(u_{\text{`}\mathsf{conf}'}) \\ &= \big(u_{\mathsf{ctag}} \in \mathsf{Corr}\big) \vee \big(u_{\text{`}\mathsf{conf}'} \in \mathsf{Corr}\big) \vee \mathbf{gsd\text{-}exp}(u_{\text{`}\mathsf{joi}'}) \\ &= \big(u_{\mathsf{ctag}} \in \mathsf{Corr}\big) \vee \big(u_{\text{`}\mathsf{conf}'} \in \mathsf{Corr}\big) \vee \big(u_{\text{`}\mathsf{joi}'} \in \mathsf{Corr}\big) \vee \\ &\quad \big(\mathbf{gsd\text{-}exp}(u_{\mathsf{par\text{-}init}}) \wedge \mathbf{gsd\text{-}exp}(u_{\mathsf{com}})\big) \\ &= \mathtt{false}.\end{aligned}$$

This is because u_{ctag}, $u_{\text{`}\mathsf{conf}'}$, $u_{\text{`}\mathsf{joi}'}$ are not sent to `Corr` or `Set-Secret` oracle, and $\mathbf{gsd\text{-}exp}(u_{\mathsf{par\text{-}init}}) = \mathtt{false}$ when $\mathsf{E}_{\mathsf{inj\text{-}c}}$ occurs. Hence, u_{ctag} is a valid challenge node, and $\mathcal{B}$ wins the GSD game.

We finally evaluate the advantage of $\mathcal{B}$. $\mathcal{B}$ can win the GSD game if $\mathcal{Z}$ distinguishes the hybrids and $\mathcal{B}$ succeeds to guess the message. If $\mathcal{Z}$ distinguishes the two hybrids with non-negligible probability ϵ, $\mathcal{B}$ wins the game with probability ϵ/Q, which is also non-negligible. This contradicts the assumption that CmPKE is Chained CmPKE conforming GSD secure. Therefore, ϵ must be negligible, and we conclude that $\mathcal{Z}$'s view when $\mathsf{E}_{\mathsf{inj\text{-}c\text{-}2}}$ occurs is indistinguishable between Hybrid 4-1 and Hybrid 4-2. □

We then prove that $\mathcal{Z}$'s view is not changed when $\mathsf{E}_{\mathsf{inj\text{-}w}}$ occurs. In other words, $\mathcal{Z}$ cannot inject a welcome message without knowing the encrypted joiner secret.

Proposition 4.4.29. *$\mathcal{Z}$'s view when $\mathsf{E}_{\mathsf{inj\text{-}w}}$ occurs is indistinguishable between the two hybrids if* CmPKE *is* Chained CmPKE *conforming GSD secure.*

Proof. The difference from the previous hybrid is that $\mathcal{S}'_{4\text{-}2}$ outputs $\perp$ when event $\mathsf{E}_{\mathsf{inj\text{-}w}}$ occurs. We show that, if $\mathcal{Z}$ can distinguish the two hybrids, then there exists an adversary $\mathcal{B}$ that can win the Chained CmPKE conforming GSD game. We first explain the description of $\mathcal{B}$ and how $\mathcal{B}$ embeds the GSD challenge. We then show the validity of the GSD challenge and finally evaluate $\mathcal{B}$'s advantage.

$\mathcal{B}$ simulates for $\mathcal{Z}$ the interaction with $\mathcal{F}^{\mathrm{IW}}_{\mathrm{AS}}$ and $\mathcal{F}^{\mathrm{IW}}_{\mathrm{KS}}$, and the protocol execution of each party as in $\mathcal{S}'_{4\text{-}2}$, except that $\mathcal{B}$ interacts with its GSD challenger instead of $\mathcal{S}_{\mathsf{GSD}}$. We assume $\mathcal{B}$ receives at most W distinct welcome messages as input to the join protocol. At the beginning of the game, $\mathcal{B}$ chooses $i \in [W]$ at random, and hopes that the i-th welcome message triggers $\mathsf{E}_{\mathsf{inj\text{-}w}}$, and the simulator in the previous hybrid outputs non-$\perp$ when it receives the message (i.e., the message is valid). ($\mathcal{B}$ succeeds to guess with probability $1/W$.) When $\mathcal{B}$ receives the i-th welcome message w_0, $\mathcal{B}$ embeds the GSD challenge and determines the challenge bit as follows. We assume w_0 is processed by id without loss of generality and let confTag be the confirmation tag in w_0. In addition, when $\mathsf{E}_{\mathsf{inj\text{-}w}}$ occurs, there exists a node u' such that $((\perp, u'), u_{\mathsf{id}}, \mathsf{ct}) \in L_{\text{`}\mathsf{enc}'}$, where u_{id} is the GSD node corresponding to id's current CmPKE key and ct is the ciphertext in w_0

1. $\mathcal{B}$ prepares new GSD nodes $u_{\text{`}\mathsf{conf}'} := u_{\mathsf{ctr}}$ (ctr is incremented) and queries $\mathtt{Hash}(u', u_{\text{`}\mathsf{conf}'}, (G.\mathsf{groupCont}(), \text{`}\mathsf{conf}'))$.
2. $\mathcal{B}$ prepares a new GSD node $u_{\mathsf{ctag}} := u_{\mathsf{ctr}}$ and queries $\mathtt{Hash}(u_{\text{`}\mathsf{conf}'}, u_{\mathsf{ctag}}, G'.\mathsf{confTransHash})$.
3. $\mathcal{B}$ queries $\mathsf{confTag}' := \mathtt{Chall}(u_{\mathsf{ctag}})$.
4. If $\mathsf{confTag} = \mathsf{confTag}'$, $\mathcal{B}$ submits 0; otherwise submits 1.

Note that $\mathcal{B}$ can issue the above queries because $\mathcal{B}$ prepares new GSD nodes. Note also that the confirmation key derived from w_0 is correctly computed to $u_{\text{'conf'}}$ because the joiner secret encrypted in ct is assigned to u'. If $\mathcal{B}$ succeeds the guess, confTag in w_0 is valid, i.e., it satisfies $\mathsf{confTag} = \mathsf{RO}(s_{\text{'conf'}}, G.\mathsf{confTransHash})$ for the confirmation key $s_{\text{'conf'}}$ assigned to $u_{\text{'conf'}}$. Moreover, $\mathcal{B}$ assigns $\mathsf{RO}(s_{\text{'conf'}}, G.\mathsf{confTransHash})$ to u_{ctag}. Thus, if the challenge oracle returns the real value (i.e., the challenge bit is 0), $\mathsf{confTag} = \mathsf{confTag}'$ holds with probability 1; otherwise with negligible probability. Therefore, $\mathcal{B}$ can output the correct challenge bit with overwhelming probability.

We check the validity of the GSD challenge. Observe that the GSD graph is acyclic and u_{ctag} is a sink node. In addition, we have

$$\begin{aligned}\textbf{gsd-exp}(u_{\mathsf{ctag}}) &= \big(u_{\mathsf{ctag}} \in \mathsf{Corr}\big) \vee \textbf{gsd-exp}(u_{\text{'conf'}}) \\ &= \big(u_{\mathsf{ctag}} \in \mathsf{Corr}\big) \vee \big(u_{\text{'conf'}} \in \mathsf{Corr}\big) \vee \textbf{gsd-exp}(u') \\ &= \texttt{false}.\end{aligned}$$

This is because u_{ctag} and $u_{\text{'conf'}}$ are not sent to `Corr` or `Set-Secret` oracle, and $\textbf{gsd-exp}(u') = \texttt{false}$ when $\mathsf{E}_{\mathsf{inj\text{-}w}}$ occurs. Hence, u_{ctag} is a valid challenge node, and $\mathcal{B}$ wins the GSD game.

We finally evaluate the advantage of $\mathcal{B}$. $\mathcal{B}$ can win the GSD game if $\mathcal{Z}$ distinguishes the hybrids and $\mathcal{B}$ succeeds in the guess. If $\mathcal{Z}$ distinguishes the two hybrids with non-negligible probability ϵ, $\mathcal{B}$ wins the game with probability ϵ/Q, which is also non-negligible. This contradicts the assumption that CmPKE is Chained CmPKE conforming GSD secure. Therefore, ϵ must be negligible, and we conclude that $\mathcal{Z}$'s view when $\mathsf{E}_{\mathsf{inj\text{-}w}}$ occurs is indistinguishable between Hybrid 4-1 and Hybrid 4-2. □

We finally prove that the probability the simulator aborts due to $\mathsf{abort}_{\mathsf{attach}}$ is negligible.

Proposition 4.4.30. *The probability* abort_{attach} *occurs is negligible if* CmPKE *is* Chained CmPKE *conforming GSD secure.*

Proof. The difference from the previous hybrid is $\mathcal{S}'_{4\text{-}2}$ aborts when event $\mathsf{abort}_{\mathsf{attach}}$ occurs. We show that, if $\mathcal{Z}$ can distinguish the two hybrids, then there exists an adversary $\mathcal{B}$ that can win the Chained CmPKE conforming GSD game. We first explain the description of $\mathcal{B}$ and how $\mathcal{B}$ embeds the GSD challenge. We then show the validity of the GSD challenge and finally evaluate $\mathcal{B}$'s advantage.

$\mathcal{B}$ simulates for $\mathcal{Z}$ the interaction with $\mathcal{F}^{\mathrm{IW}}_{\mathrm{AS}}$ and $\mathcal{F}^{\mathrm{IW}}_{\mathrm{KS}}$, and the protocol execution of each party as in $\mathcal{S}'_{4\text{-}2}$, except that $\mathcal{B}$ interacts with its GSD challenger instead of $\mathcal{S}_{\mathsf{GSD}}$. We assume $\mathcal{Z}$ generates at most Q commit message by invoking the commit protocol. At the beginning of the game, $\mathcal{B}$ chooses $i \in [Q]$ at random, and hopes that $\mathsf{abort}_{\mathsf{attach}}$ occurs while $\mathcal{B}$ generates the i-th commit message. ($\mathcal{B}$ succeeds to guess such a message with probability $1/Q$.) When $\mathcal{B}$ generates the i-th commit message c_0, $\mathcal{B}$ embeds the GSD challenge and determines the challenge bit as follows. We assume c_0 is generated by id without loss of generality. Let $u_{\mathsf{par\text{-}init}}$ be the GSD node assigned to the initial secret at epoch $\mathsf{Ptr}[\mathsf{id}]$ and let u_{com} be the GSD node assigned to the commit secret. Note that the confirmation hash $G'.\mathsf{confTransHash}$ of the new epoch is computed before deriving the epoch secret.

1. $\mathcal{B}$ prepares new GSD nodes $u_{\text{'joi'}}$ and $u_{\text{'conf'}}$ (their values are set to u_{ctr} and $u_{\mathsf{ctr}+1}$. ctr is incremented.) and queries $\texttt{Join-Hash}(u_{\mathsf{par\text{-}init}}, u_{\mathsf{com}}, u_{\text{'joi'}}, \text{'joi'})$ and $\texttt{Hash}(u_{\text{'joi'}}, u_{\text{'conf'}}, (G'.\mathsf{groupCont}(), \text{'conf'}))$.
2. $\mathcal{B}$ prepares a new GSD node $u_{\mathsf{ctag}} := u_{\mathsf{ctr}}$ and queries $\texttt{Hash}(u_{\text{'conf'}}, u_{\mathsf{ctag}}, G'.\mathsf{confTransHash})$.
3. $\mathcal{B}$ queries $\mathsf{confTag} := \texttt{Chall}(u_{\mathsf{ctag}})$ instead of $\texttt{Corr}(u_{\mathsf{ctag}})$.

4. If $(\bot, (\bot, \bot), *, G'.\mathsf{confTransHash}, \mathsf{confTag}') \in L_{\text{'epoch'}}$ exists for some $\mathsf{confTag}'$ such that $\mathsf{confTag} = \mathsf{confTag}'$, $\mathcal{B}$ submits 0 to the GSD challenger; otherwise submits 1.

Note that $\mathcal{B}$ can issue the above queries to oracles `Hash` and `Join-Hash` because the inputs are new GSD nodes. Note also that the confirmation key derived from the current initial secret and u_{com} is correctly computed to $u_{\text{'conf'}}$. $\mathcal{B}$ assigns $\mathsf{RO}(s_{\text{'conf'}}, G'.\mathsf{confTransHash})$ to u_{ctag}. If $\mathsf{abort}_{\mathsf{attach}}$ occurs, $\mathcal{B}$ has obtained $\mathsf{confTag}'$ (stored in $L_{\text{'epoch'}}$) that satisfies $\mathsf{confTag}' = \mathsf{RO}(s_{\text{'conf'}}, G'.\mathsf{confTransHash})$. Thus, if the challenge oracle returns the real value (i.e., the challenge bit is 0), $\mathsf{confTag} = \mathsf{confTag}'$ holds with probability 1; otherwise with negligible probability. Therefore, $\mathcal{B}$ can output the correct challenge bit with overwhelming probability.

We check the validity of the GSD challenge. Observe that GSD graph is acyclic and u_{ctag} is a sink node. In addition, by the structure of the GSD graph, we have

$$\begin{aligned}
\mathbf{gsd\text{-}exp}(u_{\mathsf{ctag}}) &= \big(u_{\mathsf{ctag}} \in \mathsf{Corr}\big) \vee \mathbf{gsd\text{-}exp}(u_{\text{'conf'}}) \\
&= \big(u_{\mathsf{ctag}} \in \mathsf{Corr}\big) \vee \big(u_{\text{'conf'}} \in \mathsf{Corr}\big) \vee \mathbf{gsd\text{-}exp}(u_{\text{'joi'}}) \\
&= \big(u_{\mathsf{ctag}} \in \mathsf{Corr}\big) \vee \big(u_{\text{'conf'}} \in \mathsf{Corr}\big) \vee \big(u_{\text{'joi'}} \in \mathsf{Corr}\big) \vee \\
&\quad \big(\mathbf{gsd\text{-}exp}(u_{\mathsf{par\text{-}init}}) \wedge \mathbf{gsd\text{-}exp}(u_{\mathsf{com}})\big) \\
&= \mathtt{false}.
\end{aligned}$$

This is because u_{ctag}, $u_{\text{'conf'}}$, $u_{\text{'joi'}}$ are not sent to `Corr` or `Set-Secret` oracle, and $\mathbf{gsd\text{-}exp}(u_{\mathsf{par\text{-}init}}) = \mathtt{false}$ when $\mathsf{abort}_{\mathsf{attach}}$ occurs. Hence, u_{ctag} is a valid challenge node, and $\mathcal{B}$ wins the GSD game.

We finally evaluate the advantage of $\mathcal{B}$. $\mathcal{B}$ can win the GSD game if $\mathcal{Z}$ distinguishes the hybrids and $\mathcal{B}$ succeeds in the guess. If $\mathcal{Z}$ distinguishes the two hybrids with non-negligible probability ϵ, $\mathcal{B}$ wins the game with probability ϵ/Q, which is also non-negligible. However, This contradicts the assumption that CmPKE is Chained CmPKE conforming GSD secure. Therefore, ϵ must be negligible, and we conclude that the probability $\mathsf{abort}_{\mathsf{attach}}$ occurs is negligible, i.e., the two hybrids are indistinguishable for $\mathcal{Z}$. □

From the above propositions, we conclude that Hybrid 4-1 and Hybrid 4-2 are indistinguishable for $\mathcal{Z}$. □

From Hybrid 4-3 to 4-4: Lemma 4.4.32. Before proving Lemma 4.4.32, we provide the key proposition that establishes the relationship between the safety predicate (**safe** and **know** shown in Figure 4.25) and the **gsd-exp** predicate. It will be used in the proof of Lemmata 4.4.32 and 4.4.34.

Proposition 4.4.31. *Let $u_{\text{'app'}}, u_{\text{'memb'}}, u_{\text{'conf'}}, u_{\text{'init'}}$ be the GSD node assigned to each epoch secret at epoch c_0. The following statements hold.*

- *If* $\boldsymbol{know}(c_0, \text{'epoch'}) = \mathtt{false}$, *then* $\boldsymbol{gsd\text{-}exp}(u) = \mathtt{false}$ *for* $u \in \{ u_{\text{'memb'}}, u_{\text{'conf'}}, u_{\text{'init'}} \}$.
- *If* $\boldsymbol{safe}(c_0) = \mathtt{true}$, *then* $\boldsymbol{gsd\text{-}exp}(u_{\text{'app'}}) = \mathtt{false}$.

Proof. We show the contraposition of the statements. That is, we prove

- If $\mathbf{gsd\text{-}exp}(u) = \mathtt{true}$ for $u \in \{ u_{\text{'memb'}}, u_{\text{'conf'}}, u_{\text{'init'}} \}$, then $\mathbf{know}(c_0, \text{'epoch'}) = \mathtt{true}$, and
- If $\mathbf{gsd\text{-}exp}(u_{\text{'app'}}) = \mathtt{true}$, then $\mathbf{safe}(c_0) = \mathtt{false}$.

Recalling the definition of **gsd-exp** and the GSD graph created by $\mathcal{S}'_{4\text{-}2}$, for each $\hat{u} \in \{ u_{\text{'memb'}}, u_{\text{'conf'}}, u_{\text{'init'}}, u_{\text{'app'}} \}$, we have

$$
\begin{aligned}
\textbf{gsd-exp}(\hat{u}) &\iff \big(\hat{u} \in \mathsf{Corr}\big) \vee \textbf{gsd-exp}(u_{\text{'joi'}}) \\
&\iff \big(\hat{u} \in \mathsf{Corr}\big) \vee \big(u_{\text{'joi'}} \in \mathsf{Corr}\big) \vee \textbf{gsd-exp}(u_{\mathsf{id},\text{'joi'}}) \vee \big(\textbf{gsd-exp}(u_{\mathsf{com}}) \wedge \textbf{gsd-exp}(u_{\mathsf{par\text{-}init}})\big) \\
&\iff \big(\hat{u} \in \mathsf{Corr}\big) \vee \big(u_{\text{'joi'}} \in \mathsf{Corr}\big) \vee \big(u_{\mathsf{id},\text{'joi'}} \in \mathsf{Corr}\big) \\
&\quad \vee \big(\big((u_{\mathsf{com}} \in \mathsf{Corr}) \vee \textbf{gsd-exp}(u_{\mathsf{id},\text{'com'}})\big) \wedge \textbf{gsd-exp}(u_{\mathsf{par\text{-}init}})\big) \\
&\iff \hat{u} \in \mathsf{Corr} \cdots \text{ Case (A)} \\
&\quad \vee \big(u_{\text{'joi'}} \in \mathsf{Corr}\big) \vee \big(u_{\mathsf{id},\text{'joi'}} \in \mathsf{Corr}\big) \cdots \text{ Case (B)} \\
&\quad \vee \big((u_{\mathsf{com}} \in \mathsf{Corr}) \vee (u_{\mathsf{id},\text{'com'}} \in \mathsf{Corr})\big) \wedge \textbf{gsd-exp}(u_{\mathsf{par\text{-}init}}), \cdots \text{ Case (C)}
\end{aligned}
$$

where $u_{\text{'joi'}}$ (resp. u_{com}) is the GSD node assigned to the joiner (resp. commit) secret at epoch c_0, $u_{\mathsf{id},\text{'joi'}}$ (resp. $u_{\mathsf{id},\text{'com'}}$) is the GSD node assigned to an encryption key used to encrypt $u_{\text{'joi'}}$ (resp. u_{com}) at c_0, and $u_{\mathsf{par\text{-}init}}$ is the GSD node assigned to the initial secret at the parent node of c_0. Note that Case (C) is evaluated when c_0 is a non-root node because otherwise, the GSD graph starts from $u_{\text{'joi'}}$. In the following, we analyze Case (A), Case (B), and Case (C) in order.

[Case (A): $\big(\hat{u} \in \mathsf{Corr}\big) = \mathsf{true}$**].** $\hat{u} \in \mathsf{Corr}$ becomes true when $\mathcal{S}'_{4\text{-}2}$ queries $\mathtt{Corr}(\hat{u})$. (Note that $\mathcal{S}'_{4\text{-}2}$ never queries $\mathtt{Set\text{-}Secret}(\hat{u}, *)$.) By the description of $\mathcal{S}'_{4\text{-}2}$, it queries $\mathtt{Corr}(\hat{u})$ if (1) a party at c_0 is corrupted via Corruption query; or (2) the committer of c_0 is corrupted via Corruption query while it is at the parent node of c_0 (this corresponds to the fact that the committer stores pending commits in pendCom array).

Case (A-1): It immediately implies $\textbf{know}(c_0, \text{'epoch'}) = \mathtt{true}$ because $\mathsf{Node}[c_0].\mathsf{exp} \neq \emptyset$ becomes true when a party at c_0 is corrupted. In particular, if the party is corrupted before Key query is issued to the epoch c_0, $(*, \mathtt{true}) \in \mathsf{Node}[c_0].\mathsf{exp}$ becomes true. This implies $\textbf{safe}(c_0) = \mathtt{false}$.

Case (A-2): When the committer is corrupted while it is at the parent node of c_0, the status of the node c_0 is set to 'bad' (cf. `*update-stat-after-exp`). This implies $\textbf{know}(c_0, \text{'epoch'}) = \mathtt{true}$ and $\textbf{safe}(c_0) = \mathtt{false}$ due to Condition (a) of ***secrets-injected**.

[Case (B): $\big(u_{\text{'joi'}} \in \mathsf{Corr}\big) \vee \big(u_{\mathsf{id},\text{'joi'}} \in \mathsf{Corr}\big) = \mathsf{true}$**].** First, $u_{\text{'joi'}} \in \mathsf{Corr}$ becomes true when $\mathcal{S}'_{4\text{-}2}$ queries $\mathtt{Corr}(u_{\text{'joi'}})$ or $\mathtt{Set\text{-}Secret}(u_{\text{'joi'}}, *)$.

Recalling the description of $\mathcal{S}'_{4\text{-}2}$, it issues $\mathtt{Corr}(u_{\text{'joi'}})$ if (1) in the commit protocol, a new member is added with a malicious key package generated by $\mathcal{A}$; or (2) in the commit protocol, the committer encrypts the joiner secret at c_0 using bad randomness.

Case (B-1): When the adversary succeeds to inject a malicious key package, the signing key used to generate the key package must be exposed due to the modification we made in Hybrid 4-1. Thus, due to Condition (a) of ***can-traverse**, $\textbf{*can-traverse}(c_0) = \mathtt{true}$.

Case (B-2): If the committer encrypts the joiner secret, $\mathsf{Node}[c_0].\mathsf{prop}$ contains at least one add proposal. In addition, the status of c_0 is set to 'bad' because the committer uses bad randomness. Thus, due to Condition (c) of ***can-traverse**, $\textbf{*can-traverse}(c_0) = \mathtt{true}$.

It issues $\mathtt{Set\text{-}Secret}(u_{\text{'joi'}}, *)$ if (3) in the create protocol, the group creator initializes the group using bad randomness; or (4) in the join protocol, a party joins a group that $\mathcal{S}'_{4\text{-}2}$ has not created (i.e., the party assigns to a detached root).

Case (B-3): When the group creator initialize c_0 (= root_0) using bad randomness, the status of c_0 is set to 'bad'. Thus, due to Condition (a) of ***secrets-injected**, **know**$(c_0, \mathsf{id}_{\mathsf{creator}}) = \texttt{true}$. Since c_0 is a root node, due to Condition (d) of ***can-traverse**, we have ***can-traverse**$(c_0) = \texttt{true}$.

Case (B-4): When a party joins a group that the adversary creates, the party is assigned to a detached root, and its status is 'adv'. Thus, due to Condition (a) of ***secrets-injected**, **know**$(c_0, \mathsf{id}_c) = \texttt{true}$, where id_c is the sender of the welcome message. Moreover, c_0 is a root node. Thus, due to Condition (d) of ***can-traverse**, ***can-traverse**$(c_0) = \texttt{true}$.

Next, $u_{\mathsf{id},\text{'joi'}} \in \mathsf{Corr}$ becomes true when $\mathcal{S}'_{4\text{-}2}$ queries $\texttt{Corr}(u_{\mathsf{id},\text{'joi'}})$ or $\texttt{Set-Full-Secret}(u_{\mathsf{id},\text{'joi'}}, *)$. Recalling the description of $\mathcal{S}'_{4\text{-}2}$, it issues $\texttt{Corr}(u_{\mathsf{id},\text{'joi'}})$ if (5) the adversary corrupts the new member id_t before it joins a group; or (6) the encryption key of a new member id_t used to encrypt the joiner secret at c_0 is corrupted after epoch c_0.

Case (B-5): When the adversary corrupts a member id_t before id_t joins a group, the signing key used to generate the key package is marked as exposed (cf. $(\texttt{exposed}, \mathsf{id}_t)$ query to $\mathcal{F}^{\mathrm{IW}}_{\mathrm{KS}}$). Thus, due to Condition (a) of ***can-traverse**, ***can-traverse**$(c_0) = \texttt{true}$.

Case (B-6): This case happens if a party id_t holds the same encryption key at both c_0 and c'_0, where c'_0 is a descendant node of c_0, and id_t is corrupted at c'_0. Since id_t is corrupted at c'_0, $(\mathsf{id}_t, *) \in \mathsf{Node}[c'_0].\mathsf{exp}$ becomes true, and we have **know**$(c'_0, \mathsf{id}_t) = \texttt{true}$. In addition, since id_t holds the same encryption key at both c_0 and c'_0, id_t did not perform any actions that replace id_t's encryption key between c_0 and c'_0. Thus, $\neg$***secrets-replaced**$(c''_0, \mathsf{id}_t) = \texttt{true}$ for each c''_0 on c_0–c'_0. Therefore, due to Condition (b) of ***can-traverse**, ***can-traverse**$(c_0) = \texttt{true}$ (cf. ***reused-welcome-key-leaks**).

It issues $\texttt{Set-Full-Secret}(u_{\mathsf{id},\text{'joi'}}, *)$ if (7) the key package in the add proposal is generated using bad randomness.

Case (B-7): When the key package for add proposals is generated using bad randomness, $\mathcal{F}^{\mathrm{IW}}_{\mathrm{KS}}$ marks that the signing key is exposed (cf. `register-kp` query with bad randomness). Thus, due to Condition (a) of ***can-traverse**, ***can-traverse**$(c_0) = \texttt{true}$.

In all cases, if Case (B) is true, then ***can-traverse**$(c_0) = \texttt{true}$; It implies **know**$(c_0, \text{'epoch'}) = \texttt{true}$ and **safe**$(c_0) = \texttt{false}$.

[Case (C): $\big((u_{\mathbf{com}} \in \mathsf{Corr}) \vee (u_{\mathsf{id},\text{'}\mathbf{com}\text{'}} \in \mathsf{Corr})\big) \wedge$ **gsd-exp**$(u_{\mathbf{par\text{-}init}}) = \mathsf{true}$**].** We below show that $u_{\mathsf{com}} \in \mathsf{Corr} \vee u_{\mathsf{id},\text{'com'}} \in \mathsf{Corr}$ (i.e., **gsd-exp**$(u_{\mathsf{com}}) = \texttt{true}$) implies **know**$(c_0, \mathsf{id}) = \texttt{true}$ for some $\mathsf{id} \in \mathsf{Node}[c_0].\mathsf{mem}$. By applying Proposition 4.4.31 to the parent node of c_0 (i.e., $\mathsf{Node}[c_0].\mathsf{par}$), we can show that **gsd-exp**$(u_{\mathsf{par\text{-}init}}) =$ `true` implies **know**$(\mathsf{Node}[c_0].\mathsf{par}, \text{'epoch'}) = \texttt{true}$. As a result, due to Condition (d) of ***can-traverse**, if Case (C) is true, then ***can-traverse**$(c_0) = \texttt{true}$, which implies **know**$(c_0, \text{'epoch'}) = \texttt{true}$ and **safe**$(c_0) =$ `false`.

First, $u_{\mathsf{com}} \in \mathsf{Corr}$ becomes true when $\mathcal{S}'_{4\text{-}2}$ queries $\texttt{Set-Secret}(u_{\mathsf{com}}, *)$.[28] Recalling the description of $\mathcal{S}'_{4\text{-}2}$, it issues $\texttt{Set-Secret}(u_{\mathsf{com}}, *)$ if (1) in the commit protocol, a malicious encryption key generated by $\mathcal{A}$ is used to encrypt the commit secret at c_0; (2) in the commit protocol, the committer generates c_0 using bad randomness; or (3) in the process protocol, a party processes an injected commit message.

Case (C-1): This case occurs if the committer of c_0 has applied (I) an injected update proposal that contains a malicious encryption key or (II) an add proposal that contains a malicious key package generated by $\mathcal{A}$ at a previous epoch, and the committer uses the same encryption key to generate c_0.

[28] By definition, $\mathcal{S}'_{4\text{-}2}$ never queries $\texttt{Corr}(u_{\mathsf{com}})$.

Case (C-1-I): Assume the committer has applied the injected update proposal sent from id at the node c_0', which is an ancestor of c_0 (including the case $c_0' = c_0$). When the adversary succeeds to inject an update proposal that contains the encryption key generated by the adversary, the corresponding proposal node has the states 'adv'. Hence, due to Condition (b) of ***secrets-injected**, we have $\mathbf{know}(c_0', \mathsf{id}) = \mathtt{true}$. In addition, if id uses the same encryption key at both c_0' and c_0, id did not perform any actions that replace id's encryption key between c_0' and c_0. Thus, $\neg$***secrets-replaced**(c_0'', id) is true for each c_0'' on c_0'–c_0 path. Therefore, Condition (c) of $\mathbf{know}(c_0, \mathsf{id})$ returns true.

Case (C-1-II): Assume the committer has applied the add proposal from id at the node c_0', which is an ancestor of c_0. When the adversary succeeds to inject a key package that contains a malicious encryption key, the signing key used to generate the key package must be exposed due to the modification we made in Hybrid 4-1 (or Hybrid 5-1). Hence, due to Condition (c) of ***secrets-injected**, we have $\mathbf{know}(c_0', \mathsf{id}) = \mathtt{true}$. In addition, since id uses the same encryption key at both c_0' and c_0, id did not perform any actions that replace id's encryption key between c_0' and c_0. Thus, $\neg$***secrets-replaced**(c_0'', id) is true for each c_0'' on c_0'–c_0 path. Therefore, Condition (c) of $\mathbf{know}(c_0, \mathsf{id})$ returns $\mathtt{true}$.

Case (C-2): If the committer generates c_0 using bad randomness, the status of c_0 is set to 'bad'. Thus, due to Condition (a) of ***secrets-injected**, $\mathbf{know}(c_0, \mathsf{id}_c) = \mathtt{true}$.

Case (C-3): Due to the modification we made in Hybrid 4-2 (cf. $\mathsf{E}_{\text{inj-c-2}}$), if the node c_0 is created by the process protocol, both $\mathbf{gsd\text{-}exp}(u_{\text{par-init}}) = \mathtt{true}$ and $\mathbf{gsd\text{-}exp}(u_{\mathsf{com}}) = \mathtt{true}$ hold. Moreover, the status of c_0 is set to 'adv'. Thus, due to Condition (a) of ***secrets-injected** and Condition (d) of ***can-traverse**, $\mathbf{know}(c_0, \text{'epoch'}) = \mathtt{true}$ and $\mathbf{safe}(c_0) = \mathtt{false}$ always hols in this case.

Next, $u_{\mathsf{id},\text{'com'}} \in \mathsf{Corr}$ becomes true if $\mathcal{S}'_{4\text{-}2}$ queries $\mathtt{Corr}(u_{\mathsf{id},\text{'com'}})$ or $\mathtt{Set\text{-}Full\text{-}Secret}(u_{\mathsf{id},\text{'com'}}, *)$. $\mathcal{S}'_{4\text{-}2}$ issues $\mathtt{Corr}(u_{\mathsf{id},\text{'com'}})$ if (4) a party id is corrupted via Corruption query while it holes the encryption key $u_{\mathsf{id},\text{'com'}}$; or (5) id issues an update proposal at the parent node of c_0 and id is corrupted before processing the commit message c_0 (this corresponds to the fact that id stores the pending update proposals including encryption keys in $\mathsf{pendUpd}$ array).

Case (C-4): Assume id is corrupted at epoch c_0'. That is, $(\mathsf{id}, *) \in \mathsf{Node}[c_0'].\mathsf{exp}$ is true. In addition, since party id uses the same encryption key $u_{\mathsf{id},\text{'com'}}$ at c_0, id does not perform any actions that replace id's encryption key between c_0 and c_0'. Thus, $\neg$***secrets-replaced**(c_0'', id) is true for each c_0'' on c_0'–c_0 or c_0–c_0' path. Due to Condition (c) or (d) of $\mathbf{know}(c_0, \mathsf{id})$, we have $\mathbf{know}(c_0, \mathsf{id}) = \mathtt{true}$.

Case (C-5): If id is corrupted at the parent node of c_0, the status of the corresponding proposal nodes stored in $\mathsf{pendUpd}$ array is set to 'bad' (cf. $\mathtt{*update\text{-}stat\text{-}after\text{-}exp}$). Thus, due to Condition (b) of ***secrets-injected**, we have $\mathbf{know}(c_0, \mathsf{id}) = \mathtt{true}$.

$\mathtt{Set\text{-}Full\text{-}Secret}(u_{\mathsf{id},\text{'com'}}, *)$ was queried to $u_{\mathsf{id},\text{'com'}}$ if (6) the committer of c_0 applied an update proposal from id generated with bad randomness and uses the same encryption key at c_0 or (7) the committer applied an add proposal from id generated with bad randomness and uses the same encryption key at c_0.

Case (C-6): Assume id issues an update proposal using bad randomness and the committer applies it at epoch c_0'. At the epoch, $\mathcal{S}'_{4\text{-}2}$ issues $\mathtt{Set\text{-}Full\text{-}Secret}(u_{\mathsf{id},\text{'com'}}, *)$ during key package generation, and due to Condition (b) of ***secrets-injected**, $\mathbf{know}(c_0', \mathsf{id})$ is true. In addition, since party id uses the encryption key $u_{\mathsf{id},\text{'com'}}$ at c_0, id does not perform any actions that replace id's encryption key between

c_0 and c_0'. Thus, $\neg$***secrets-replaced**(c_0'', id) is true for each c_0'' on c_0'–c_0 path. Due to Condition (d) of **know**(c_0, id), we have **know**$(c_0, \mathsf{id}) = \texttt{true}$.

Case (C-7): Assume id added a group at epoch c_0' using an add proposal generated with bad randomness. When the key package was generated, $\mathcal{S}'_{4\text{-}2}$ issues $\texttt{Set-Full-Secret}(u_{\mathsf{id},\text{'com'}}, *)$, and due to Condition (c) of ***secrets-injected**, **know**(c_0', id) is true. In addition, since party id uses the encryption key $u_{\mathsf{id},\text{'com'}}$ at c_0, id does not perform any actions that replace id's encryption key between c_0 and c_0'. Thus, $\neg$***secrets-replaced**(c_0'', id) is true for each c_0'' on c_0'–c_0 path. Due to Condition (d) of **know**(c_0, id), we have **know**$(c_0, \mathsf{id}) = \texttt{true}$.

In all cases, **know**$(c_0, \mathsf{id}) = \texttt{true}$ for some $\mathsf{id} \in \mathsf{Node}[c_0].\mathsf{mem}$. Therefore, if **gsd-exp**$(u_{\mathsf{par\text{-}init}}) = \texttt{true}$ holds additionally, Case (C) implies **know**$(c_0, \text{'epoch'}) = \texttt{true}$ and **safe**$(c_0) = \texttt{false}$.

From the above discussion, we obtain the following statements.

- If **know**$(c_0, \text{'epoch'}) = \texttt{false}$, then **gsd-exp**$(u) = \texttt{false}$ for $u \in \{ u_{\text{'memb'}}, u_{\text{'conf'}}, u_{\text{'init'}} \}$.
- If **safe**$(c_0) = \texttt{true}$, then **gsd-exp**$(u_{\text{'app'}}) = \texttt{false}$.

□

Now we ready to prove Lemma 4.4.32.

Lemma 4.4.32. *Hybrid 4-3 and Hybrid 4-4 are identical.*

Proof. The difference between Hybrid 4-3 and Hybrid 4-4 is that in Hybrid 4-4 we use the original `auth-invariant` predicate and the functionally $\mathcal{F}_{\text{CGKA},4}$ halts if `auth-invariant` returns false. We show that the simulator $\mathcal{S}_{4\text{-}4}$ never creates history graph nodes such that `auth-invariant` returns false, that is, $\mathcal{F}_{\text{CGKA},4}$ never halts. We consider Condition (a) and Condition (b) of `auth-invariant` in order.

Condition (a) of `auth-invariant`. We first show that, for all non-root node c_0 in the history graph created by $\mathcal{S}_{4\text{-}4}$, if $\mathsf{Node}[c_0].\mathsf{stat} = \text{'adv'}$, then **mac-inj-allowed**$(c_p) = \texttt{true}$, where $c_p := \mathsf{Node}[c_0].\mathsf{par}$ (non-root implies $c_p \neq \bot$). By the definition of the functionality, a non-root commit node c_0 with status 'adv' is created when (1) an existing detached root is attached to a commit node generated by the commit protocol using bad randomness; or (2) the commit node is created by the process protocol. On the other hand, (1) $\mathcal{S}_{4\text{-}4}$ aborts when an existing detached root will be attached to a commit node generated using bad randomness if **gsd-exp**$(u_{\mathsf{par\text{-}init}}) = \texttt{false}$ (cf. $\mathsf{abort}_{\mathsf{attach}}$); and (2) $\mathcal{S}_{4\text{-}4}$ always rejects the injected commit message c_0 in the process protocol if **gsd-exp**$(u_{\mathsf{par\text{-}init}}) = \texttt{false}$ (cf. $\mathsf{E}_{\mathsf{inj\text{-}c\text{-}1}}$). Here, $u_{\mathsf{par\text{-}init}}$ is the GSD node assigned to the initial secret at the parent epoch of c_0. Thus, commit nodes with status 'adv' are created only if **gsd-exp**$(u_{\mathsf{par\text{-}init}}) = \texttt{true}$. Moreover, due to Proposition 4.4.31, if **gsd-exp**$(u_{\mathsf{par\text{-}init}}) = \texttt{true}$, then **know**$(c_p, \text{'epoch'}) = \texttt{true}$, i.e., **mac-inj-allowed**$(c_p) = \texttt{true}$. Therefore, commit nodes with status 'adv' are created only if **mac-inj-allowed**$(c_p) = \texttt{true}$. In other words, there exists no node c_0 such that $\mathsf{Node}[c_0].\mathsf{stat} = \text{'adv'}$ and **mac-inj-allowed**$(c_p) = \texttt{false}$ in the history graph created by $\mathcal{S}_{4\text{-}4}$.

Condition (b) of `auth-invariant`. We next show that, for all proposal node p in the history graph, if $\mathsf{Prop}[p].\mathsf{stat} = \text{'adv'}$, then **mac-inj-allowed**$(c_p) = \texttt{true}$, where $c_p := \mathsf{Prop}[p].\mathsf{par}$ (by definition, c_p always non-$\bot$). By the definition of the functionality, a proposal node with status 'adv' is created if a proposal message is not generated by the propose protocol. On the other hand, if a received proposal is not generated by the propose protocol and **gsd-exp**$(u_{\text{'memb'}}) = \texttt{false}$, the commit and process protocol always outputs $\bot$ (i.e., rejects the message), where $u_{\text{'memb'}}$ is the GSD node assigned to the corresponding membership key at c_p. In other words, proposal nodes with status 'adv' is created

only if **gsd-exp**$(u_{\text{'memb'}}) = \texttt{true}$. Moreover, due to Proposition 4.4.31, if **gsd-exp**$(u_{\text{'memb'}}) = \texttt{true}$, then **know**$(c_p, \text{'epoch'}) = \texttt{true}$, i.e., **mac-inj-allowed**$(c_p) = \texttt{true}$. Therefore, proposal nodes with status 'adv' are created only if **mac-inj-allowed**$(c_p) = \texttt{true}$. In other words, there exists no proposal node p such that $\mathsf{Prop}[p].\mathsf{stat} = \text{'adv'}$ and **mac-inj-allowed**$(c_p) = \texttt{true}$ in the history graph created by $\mathcal{S}_{4\text{-}4}$.

From the above discussion, $\mathcal{S}_{4\text{-}4}$ never creates history graph nodes such that `auth-invariant` returns false, i.e., the functionality never halts. Thus, Hybrid 4-3 and Hybrid 4-4 are identical. □

From Hybrid 5 to 6: Lemma 4.4.33. In Hybrid 6, receiving `Key` query, the functionality $\mathcal{F}_{\text{CGKA}}$ returns a random application secret if **safe** is true for the queried epoch. To prove the indistinguishability of the two hybrids, we gradually replace each application secret with a random value instead of the real value if **safe** is true. We show that, if $\mathcal{Z}$ can distinguish whether the application secret is real or random, it can be used to break the Chained CmPKE conforming GSD security of CmPKE. In other words, if CmPKE is Chained CmPKE conforming GSD secure, Hybrid 5 and Hybrid 6 are indistinguishable. We below provide formal proof of the above overview.

Lemma 4.4.33. *Hybrid 5 and Hybrid 6 are indistinguishable assuming* CmPKE *is* Chained CmPKE *conforming GSD secure.*

Proof. We assume $\mathcal{Z}$ creates at most Q epochs (i.e., commit nodes). To show Lemma 4.4.33, we consider the following sub-hybrids between Hybrid 5 and Hybrid 6.

Hybrid 5-0. This is identical to Hybrid 5. We use the functionality $\mathcal{F}_{\text{CGKA,5}}$, and all application secrets are set to the real value by the simulator $\mathcal{S}_{5\text{-}0} := \mathcal{S}_5$.

Hybrid 5-*i*. i runs through $[Q]$. The simulator $\mathcal{S}_{5\text{-}i}$ is defined exactly as $\mathcal{S}_{5\text{-}(i-1)}$ except that it sets the i-th application secret to random if **safe** is true. Note that we count application secrets in the order in which `Key` query is issued. We show in Lemma 4.4.34 that Hybrid 5-$(i-1)$ and Hybrid 5-i are indistinguishable.

Hybrid 6. We replace the functionality $\mathcal{F}_{\text{CGKA,5}}$ with the original functionality $\mathcal{F}_{\text{CGKA}}$. In this hybrid, all application secrets such that **safe** is true are set to random by $\mathcal{F}_{\text{CGKA}}$. From $\mathcal{Z}$'s point of view, Hybrid 5-Q and Hybrid 6 are identical because the only difference is who sets application secrets to random.

The indistinguishability between Hybrid 5-0 and Hybrid 5-Q is derived by applying Lemma 4.4.34 for all $i \in [Q]$. Therefore, we conclude that Hybrid 5 and Hybrid 6 are indistinguishable. □

Lemma 4.4.34. *Hybrid 5-$(i-1)$ and Hybrid 5-i are indistinguishable assuming* CmPKE *is* Chained CmPKE *conforming GSD secure.*

Proof. The difference between Hybrid 5-$(i-1)$ and Hybrid 5-i is whether the i-th application secret is real or random if **safe** is true. We show if $\mathcal{Z}$ can distinguish the two hybrids, there exists an adversary $\mathcal{B}$ against the Chained CmPKE conforming GSD game. We first explain the description of $\mathcal{B}$. We then show the validity of the GSD challenge and finally evaluate $\mathcal{B}$'s advantage.

$\mathcal{B}$ simulates for $\mathcal{Z}$ the interaction with $\mathcal{F}_{\text{AS}}^{\text{IW}}$ and $\mathcal{F}_{\text{KS}}^{\text{IW}}$, and the protocol execution of each party as in $\mathcal{S}'_{4\text{-}2}$, except that $\mathcal{B}$ interacts with its GSD challenger instead of $\mathcal{S}_{\text{GSD}}$. $\mathcal{B}$ embeds the GSD challenge in the i-th application secret as follows: Assume $\mathcal{Z}$ issues `Key` query to id and $c_0 = \mathsf{Ptr}[\mathsf{id}]$ is the j-th to be queried `Key`.

- If **safe**$(c_0) = \texttt{true}$ and $j < i$, $\mathcal{B}$ returns the random value.

TABLE 4.6: Bandwidth costs of mPKEs derived from existing parametrizations (gray background) and new ones (white background), for $\kappa = 128$ bits of classical security. Standard (single-recipient) PKE instantiations of existing schemes may include a seed in the encryption key or a confirmation hash in the ciphertext (in parentheses).

Scheme	Reference	$\lvert\mathsf{ek}\rvert$	$\lvert\mathsf{ct}_0\rvert$	$\lvert\widehat{\mathsf{ct}}_i\rvert$
Kyber512	[Sch+20]	768 (+32)	640	128
Ilum512	Section 4.5	768	704	48
LPRime653	[Ber+20]	865 (+32)	865 (+32)	128
LPRime757	Section 4.5	1076	1076	32
Frodo640	[Nae+20]	9600 (+16)	9600	120
Bilbo640	Section 4.5	10240	10240	24
SIKEp434	[Jao+20]	330	330	16

- Else if $\mathbf{safe}(c_0) = \texttt{true}$ and $j = i$, $\mathcal{B}$ queries $s_{\mathsf{app}'} := \texttt{Chall}(u_{\mathsf{app}'})$ and returns the challenge value $s_{\mathsf{app}'}$, where $u_{\mathsf{app}'}$ is the GSD node corresponding to the application secret of c_0.
- Else, $\mathcal{B}$ returns the real application secret (if necessary, $\mathcal{B}$ corrupts the corresponding GSD node $u_{\mathsf{app}'}$).

Observe that, if $\mathbf{safe}(c_0) = \texttt{false}$ for the i-th application secret, the challenge is not embedded. In this case, the two hybrids proceed exactly the same.

We consider the case $\mathcal{B}$ embedded the GSD challenge. We argue the GSD challenge is valid. Observe that the GSD graph is acyclic and the challenge node $u_{\mathsf{app}'}$ is a sink node. In addition, due to Proposition 4.4.31, $\mathbf{safe}(c_0) = \texttt{true}$ implies $\mathbf{gsd\text{-}exp}(u_{\mathsf{app}'}) = \texttt{false}$. Thus, $u_{\mathsf{app}'}$ is a valid challenge node.

We finally analyze $\mathcal{B}$'s advantage. If the challenge oracle returns the real value, $\mathcal{Z}$'s view is identical to Hybrid 5-$(i-1)$; else, i.e., the challenge oracle returns a random value, $\mathcal{Z}$'s view is identical to Hybrid 5-i. Hence, if $\mathcal{Z}$ distinguishes Hybrid 5-$(i-1)$ and Hybrid 5-i with non-negligible probability, $\mathcal{B}$ wins the GSD game with non-negligible probability by using $\mathcal{Z}$'s output. This contradicts the assumption that CmPKE is Chained CmPKE conforming GSD secure. Therefore, Hybrid 5-$(i-1)$ and Hybrid 5-i are indistinguishable. □

4.5 More Efficient Lattice-Based mPKEs

To maximize the bandwidth savings of Chained CmPKE we must reduce $|\widehat{\mathsf{ct}}_i|$ as much as possible. Indeed, see Table 4.1, where the "Ours" row is only less performant than another in one column, namely Upload $|\widehat{\mathsf{ct}}_i|$. Therefore, in this section we outline the methods employed to achieve this. We adapt several PKEs from the literature to mPKEs, specifically PKEs which underly KEMs that are either finalists or alternative finalists of the final round of the NIST PQC process [Ala+20]. Throughout this section we only consider IND-CPA mPKEs, and use the notation of Definition 2.9.1. For the needs of the protocol in Section 4.4, these can be converted into IND-CCA CmPKEs with a small overhead using Theorem 4.2.7.

We start from the construction of [Kat+20], reproduced in Figure 4.26, which adapts the Lindner–Peikert framework [LP11] to the mPKE setting. As observed by [Kat+20], Figure 4.26 can be readily applied

$\mathsf{mSetup}(1^\kappa)$

1 : $\mathbf{A} \leftarrow\$ R_q^{n\times n}$
2 : **return** $\mathsf{pp} := \mathbf{A}$

$\mathsf{mGen}(\mathsf{pp})$

1 : $\mathbf{S} \leftarrow\$ D_s^{n\times \bar{n}}$
2 : $\mathbf{E} \leftarrow\$ D_e^{n\times \bar{n}}$
3 : $\mathbf{B} \leftarrow \mathbf{AS} + \mathbf{E}$
4 : **return** $\mathsf{ek} := \mathbf{B}, \mathsf{dk} := \mathbf{S}$

$\mathsf{mEnc}(\mathsf{pp}, (\mathsf{ek}_i)_{i\in[N]}, \mathsf{m})$

1 : $\mathsf{r}_0 := (\mathbf{R}, \mathbf{E}') \leftarrow\$ D_s^{\bar{m}\times n} \times D_{e'}^{\bar{m}\times n}$
2 : $\mathsf{ct}_0 := \mathsf{mEnc}^{\mathsf{i}}(\mathsf{pp}; \mathsf{r}_0)$
3 : **foreach** $i \in [N]$ **do**
4 : $\quad \mathsf{r}_i := \mathbf{E}_i'' \leftarrow\$ D_{e''}^{\bar{m}\times \bar{n}}$
5 : $\quad \widehat{\mathsf{ct}}_i := \mathsf{mEnc}^{\mathsf{d}}(\mathsf{pp}, \mathsf{ek}_i, \mathsf{m}; \mathsf{r}_0, \mathsf{r}_i)$
6 : **return** $\vec{\mathsf{ct}} := (\mathsf{ct}_0, \widehat{\mathsf{ct}}_1, \ldots, \widehat{\mathsf{ct}}_N)$

$\mathsf{mDec}(\mathsf{sk}, \mathsf{ct})$

1 : **return** $\mathsf{m} := \mathsf{Decode}(\mathbf{V} - \mathbf{US})$

$\mathsf{mEnc}^{\mathsf{i}}(\mathsf{pp}; \mathsf{r}_0)$

1 : $\mathbf{U} \leftarrow \mathbf{RA} + \mathbf{E}'$
2 : **return** $\mathsf{ct}_0 := \mathbf{U}$

$\mathsf{mEnc}^{\mathsf{d}}(\mathsf{pp}, \mathsf{ek}_i, \mathsf{m}; \mathsf{r}_0, \mathsf{r}_i)$

1 : $\mathbf{V}_i \leftarrow \mathbf{RB}_i + \mathbf{E}_i'' + \mathsf{Encode}(\mathsf{m})$
2 : **return** $\widehat{\mathsf{ct}}_i := \mathbf{V}_i$

FIGURE 4.26: Lattice-based mPKE construction of [Kat+20]. R is the base ring, $D_s, D_e, D_{e'}, D_{e''}$ are distributions over R.

to the (possibly alternative) finalists FrodoKEM [Nae+20], Kyber [Sch+20], NTRU LPRime [Ber+20] and Saber [DAn+20]. We take this one step further and propose new parametrizations of [Nae+20; Sch+20; Ber+20] that are tailored to the mPKE setting. At the cost of less than a 20% increase in $|\mathsf{ek}| + |\mathsf{ct}_0|$, we reduce $|\widehat{\mathsf{ct}}_i|$ by 60–80%. Since the size of an uploaded package is asymptotically $\sim |\widehat{\mathsf{ct}}_i| \cdot N$, we view this trade-off as favorable.

This section is arranged as follows. In Section 4.5.1, we review the techniques that one can leverage to minimize $|\widehat{\mathsf{ct}}_i|$. Then in Section 4.5.2 we provide new parametrizations of [Nae+20; Sch+20; Ber+20]. Finally, Section 4.5.3 details our cryptanalytic model, and provides security estimates for our parameter sets in this model.

4.5.1 Our Toolkit for Improving Efficiency

We review the known techniques at our disposal to minimize the size of the $(\widehat{\mathsf{ct}}_i)_i$ while increasing as little as possible the sizes of ek and ct_0, and maintaining security against known attacks. The coefficient dropping and modulus rounding techniques are already present in [Ber+20] and [Sch+20] respectively. Concretely, for modulus rounding we will focus on the Compress and Decompress functions of [Sch+20]. By *more* or *less* rounding, we mean a *smaller* or *larger* d in the definition of those functions, respectively. We note that modulus rounding techniques can be applied to the original parametrizations of [Nae+20], but save little in the $|\mathsf{ek}| + |\mathsf{ct}_0| + |\widehat{\mathsf{ct}}_i|$ (i.e., single recipient) metric. We revisit these techniques in light of the new constraints imposed by the mPKE setting, which in turn leads to new parameter sets. Throughout we reference Figure 4.26.

We note that the ciphertexts of some PKEs and mPKEs based on lattices have a small probability of decrypting to a different message than was initially encrypted. The probability of this occuring is called the decryption failure rate, or DFR. Keeping the DFR low, specifically $O(2^{-\kappa})$, is important for both correctness and security; we discuss it more in Section 4.5.3.

Coefficient Dropping. When trying to decode a message m from $(\mathbf{U}, \mathbf{V}_i)$ using $\mathbf{S}$, not all of $\mathbf{V}_i$ may be necessary. Indeed let $R = \mathbb{Z}[x]/(f)$, $d = \deg(f)$, $I < d$, and $\bar{n} = \bar{m} = 1$. If $\mathsf{Encode}(\mathsf{m}) = \alpha_{I-1}x^{I-1} + \cdots + \alpha_0$ then only the I lower order coefficients of $\mathbf{V}_i$ are useful for decoding. In general, if f is any degree d polynomial and one requires $I < d$ coefficients to encode any m, then $\mathbf{V}_i$ may consist of only low degree

coefficients of a single $v \in R_q$. This technique does not affect the DFR, improves efficiency, and cannot be worse for security.

Modulus Rounding. Rounding away the least significant bits of $\mathbf{B}, \mathbf{U}$, and $\mathbf{V}_i$ provides more compact $\mathsf{ek}, \mathsf{ct}_0$ and $\widehat{\mathsf{ct}}_i$ (respectively), but mechanically raises the DFR. Our goal is to minimize the size of $\widehat{\mathsf{ct}}_i$, so we will maximize the rounding on $\mathbf{V}_i$, while upper bounding the DFR. To do so we may round *fewer* bits from $\mathbf{B}$ or $\mathbf{U}$ to give us more DFR headroom. Thankfully, all else being equal, rounding $\mathbf{V}_i$ incurs a milder increase in the DFR than on $\mathbf{B}$ or $\mathbf{U}$. It also makes the numerous samples introduced by the $(\mathbf{V}_i)_i$ noisy enough to nullify Arora–Ge and BKW attacks, see Section 4.5.3.

Increasing the Modulus. All else being equal, increasing the modulus q reduces the DFR and therefore allows one to perform more rounding. If this extra rounding is concentrated on the $(\mathbf{V}_i)_i$, the net effect on the size of each $\widehat{\mathsf{ct}}_i$ is to decrease it. On the other hand, it slightly increases the size of ct_0 and ek and, more importantly, decreases the error rate, making lattice attacks more efficient.

Error-Correcting Codes (ECCs). Whenever in Figure 4.26 we want to encrypt κ bits, for $\kappa < |\mathsf{m}|$, we can use an ECC, i.e. $\mathsf{Encode}(\mathsf{m}) = \mathsf{Encode}(\mathsf{ECC}(\kappa))$, and lower the DFR. However, this method can lead to attacks when improperly implemented [DAn+19] or analyzed [GJY19; DVV19]. In addition, if the goal is to minimize $|\widehat{\mathsf{ct}}_i|$, then coefficient dropping seems to always be a safer and more efficient alternative. Hence we will not employ ECCs.

4.5.2 New Parametrizations

Given the methods outlined in Section 4.5.1, we make a number of alterations to the NIST Level I parameters of FrodoKEM, Kyber, and NTRU LPRime. In each case we maintain the *spirit* of the original design by e.g. keeping unique features. In all cases the new schemes satisfy the cryptanalytic model specified in Section 4.5.3, see also Table 4.10 for concrete security estimates against a number of attacks.

Note that the number of bits of shared secret encoded in $\mathbf{V}$ differs in these KEMs; Frodo640 encodes 128, whereas all parameter sets of Kyber and NTRU LPRime encode 256. For the purpose of fair comparison, in all cases we encode 128 bits. We note that in the case of Ilum512 and LPRime757, encoding 128 bits rather than 256 automatically reduces $|\widehat{\mathsf{ct}}_i|$ from 128 bytes to 64. Reductions below this size are a result of the techniques outlined in Section 4.5.1.

More subtle changes are discussed in Section 4.5.3, we briefly present them in this paragraph. For each scheme we give a table comparing (in the notation of the original scheme) the old and new parameter sets. We also give a dictionary of the form $\{\texttt{Figure: value}\}$, where `Figure` is a parameter from Figure 4.26 and `value` either comes from the relevant table or is defined in prose. The tables and descriptions of $D_{e'}$ and $D_{e''}$ in this section do not reflect wider error distributions implied by modulus rounding. We also discuss the effect of modulus rounding on security and decryption failures in Section 4.5.3. The savings achieved by our new parametrizations are given in Table 4.6.

Kyber. We introduce a new parameter set, Ilum512. We apply one less bit of rounding to $\mathbf{U}$, and one more to $\mathbf{V}$. We also drop coefficients from $\mathbf{V}$, see Table 4.7. Although altering q allowed other parametrizations, ring arithmetic over R_q consistently represents a significant fraction of the effort involved in providing embedded implementations of Kyber [Alk+20; XL21]. Keeping the same ring R_q as Kyber helps make Ilum512 fast and easy to deploy. Letting B_η be the binomial distribution over R defined in [Sch+20], we have $\{R : \mathbb{Z}[x]/(x^{256}+1), n : k, q : q, \bar{n} = \bar{m} : 1, D_s = D_e : B_{\eta_1}, D_{e'} = D_{e''} : B_{\eta_2}\}$.

FrodoKEM. We introduce a new parameter set, Bilbo640. Compared to Frodo640, Bilbo640 introduces aggressive rounding on $\mathbf{V}$, which has a positive effect on both the bandwidth cost and the security. To mitigate the effect on the DFR, we increase q to 2^{16}. We use a slightly larger new error distribution, χ_{Bilbo640},

TABLE 4.7: Parameter sets of Kyber512 and Ilum512, using the notation of [Sch+20], we drop $n - I$ coefficients.

Scheme	n	k	q	η_1	η_2	d_u	d_v	I
Kyber512	256	2	3329	3	2	10	4	256
Ilum512	256	2	3329	3	2	11	3	128

TABLE 4.8: Parameter sets of Frodo640 and Bilbo640, using the notation of [Nae+20], plus b/s to denote the random bits needed to sample an integer coefficient, and $\{\mathsf{D}_{\mathbf{B}}, \mathsf{D}_{\mathbf{U}}, \mathsf{D}_{\mathbf{V}}\}$ to denote the bits/coefficient in $\{\mathbf{B}, \mathbf{U}, \mathbf{V}\}$ (instead of a common D in [Nae+20]).

Scheme	n	$\mathsf{D}_{\mathbf{B}}$	$\mathsf{D}_{\mathbf{U}}$	$\mathsf{D}_{\mathbf{V}}$	σ	B	I	$\bar{m}$	$\bar{n}$	b/s
Frodo640	640	15	15	15	2.8	2	128	8	8	16
Bilbo640	640	16	16	3	2.9	2	128	8	8	32

which requires 32 bits of randomness per sample, see Table 4.8. We have $\{R : \mathbb{Z}, n : n, q : 2^{16}, \bar{n} : \bar{n}, \bar{m} : \bar{m}, D_s = D_e = D_{e'} = D_{e''} = \chi_{\mathsf{Bilbo640}}\}$.

NTRU LPRime. We introduce a new parameter set, LPRime757. We reduce the number of bits per entry of $\mathbf{V}$ from 4 to 2, and must increase the modulus, and decrease the weight, to account for this, see Table 4.9. The authors of NTRU LPRime [Ber+20] place a great emphasis on having $(x^p - x - 1)$ irreducible in $\mathbb{Z}_q$ and a DFR equal to zero. This is also the case for LPRime757.

We slightly alter the rounding function Top to Top′ which maintains perfect correctness while allowing us a larger weight than otherwise, see Section 4.5.3. We keep the original Right. As NTRU LPRime uses rounding for its errors the syntax of Figure 4.26 is not strictly correct, and we will report the errors induced by rounding. Let Short define the distribution that samples uniformly from the set Short of [Ber+20], let X assign probability $(q-1)/3q$ to ± 1 and $(q+2)/3q$ to 0, and let Y denote the probability mass function for a particular error value $\mathsf{Right}(\mathsf{Top}'(C)) - C$ over all $C \in \mathbb{Z}_q$. We have $\{R : \mathbb{Z}[x]/(x^p - x - 1), n = \bar{n} = \bar{m} : 1, q : q, D_s : \mathsf{Short}, D_e = D_{e'} : X, D_{e''} : Y\}$.

A Note on Isogeny-Based mPKE**s.** One of our instantiations of Chained CmPKE uses a mPKE variant of SIKE proposed in [Kat+20]. Bandwidth-wise, it seems asymptotically optimal, as $\widehat{\mathsf{ct}}_i$ is κ bits. Security-wise, [Kat+20] provides a security reduction to the SSDDH problem [FJP14], with a loss of $1/N$ in the advantage. This security loss is minimal: concretely, it means that using mPKE-SIKE with N recipient loses at most

TABLE 4.9: Parameter sets of LPRime653 and LPRime757, using the notation of [Ber+20]. We drop $p - I$ coefficients from $\mathbf{V}$.

Scheme	p	q	w	δ	τ	I
LPRime653	653	4621	252	289	16	256
LPRime757	757	7879	242	2001	4	128

$\lceil \log N \rceil$ bits of security compared to one recipient, which is small even for large groups. A downside of using SIKE is its slower running time, see Figure 4.29.

4.5.3 Cryptanalysis in the mPKE setting

In this section, we describe our cryptanalytic model, how it differs from a more standard cryptanalytic model for PKEs, and how we incorporate parts of the cryptanalysis of the schemes for which we have provided alternative parametrizations in Section 4.5.2. At a high level, the main difference is the availability in the mPKE setting of many additional 'samples' (see below) to an adversary from the $N-1$ ciphertexts $\widehat{\mathsf{ct}}_i$.

In the PKE setting $N = 2$, whereas we consider up to $N = 2^{16}$ in the mPKE setting. The number of extra available samples becomes considerable when taking $N = 2^{16}$. For example, an adversary attacking Frodo640 has $640 + 8 = 648$ samples available for a row of $\mathbf{R}$ in the PKE case. This becomes $640 + 8 \cdot (N-1) \approx 2^{19}$ in the $N = 2^{16}$ Bilbo640 case, though these extra samples have different properties to the original 640 in our new parametrization. In particular, they have much larger errors, due to the modulus rounding.

Nonetheless, these extra samples require us to consider two attacks that are usually absent from concrete security analyses of lattice PKEs, namely the sample-heavy Arora–Ge and BKW style attacks.

We first describe the number of samples available to an adversary in more detail, and the error these samples have. We then describe the three attacks we are considering, following broadly from the cryptanalysis present in several NIST final round lattice candidates, but adding Arora–Ge and BKW style attacks. After this, in Section 4.5.3 we describe our cryptanalytic model, which targets NIST Security Level I. A scheme satisfying this definition of security should have security comparable to AES-128, both against classical and quantum adversaries. Finally, we discuss in more depth the following aspects of our new parametrizations; any subtle algorithmic changes beyond the reparametrizations, any ways in which our cryptanalysis significantly differs from what is present in their respective submission documents, and where we have opted to incorporate parts of their individual cryptanalyses. We reference Figure 4.26 throughout, and in particular let $R = \mathbb{Z}[x]/(f)$ and $d = \deg(f)$, noting that we can recover Bilbo640 by setting $f(x) = x$.

Samples and Attacks

Samples. We talk of two distinct types of samples, a sample for $\mathbf{S}$ or a sample for $\mathbf{R}$. (See Figure 4.26). The number of samples given for (a single column of) $\mathbf{S}$ is $d \cdot n$, and they come from $\mathbf{B}$. Only Bilbo640 has $\bar{n} > 1$, and these extra columns of $\mathbf{S}$ can be accounted for by a hybrid argument. We can similarly count the samples for $\mathbf{R}$. The number of samples given for (a single row of) $\mathbf{R}$ is $d \cdot n$ from $\mathbf{U}$ and $d \cdot \bar{n} - C$ from $\mathbf{V}$. Here C represents the number of coefficients dropped, so in particular $d \cdot \bar{n} - C$ is 128 for LPRime757 and Ilum512, and 8 for Bilbo640. Again it is only Bilbo640 for which $\bar{m} > 1$, and a similar hybrid argument can account for these extra rows of $\mathbf{R}$. The error for a sample is given by the appropriate choice from D_e, $D_{e'}$ and $D_{e''}$, for samples from $\mathbf{B}$, $\mathbf{U}$ and $\mathbf{V}$ respectively, plus any modulus rounding that may be applied to $\mathbf{U}$ or $\mathbf{V}$. Note that, for NTRU LPRime, all errors come from rounding, so there is no 'extra' modulus rounding.

We consider up to $N = 2^{16}$ users and reevaluate the following attacks; primal lattice, Arora–Ge with Gröbner bases, and Coded BKW. This value of N comes from [Oma+21, §2.4], i.e. we have chosen the smallest power of 2 such that $N \geq 50000$. For all three of these attacks, the standard deviations of the distributions from which errors and secrets are sampled play an important role, and we summarize them in Table 4.11. Our scripts to estimate the complexity of these attacks, along with various other tasks, are

available at `https://anonymous.4open.science/r/chained-cmpke-2C20/README.md`. These scripts make use of the lwe-estimator,[29] an automated estimator based on [APS15], for the Arora–Ge and BKW style attacks. For the primal lattice attack, we make use of the leaky-LWE-estimator,[30] see the Primal Attack paragraph below. We do not consider the dual lattice attack for the same reasons as argued in [Sch+20, §5.2.1], that is, the assumptions that make it competitive with the primal lattice attack in the core-SVP model are not compatible with recent advances in lattice sieving, i.e. the dimensions for free techniques [Duc18], used in the 'Beyond core-SVP hardness' model. We use this latter model in our primal lattice attack estimation. We also do not, beyond their inclusion in the NTRU LPRime estimation script,[31] consider hybrid attacks. In particular, we do not consider them against either Bilbo640 or Ilum512, as [Nae+20; Sch+20] do not consider them against either FrodoKEM or Kyber. A common theme throughout will be, though an adversary against mPKEs is granted a large number of extra samples, these extra samples are less useful than the majority of samples an adversary against an ordinary PKE would also receive, namely the $d \cdot n$ from either $\mathbf{B}$ or $\mathbf{U}$. Indeed, by a serendipitous turn of events, our desire to minimize $|\widehat{\mathrm{ct}}_i|$ also minimizes the usefulness of these extra ciphertexts from a cryptanalytic perspective. For example, performing more rounding on these ciphertexts increases their error, and performing as much coefficient dropping as possible reduces their number; the hope is that the potential new avenues for cryptanalysis are nullified by these facts.

Primal Attack. The primal attack embeds [Kan87; BG14] a vector containing the error, and possibly also the secret, of an LWE or NTRU instance as a unique short vector in a lattice. It then applies lattice reduction to retrieve this vector. The primal attack requires a small number of samples and can therefore be used against either $\mathbf{S}$ or $\mathbf{R}$. In particular the optimal primal attack requires some linear multiple $c_p \cdot d \cdot n$ of samples, and typically $c_p \in [1, 2]$. We will use the methodology espoused in [Nae+20; Sch+20] for the primal attack. This uses the NIST-round3 branch of the leaky-LWE-estimator, an implementation of [Dac+20] which studies the probabilistic behavior of the primal attack. In the case of attacking $\mathbf{S}$ we have $d \cdot n$ samples from $\mathbf{B}$, i.e. $c_p = 1$. In the case of attacking $\mathbf{R}$ we have $d \cdot n$ samples from $\mathbf{U}$ and numerous samples from the $\mathbf{V}_i$. Due to our heavy rounding on the $\mathbf{V}_i$, the errors are far larger than those on $\mathbf{U}$ after rounding, see $\sigma_{\mathsf{r}(e')}$ and $\sigma_{\mathsf{r}(e'')}$ in Table 4.11 for their respective standard deviations. To be conservative while using the above primal attack methodology we use the smaller $\sigma_{\mathsf{r}(e')}$ for all ciphertext errors when attacking $\mathbf{R}$. We increase the number of samples available until the complexity estimate converges, which always occurs for $c_p < 2$, and take the 'Attack Estimation via simulation + probabilistic model' estimate. If we fix $c_p = 1$, that is, use only the samples from $\mathbf{U}$, then our estimates increase by less than a factor of two; in short the primal attack makes effectively no use of the extra samples afforded to it in our setting, even if we artificially assume they have much narrower errors. Our adaptation of the NIST-round3 branch outputs both classical and quantum gate counts using the estimated values for lattice sieves given in [Alb+20].

Coded BKW. The BKW style of attacks against LWE originates from the first subexponential time algorithm against the LPN problem [BKW00]. They add samples together in such a way that the dimension of the instance is iteratively decreased, while keeping the error small enough to solve the final instance, for a practical explanation in the LWE case see [Alb+15b]. The BKW style attacks are sample-heavy, requiring super-polynomially many samples in $d \cdot n$. There are methods [Bud+20, §3.3] in the literature used to form new samples from already known samples, and some experimental evidence on small dimensional instances suggesting the increase in the number of samples required is small when these 'sample amplification' techniques are used [GMW21, p. VI]. We note that these results say that the *number*

[29] `https://bitbucket.org/malb/lwe-estimator/src/master/`.

[30] `https://github.com/lducas/leaky-LWE-Estimator/tree/NIST-round3`.

[31] `https://ntruprime.cr.yp.to/estimate-20200927.sage`.

of samples required does not grow too much when sample amplification techniques are used, not that the complexity of the attack remains the same. This is discussed more below. In any case, it is standard not to consider BKW style attacks when attacking $\mathbf{S}$. In the case of attacking $\mathbf{R}$ in an mPKE the picture becomes somewhat more mixed. We have $d \cdot n$ samples from $\mathbf{U}$ with error standard deviation $\sigma_{\mathsf{r}(e')}$ and $(N-1) \cdot (d \cdot \bar{n} - C)$ samples from the $\mathbf{V}_i$ with a larger error standard deviation $\sigma_{\mathsf{r}(e'')}$. Again, in the PKE case where $N = 2$, it is standard to assume that this is not enough samples to perform a BKW style attack. However, for larger N this number of samples may be sufficient, and as such a BKW adversary may use some combination of these samples with different errors. We, therefore, report the estimates given by the lwe-estimator for the cost of the Coded-BKW [GJS15] attack assuming first that the adversary has access to unlimited samples 'from $\mathbf{U}$', and second that the adversary has access to unlimited samples 'from the $\mathbf{V}_i$', and assume that the cost for Coded-BKW lies somewhere between these estimates.

In general, there is limited experimental data on the performance of the numerous BKW variants against LWE, especially on medium-sized instances. Theoretical works focus on parametrizations that use standard deviations well above what is seen in practical schemes and assume infinitely many fresh samples, although BKW does perform favorably to lattice attacks in asymptotic settings [HKM18; Guo+19]. We also note that there have been some improvements to BKW style attacks since Coded-BKW. In particular, there has been Coded-BKW with sieving [Guo+17], which also allows quantum speedups to be incorporated during the sieving subroutine and a number of other improvements [Bud+20]. The above and the lack of publicly accessible estimation scripts for these new approaches make it difficult to precisely cost this attack against the parametrizations we suggest. We will appeal to the limited simulated and experimental results of the most recent practical study [Bud+20], namely Table 2 and Table 3, respectively. In Table 2 we see the primal attack remaining the most efficient for all simulated parameters in the low error rate setting, in particular for $\alpha = \sigma/q = 0.005$. If we restrict the primal attack to using only the samples from $\mathbf{U}$ then the highest error rate of our three parametrizations is $1.18/3329$, more than an order of magnitude smaller. We recall that the primal attack makes effectively no use of samples from the $\mathbf{V}_i$. We also note that the BKW complexities estimated here assume access to an unlimited amount of samples. Looking at experimental data, the required number of samples from [Bud+20, Tab. 3] suggests that significant sample amplification would be required, e.g. for the $(n, \alpha) = (40, 0.005)$ case with 40 available samples from $\mathbf{U}$, one is required to combine 6-tuples of samples as in [Bud+20, §3.3] to receive the required 45000000 samples. When assuming an unbounded number of fresh samples [Bud+20, Tab. 3] reports that attacking these parameters takes 12 minutes. The same work reports on solving the same instance, but is limited to 1600 samples.[32] They therefore only need to take triples of samples to receive the required number and report on some subtle difficulties encountered when creating enough triples of the correct form. This attack using triples for their sample amplification is reported to take over 3 hours. This increase in time complexity can be explained by an increase in the error standard deviation by a factor of $\sqrt{3}$ due to the sample amplification. We note the experiments of [GMW21, p. VI] mentioned earlier lowered the error standard deviation by this factor before performing sample amplification to examine the effect on the required number of samples in isolation. In the more realistic setting of an adversary receiving $d \cdot n$ samples from $\mathbf{U}$, and therefore having to perform more sample amplification, we assume the complexity increase will be greater still.

In conclusion, depending on the relative sizes of $\sigma_{\mathsf{r}(e')}$ and $\sigma_{\mathsf{r}(e'')}$ an adversary will choose to perform a certain amount of sample amplification on the samples from $\mathbf{U}$, and potentially subsequently use samples from the $\mathbf{V}_i$. In either case, we expect the estimate we produce for an adversary given unlimited samples 'from $\mathbf{U}$' will be an underestimate of the complexity of a BKW-style attack. In general, more experimental

[32] This value is n^2 and comes from `https://www.latticechallenge.org/lwe_challenge/challenge.php`. It does not represent a lattice scheme.

work is needed to understand the performance of BKW variants in medium-sized instances, using limited numbers of samples. We also note that the practical implications of sample amplification techniques in the ring setting [Sta20], or whether the rounding we apply affects the algebraic structure they use, have not been investigated.

Arora–Ge with Gröbner Bases. The Arora–Ge attack [AG11] is a linearization attack that, by knowing the support of the error distribution, is able to create a linear system such that part of the solution encodes the secret. It then attempts to solve this linear system, in the original work by matrix inversion, and in the work that followed [Alb+15a] by using Gröbner bases. The best known Arora–Ge style attacks require a superlinear number of samples [Alb+15a] in $d \cdot n$, even in the case of the bounded error, and therefore can only be used against $\mathbf{R}$. The complexity of the linear system to be solved is very sensitive to the support size of the error distributions being considered, intuitively explaining why our heavily rounded extra samples do not give us a practical attack. We again use the lwe-estimator and can take into account the differences between the errors of ct_0 and the $\widehat{\mathsf{ct}}_i$. If an adversary uses M of its available samples with error from some distribution D, we calculate the expected number e of distinct elements of $\mathsf{Supp}(D)$ that are sampled in these M samples. We assume the adversary can guess with probability one which e elements of $\mathsf{Supp}(D)$ have been sampled, and restrict the support of the error distribution to have size e for this estimate, making the attack cheaper. We always assume the adversary will use all the samples from $\mathbf{B}$ and $\mathbf{U}$, and then increase the number of samples used from the $\mathbf{V}_i$, reporting the lowest complexity. For our parametrizations the most efficient Arora–Ge adversary uses very few of the samples available from the $\mathbf{V}_i$, in particular never more than those given in $N = 3$ users case. In the case of Frodo640, where there is no rounding on the $\mathbf{V}_i$, the most efficient Arora–Ge attack makes use of *all* the samples available from $N = 2^{16}$ users. While it is still secure against the attack (the estimated complexity is 2^{3193}), it shows the positive effect that rounding the modulus, and therefore increasing the size of the error support, has against the Arora–Ge attack. To make this effect more extreme we give an artificial 'Kyber like' parameter set which is Kyber512 except $D_s = D_e = B_2$, $D_{e'} = D_{e''} = B_1$ and somehow the implementer forgets to include *any* modulus rounding. We of course stress that these are not parameters suggested in [Nae+20], and even if they were, they would not have been suggested for the mPKE setting. Even so, these parameters are almost secure under our primal attack estimation methodology, at an estimated 2^{133} classical gates. It might be that hybrid attacks are relevant for these parameters, but assuming they are in a similar ballpark, then our Arora–Ge estimate, which suggests a complexity of 2^{62}, is far cheaper. Again, it uses all possible samples from the $\mathbf{V}_i$ as they have no modulus rounding, and shows in theory the necessity of a cryptanalytic model tailored to the mPKE setting.

The Cryptanalytic Model

Here we introduce the requirements we make for an mPKE scheme to be called secure. It is effectively the same model that NIST laid out in their call for proposals [NIS17] but we take into account sample-heavy attacks, and the impact of having many more samples than usual, as described above.

Cost of Attacks. We require an mPKE to be parametrized such that none of the attacks listed above give costs (whether in gate count, or in the 'ring operations' reported by the lwe-estimator) of less than 2^{143} classically, and 2^{117} quantumly (where appropriate).

These gate counts are from [NIS17] and [Jaq+20]. Indeed, for Security Level I, [NIS17] requires 2^{143} classical gates, and, using the updated values of [Jaq+20, Tab. 12] requires 2^{117} quantum gates. We note the strange phenomenon that the lower the MAXDEPTH allowed to a quantum computer, the harder the quantum gate count requirement becomes to satisfy. This follows from the poor parallelizability of quantum search, and therefore the more constrained the depth of quantum computation, the more it must

TABLE 4.10: Security estimations of each scheme against well-known attacks. All values are given as log base 2. The columns P-S-c, P-S-q, P-R-c, and P-R-q represent the classical primal attack against **S**, the quantum primal attack against **S**, the classical primal attack against **R**, and the quantum primal attack against **R**, respectively. The columns BKW-U and BKW-V represent the Coded-BKW attack assuming unlimited samples 'from **U**' and 'from the $\mathbf{V}_i$', respectively. The column AG represents the Arora–Ge with Gröbner bases attack.

Scheme	P-S-c	P-S-q	P-R-c	P-R-q	BKW-U	BKW-V	AG	DFR
Bilbo640	164	154	163	154	224	334	4601	-129
Ilum512	151	143	150	142	157	224	2227	-125
LPRime757	177	166	177	166	184	259	1493	$-\infty$

rely on parallelization, and the less efficient it becomes. In our case, this means that as the MAXDEPTH decreases, breaking AES-128 becomes harder, see [Jaq+20] for detailed exposition. One could therefore argue that taking a smaller MAXDEPTH could render our parametrizations insecure with respect to quantum gate count, however, we follow [NIS17] in setting the minimum considered MAXDEPTH as 2^{40}. See the discussion [Sch+20, §5.3] for the potential impact of refinements to the primal attack on our gate count estimates.

Decryption Failure Rate. We require an mPKE to be such that the DFR remains below 2^{-120}, the largest of a final round lattice KEM [DAn+20, Tab. 1]. The largest DFR of any of our parametrizations is 2^{-125} for Ilum512. We note that for classical PKEs the DFR is often 0, that is, they exhibit perfect correctness. This is also the case for NTRU LPRime and our reparametrization thereof. The DFR is formally defined as the amount the expectation in Definition 2.9.3 differs from 1. In the lattice PKEs with non zero DFR, a decryption failure can be used within *reaction attacks* [HGS99; GJS16; DRV20] to learn information about the secret. Decryption failures also make future decryption failures easier to trigger [DRV20]. Even successful decryptions can be used to inform the search for decryption failures [BS20a]. Therefore, PKEs which are not parametrized to have perfect correctness instead aim to minimize their DFR. The concrete effect of the DFR in the mKEM setting is described by [Kat+20, Thm. 4.1]. We leave as an open research problem the concrete importance of the DFR in the CmPKE setting.

The results of our security estimation are given in Table 4.10. In all cases, it is the primal lattice attack that remains the most efficient. We do not cost quantum variants of Coded-BKW or Arora–Ge with Gröbner bases, and leave this as future work. Below we discuss each of the new parametrizations in turn. In particular, we discuss any subtle algorithmic changes and any differences (beyond the newly considered attacks) with their original cryptanalyses. We also mention that any elements of their original cryptanalyses that we are able to incorporate are not required to satisfy our cryptanalytic model for mPKEs.

Ilum512. We make one substantive change in our cryptanalysis of Ilum512 compared to Kyber512. It comes from the different amounts of rounding on **U** and the $\mathbf{V}_i$, which firstly increases the DFR to 2^{-125}. More importantly, though, we no longer satisfy the arguments of [Sch+20, §4.4] regarding estimating the primal attack on **R** with equal standard deviation for the secret and error distributions. That is, as we have reduced the amount of rounding on **U**, we have $\sigma_{\mathsf{r}(e')} < \sigma_s \approx 1.22$, where the latter standard deviation is of the distribution used to generate **R**. Using the reduction of [App+09] which allows one to sample the secret of an LWE instance from the same distribution as the error (and ignoring the samples this costs), we must therefore assume the elements of **R** are also drawn from the distribution with the smaller standard

TABLE 4.11: Standard deviations for the various secret and error distributions, see Figure 4.26. The values $\sigma_{\mathsf{r}(e')}$ and $\sigma_{\mathsf{r}(e'')}$ denote the standard deviation of errors after rounding $\mathbf{U}$ and $\mathbf{V}$ respectively. As LPRime757 uses rounding for all errors, we report these errors as $\sigma_e, \sigma_{e'}$, and $\sigma_{e''}$.

Scheme	σ_s	σ_e	$\sigma_{e'}$	$\sigma_{e''}$	$\sigma_{\mathsf{r}(e')}$	$\sigma_{\mathsf{r}(e'')}$
Bilbo640	2.91	2.91	2.91	2.91	2.91	2364
Ilum512	$\sqrt{3/2}$	$\sqrt{3/2}$	1	1	1.18	120
LPRime757	$\sqrt{242/757}$	$\sqrt{2/3}$	$\sqrt{2/3}$	568	$\sigma_{e'}$	$\sigma_{e''}$

deviation $\sigma_{\mathsf{r}(e')}$. This must be taken into account for all three attacks we consider.

Bilbo640. Other than increasing the number of bits of randomness required to sample from $D_s, D_e, D_{e'}$, and $D_{e''}$ (see below for an explanation why) and performing modulus rounding on the $\mathbf{V}_i$, we make no substantive changes to the algorithms of FrodoKEM to attain Bilbo640. Nor, other than including Arora–Ge and BKW style attacks, do we make any changes to our cryptanalysis. We may however reuse part of the security methodology of FrodoKEM. For example, we may wish to appeal to [Nae+20, Thm. 5.9], which relates the IND-CPA security of the PKE to the LWE problem, for up to our $N = 2^{16}$ users. As in [Kat+20, Lem. 5.1], modulo notational differences, we can adapt this to the mPKE setting as follows

$$\mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{mPKE},N}(\mathcal{A}) \leq N \cdot \bar{n} \cdot \mathsf{Adv}^{\mathsf{LWE}}_{n,n}(\mathcal{B}_1) + \bar{m} \cdot \mathsf{Adv}^{\mathsf{LWE}}_{n,n+N\bar{m}}(\mathcal{B}_2).^{33} \tag{4.1}$$

Intuitively we have a hybrid over $N \cdot \bar{n}$ columns of $\mathbf{S}$, each having n samples, and a hybrid over $\bar{m}$ rows of $\mathbf{R}$, each having $n + N \cdot \bar{n}$ samples. For both $\mathbf{S}$ and $\mathbf{R}$ the columns and rows, respectively, are secrets of length n. Conservatively, therefore, in our setting, we may take the larger of the two advantages and multiply by $N\bar{n} + \bar{m}$ to upper bound the advantage against the mPKE . From Table 4.10 we assume a uniform t gate classical adversary has an advantage no more than $2^{-163} \cdot t$ against either of the LWE problems, noting that we have gone from a Core-SVP estimate for this quantity in [Nae+20] to a gate count estimate here. Therefore a t gate IND-CPA $\mathcal{A}$ has an advantage of no more than $2^{-144} \cdot t$, and this mPKE is then a starting point for the constructions of Section 4.2.2. Another facet of FrodoKEM's security analysis we may wish to reuse is their Rényi divergence argument. The main security theorem of FrodoKEM [Nae+20, Thm. 5.1] regarding the IND-CCA security of the KEM, while not applicable here, accounts for the Rényi divergence between the actually sampled distribution χ_{Frodo} and the rounded Gaussian Ψ_s, as well as the number of samples drawn from χ_{Frodo}. The number of samples drawn from χ_{Frodo} is $2n\bar{n} + 2\bar{m}n + \bar{m}\bar{n} = 20554$, which increases to $2\bar{m}n + N \cdot (2n\bar{n} + \bar{m}\bar{n}) \leq 675293184$ for χ_{Bilbo640} in the mPKE setting with $N \leq 2^{16}$. As Theorem 4.2.7, the respective theorem for CmPKEs, does not proceed via a search problem, i.e. the OW-PCA problem of FrodoKEM, similar Rényi divergence arguments are not made. However, we give here a distribution to show the plausibility of efficiently sampling sufficiently close distributions in the CmPKE setting. Using the methods of [How+20, §5.2] we produce the following distribution χ_{Bilbo640}, which has a Rényi divergence of 2.144×10^{-10} from $\Psi_{2.9\sqrt{2\pi}}$[34] at order 200. It is symmetric around 0 and described in the following figure as $\{\pm x : p(\pm x) \cdot 2^{32}\}$,

[33] The notation of the advantage is borrowed from [Kat+20].

[34] We have the relation $s = \sigma\sqrt{2\pi}$ for Ψ_s.

- 0 : 587928496
- ±1 : 554318271
- ±2 : 464582536
- ±3 : 346126223
- ±4 : 229230439
- ±5 : 134950272
- ±6 : 70621314
- ±7 : 32851452
- ±8 : 13583937
- ±9 : 4992798
- ±10 : 1631188
- ±11 : 473696
- ±12 : 122271
- ±13 : 28052
- ±14 : 5720
- ±15 : 1037
- ±16 : 167
- ±17 : 24
- ±18 : 3

This means that by using exactly twice as much randomness to sample an element of χ_{Bilbo640} we can keep the $\exp(s \cdot \mathsf{D}_\alpha(P\|Q))^{1-1/\alpha}$ term of [Nae+20, Thm. 5.1] below its value in the Frodo640 case, even in the presence of these extra samples.

LPRime757. We make a small algorithmic change in LPRime757 compared to NTRU LPRime to reduce the size of the $\mathbf{V}_i$ and allow slightly larger weights w than otherwise. To reduce the size of $\mathbf{V}$ in LPRime757 we must ensure the rounding procedure Top has codomain $\{0, \ldots, \tau - 1\}$ for $\tau < 16$. In particular we define Top′ which achieves this for $\tau = 4$ as follows

$$\mathsf{Top}'(C) = (\tau_1(C + \tau_0) + 2^{15})/2^{16}, (\tau_0, \tau_1, \tau_2, \tau_3) = (3011, 33, 1995, 1978).$$

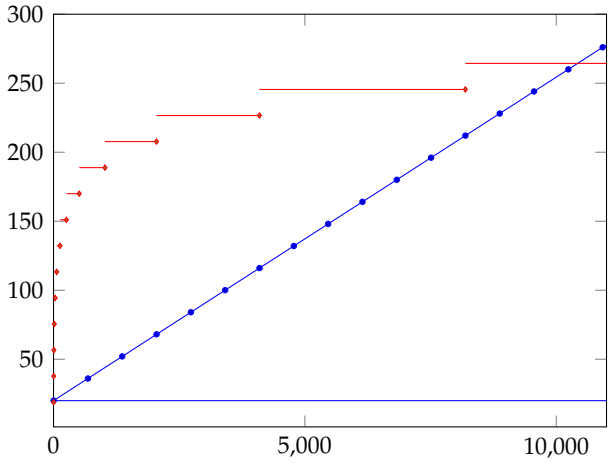
We note that the powers of 2 have each increased by one, compared to [Ber+20, §3.3]. This allows us a slightly larger weight w than otherwise. We do not alter Right. For our cryptanalysis, we calculate a 'per coefficient' variance for secret polynomials of NTRU LPRime. The secret polynomials of NTRU LPRime are degree d and have exactly w non-zero coefficients. These w positions are chosen uniformly and the value for each of them is independently and uniformly sampled from $\{-1, 1\}$, i.e. they are fixed weight, but not fixed sum. Given (w, d) and a fixed coefficient in an NTRU LPRime secret polynomial, its probability taken over all possible secret polynomials of being 0 is $1 - w/d$. Similarly, its probability of being either 1 or -1 is $w/2d$. We therefore calculate the variance using $p(\pm 1) = w/2d$ and $p(0) = 1 - w/d$. We reuse part of the security methodology of [Ber+20]. In particular we choose a weight w such that $1/4 \leq w/d \leq 1/2$, and parameters that satisfy both 'bulletproof' definitions for Level I security.[35] We also note that by the necessary alterations to Equation (4.1) in the LPRime757 mPKE case we can absorb the hybrid loss factor of $N + 1$.

4.6 Instantiation and Implementation of Chained CmPKE

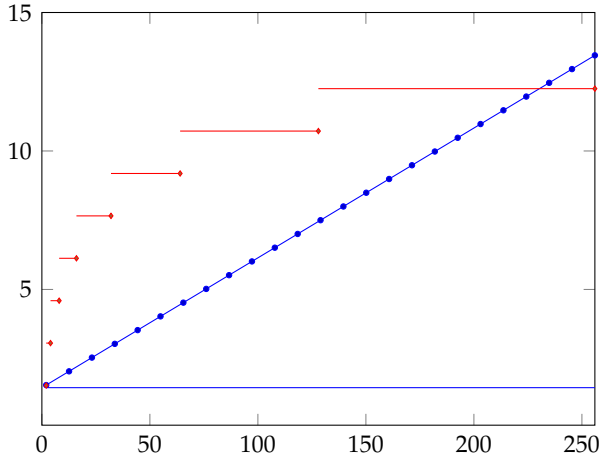
We instantiate Chained CmPKE as follows:

- *One-time* IND-CCA SKE. Since the message to be encrypted has $\kappa = 128$ bits, we may take plain AES-128 without a need for a mode. If we model plain AES as a pseudorandom permutation (PRP), then it satisfies one-time IND-CCA security Definition 2.3.4. We then obtain key-commitment property by applying [Alb+22, Sec. 5.2.].
- *Signature scheme.* We choose Dilithium for two reasons: (a) its performances are well-balanced, (b) it claims sEUF-CMA security from standard lattice assumptions [Lyu+20].

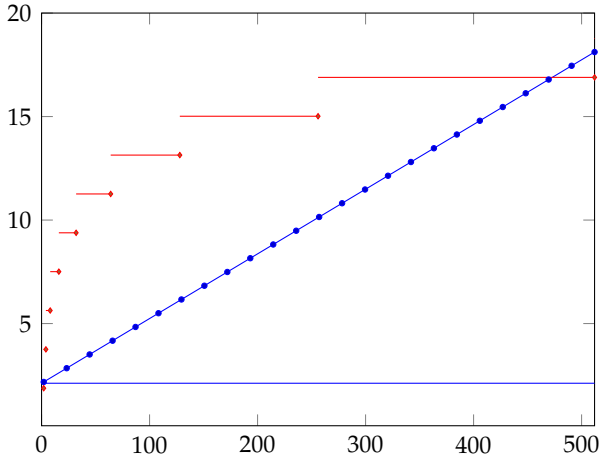
[35] `https://ntruprime.cr.yp.to/estimate-20200927.sage` using `run(757, 7879, 242, 'product')`.



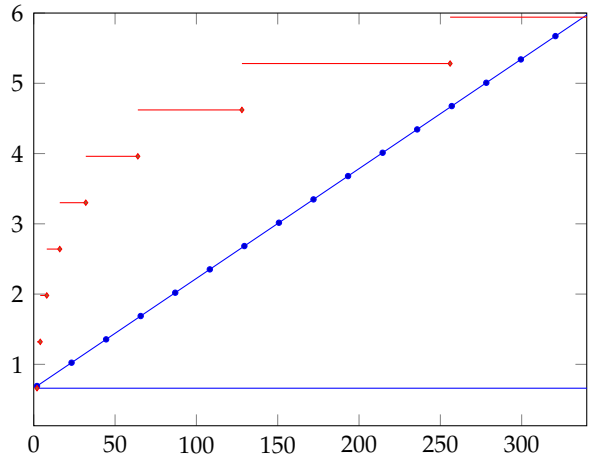
(A) Bilbo640 vs. Frodo640



(B) Ilum512 vs. Kyber512



(C) LPRime757 vs. LPRime653



(D) SIKEp434 vs. SIKEp434

FIGURE 4.27: The graphs "*X vs Y*" give the bandwidth overhead (in term of encryption keys and ciphertexts) of commit messages when using Chained CmPKE with the CmPKE X (Section 4.6 when uploaded, Section 4.6 when downloaded), compared to TreeKEM with the KEM Y (Section 4.6 *both* when uploaded and downloaded). The x-axis is the group size N, the y-axis is the overhead in KiB.

- mPKE. If we choose to rely on isogeny-based assumptions, we may use the SIKE mPKE from [Kat+20]. If we rely on lattice-based assumptions, we may use one of our three lattice-based mPKEs from Section 4.5: Bilbo640, Ilum512, LPRime757.

The mKEMs which are at the core of the mPKEs are implemented in C, starting from the optimized public platform-independent implementations of [Jao+20; Ber+20; Nae+20; Sch+20]. For Ilum512 and SIKEp434, the changes are straightforward. The modifications for Bilbo640 are only slightly more involved due to the new distribution and the Kyber-style compression. Finally, LPRime757 required most work: all encoding/decoding routines, rounding, Top and Barrett reduction had to be modified. We also improved polynomial multiplication performance, by computing them in the larger ring $GF(q')[x]/_{\langle 2^{p'}+1\rangle}$, with $q' = 1907713 > w(q-1)$ and $p' = 1536 = 3 \cdot 2^9$, which admits fast NTT-based multiplication as $3 \cdot 2^8 \mid q' - 1$. We do not use a full NTT, but leave out the layer corresponding to the factor 3 and multiply degree 2 polynomials in the NTT-domain, which is slightly more efficient than a full NTT. Chained CmPKE and the mPKEs are implemented in Go, using C bindings for the mKEMs.

Bandwidth Consumption. In Figure 4.28, we compare the total bandwidth overheads of TreeKEM and Chained CmPKE in terms of ciphertexts and encryption keys. For a better comparison, terms that are identical between both protocols, such as signatures, MACs, etc, are ignored. For readability, the bandwidth cost of each graph is normalized by the group size N. As predicted by the theory, the proposed protocol performs better than TreeKEM by factors $\Omega(\log N)$ for similar instantiations. In addition, while the size of our *uploaded* commit messages is asymptotically worse compared to TreeKEM ($O(N)$ vs $\Omega(\log N)$), in practice we compare favourably against comparable post-quantum instantiations of TreeKEM, even for groups of hundreds of users, see Figure 4.27.[36]

Computational Efficiency. In Figure 4.29, we provide timings for what we expect to be the two computational bottlenecks of the proposed protocol: `Commit` (Figure 4.29b) and `Process` (Figure 4.29c). We also provide timings for CmEnc (Figure 4.29a).

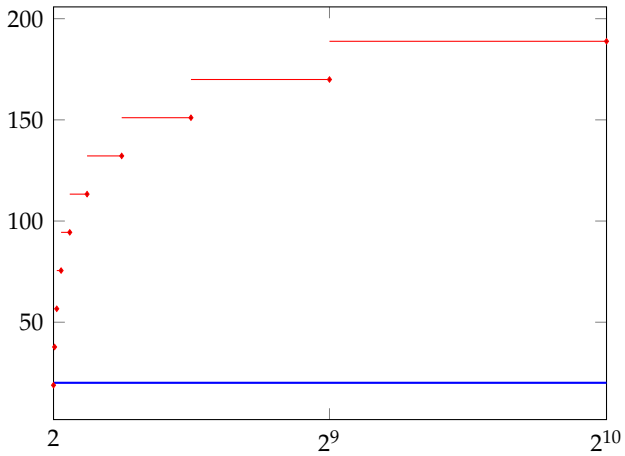
Even for group of 2^{10} members, lattice-based CmPKEs perform a multi-recipient encryption in less than 100 ms. This operation – and by extension, `Commit`– may take significantly longer when instantiating Chained CmPKE with SIKEp434 (about 7.5 s for 2^{10} recipients). Note however that `Commit` is a transparent operation for end users, and can be performed even when the end device is locked. We conclude from our measurements that the computational efficiency of Chained CmPKE is likely to have a minimal impact on the user experience.

Note that large groups also provide an amortization effect on the *computational* efficiency of CmPKEs. For example, encrypting a message to 2^{10} recipients with Bilbo640 (resp. Ilum512, LPRime757, SIKEp434) is about 29 (resp. 4, 3, 2) times faster than to perform 2^{10} encryptions. Finally, even though `Process` only entails a constant number of public-key operations, its running time eventually gets linear in N (Figure 4.29c), due to the hashing of N encryption keys when verifying the group state. This is also the case in TreeKEM, and can be mitigated to some extent by storing the hashes of the encryption keys.

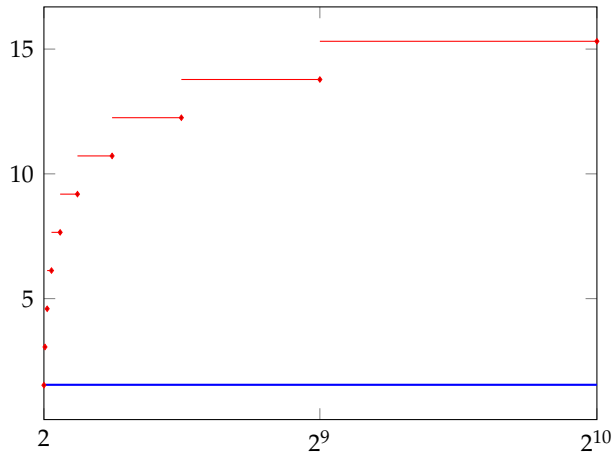
Code. Our code is available at the following repository:

`https://github.com/PQShield/chained-cmpke`

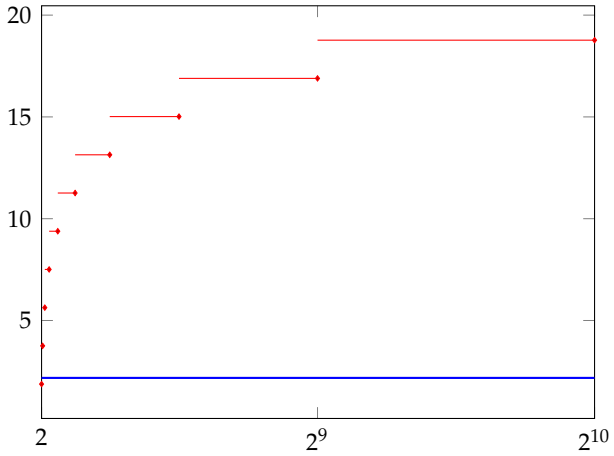
[36] In the absence of post-quantum parameter sets for TreeKEM in MLS, we came up with our own parameter sets relying on NIST PQC KEMs (finalists or alternate).



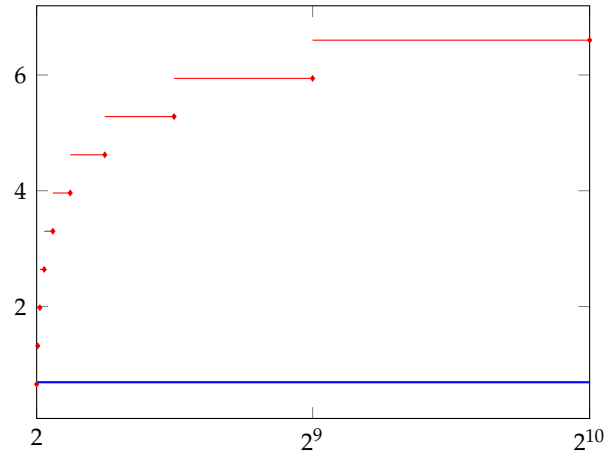
(A) Bilbo640 vs. Frodo640



(B) Ilum512 vs. Kyber512



(C) LPRime757 vs. LPRime653



(D) SIKEp434 vs. SIKEp434

FIGURE 4.28: The graphs "*X vs. Y*" (Figures 4.28a to 4.28d) give the *normalized* total bandwidth overhead (in term of encryption keys and ciphertexts) of a commit message with Chained CmPKE using the CmPKE X (Section 4.6), compared to TreeKEM using the KEM Y (Section 4.6). The x-axis is the group size N, the y-axis is the total bandwidth cost in KiB normalized by N. Graphs are computed using Tables 4.1 and 4.6.

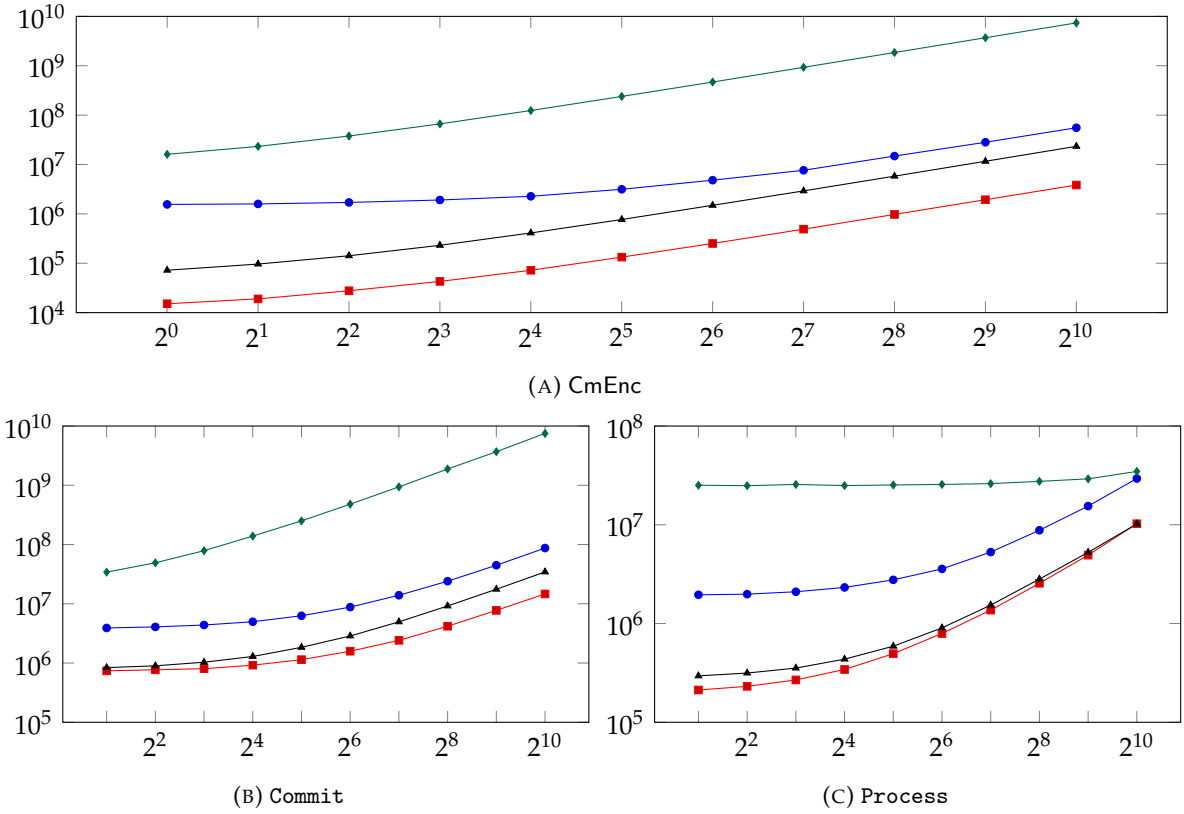


FIGURE 4.29: Running time in nanoseconds of some procedures as functions of the group size N, for Ilum512 (—■—), Bilbo640 (—●—), LPRime757 (—▲—) and SIKEp434 (—◆—). All measurements were obtained on an Apple M1 CPU @3.2 GHz (single-threaded).

4.7 A Variant of GSD Security Tailored to Chained CmPKE

We introduce a variant of the generalized selective decryption (GSD) security notion for public-key encryption tailored to our Chained CmPKE security proof, which we coin as a Chained CmPKE conforming GSD security. We then show that such variant is secure in the random oracle model assuming the hardness of the CmPKE. The formalization of our Chained CmPKE conforming GSD security is inspired by [Kle+21; Alw+20b; AJM22] but differs in the following way: we consider a (committing) multi-recipient encryption oracle rather than a single-recipient oracle; we restrict the hash oracles to *not* take as input the secrets used to generate the keys for CmPKE; the proof is simplified.[37] The restriction on the hash oracle is new to this work and effectively, this simplifies the proof while still being sufficient for proving our Chained CmPKE protocol.

In more detail, our Chained CmPKE conforming GSD is defined in Figure 4.30. The game maintains a graph with M-vertices, where each node u stores a seed s_u initially set to $\perp$. If u is a source node, then s_u is further used to generate a key pair $(\mathsf{ek}_u, \mathsf{dk}_u)$ for CmPKE (see `*gen-full-key-if-nec`). An edge corresponds to dependencies between seeds, where there are three types of edges. One edge is a (multi-recipient) encryption edge created by `CmEnc`: if there is an edge from a source node u leading into v, then s_v is encrypted using ek_u. The second edge is a hash edge created by `Hash`: if there is an edge from a non-source node u leading into v, then s_v is (informally) the output of a hash function H on input s_u. The final type of edge is a join hash edge created by `Join-Hash`: if there is an edge from non-source nodes u and u' leading into v, then s_v is (informally) the output of a hash function H on input s_u and $s_{u'}$. See Figure 4.1 for what these edges mean in the context of the CGKA protocol. In Figure 4.1, the solid edge corresponds to encryption edges and the dashed edges correspond to either hash or join hash edges. Observe that `Hash` does not take as input an encryption node (i.e., EncSource) and `CmEnc` does not take as input a (non-join) hashed input (i.e., $s_v = \perp$ or $v \in \mathsf{WelcomeNode}$). This restriction is sufficient to prove security of our Chained CmPKE protocol, while also having the benefit that the GSD security proof will be simplified.

The GSD adversary can adaptively create the edges of the graph and also adaptively corrupt the nodes to obtain the stored secret seed. The **gsd-exp** function determines if the secret seed in node v is trivially exposed to the adversary. Specifically, u is exposed if it is corrupted or can be traversed from any corrupted nodes. Then, the security of GSD states that as long as node u does not satisfy **gsd-exp** (i.e., does not trivially expose the secret), then the secret seed s_u remains hidden.

More formally, we define the security notion as follows.

Definition 4.7.1 (Chained CmPKE **Conforming GSD Security).** *The security notion is defined by a game illustrated in Figure 4.30, where $M = \mathsf{poly}(\kappa)$ is an integer. We say a* CmPKE *is* Chained CmPKE conforming GSD secure *if for any efficient adversaries $\mathcal{A}$, we have*

$$\left| \Pr\left[\mathsf{Game}^{\mathsf{GSD}}_{\mathsf{CmPKE},\mathcal{A}}(\kappa, M) = 1 \right] - \frac{1}{2} \right| \leq \mathsf{negl}(\kappa).$$

We say it is selectively Chained CmPKE *conforming GSD secure if $\mathcal{A}$ is required to commit to all of its oracle queries at the outset of the game.*

The following is the main theorem of this section.

Theorem 4.7.2. *A* CmPKE *is* Chained CmPKE *conforming GSD secure if* CmPKE *is* IND-CCA *secure with adaptive corruption. Additionally, it is selectively secure if* CmPKE *is only* IND-CCA *secure.*

[37] We also noticed that we would require oracles `Set-Secret` and `Set-Full-Secret` in the security proof.

$\mathsf{Game}^{\mathsf{GSD}}_{\mathsf{CmPKE},\mathcal{A}}(\kappa, M)$

1 : $(V, E) \leftarrow ([M], \emptyset)$
2 : $\mathsf{Corr}, \mathsf{Ctxt} \leftarrow \emptyset$
3 : $\mathsf{EncSource} \leftarrow \emptyset$
4 : $\mathsf{WelcomeNode} \leftarrow \emptyset$
5 : **foreach** $u \in [M]$ **do**
6 : $\quad s_u, \mathsf{ek}_u, \mathsf{dk}_u \leftarrow \bot$
7 : // challenge node
8 : $u^* \leftarrow \bot$
9 : $b \leftarrow\$ \{0,1\}$
10 : $s^* \leftarrow\$ \{0,1\}^\kappa$
11 : $\mathsf{pp} \leftarrow \mathsf{CmSetup}(1^\kappa)$
12 : $b' \leftarrow \mathcal{A}^{\mathcal{O}}(\mathsf{pp})$
13 : **req** (V, E) is acyclic
$\quad \wedge\, u^*$ is a sink
$\quad \wedge$ **gsd-exp**$(u^*) = 0$
14 : **return** $[\![b = b']\!]$

Oracle $\texttt{Chall}(u)$

1 : **req** $u^* = \bot$
2 : $u^* \leftarrow u$
3 : **if** $b = 0$ **then**
4 : $\quad$ **return** s_u
5 : **else**
6 : $\quad$ **return** s^*

Oracle $\texttt{Corr}(u)$

1 : **req** $s_u \neq \bot$
2 : $\mathsf{Corr} \mathrel{+}\!\leftarrow u$
3 : **if** $\mathsf{dk}_u = \bot$ **then**
4 : $\quad$ **return** s_u
5 : **else**
6 : $\quad$ **return** dk_u

Oracle $\texttt{Set-Secret}(u, s)$

1 : **req** $s_u = \bot$
2 : $s_u \leftarrow s$
3 : $\mathsf{Corr} \mathrel{+}\!\leftarrow u$

Oracle $\texttt{Set-Full-Secret}(u, s)$

1 : **req** $s_u = \bot$
2 : $s_u \leftarrow s$
3 : $(\mathsf{ek}_u, \mathsf{dk}_u) \leftarrow \mathsf{CmGen}(\mathsf{pp}; \mathsf{H}(s_u))$
4 : $\mathsf{Corr} \mathrel{+}\!\leftarrow u$

Oracle $\texttt{CmEnc}(S, v)$

1 : **foreach** $u \in S \subseteq [M]$ **do**
2 : $\quad$ **req** $s_u = \bot \vee u \in \mathsf{EncSource}$
3 : **req** $s_v = \bot \vee v \in \mathsf{WelcomeNode}$
4 : $\texttt{*gen-key-if-nec}(v)$
5 : **foreach** $u \in S$ **do**
6 : $\quad \texttt{*gen-full-key-if-nec}(u)$
7 : $\quad E \mathrel{+}\!\leftarrow (u, v, \texttt{'enc'}, \bot)$
8 : $(\mathsf{T}, \vec{\mathsf{ct}}) \leftarrow \mathsf{CmEnc}((\mathsf{ek}_u)_{u \in S}, s_v)$
$\quad$ // $\vec{\mathsf{ct}} = (\mathsf{ct}_u)_{u \in S}$
$\mathsf{Ctxt} \mathrel{+}\!\leftarrow (S, \mathsf{T}, \vec{\mathsf{ct}})$
9 : $\mathsf{EncSource} \mathrel{+}\!\leftarrow S$
10 : **return** $((\mathsf{ek}_u)_{u \in S}, \mathsf{T}, \vec{\mathsf{ct}})$

Oracle $\texttt{CmDec}(u, \mathsf{T}, \mathsf{ct})$

1 : **req** $u \in \mathsf{EncSource}$
2 : **foreach** $(S, \mathsf{T}', \vec{\mathsf{ct}}) \in \mathsf{Ctxt}$ **do**
3 : $\quad$ // $\vec{\mathsf{ct}} = (\mathsf{ct}_{u'})_{u' \in S}$
4 : $\quad$ **if** $u \in S \wedge (\mathsf{T}, \mathsf{ct}) = (\mathsf{T}', \mathsf{ct}_u)$ **then**
5 : $\quad\quad$ **return** $\bot$
6 : **return** $\mathsf{CmDec}(\mathsf{dk}_u, \mathsf{T}, \mathsf{ct})$

Oracle $\texttt{Hash}(u, v, \mathsf{lbl})$

1 : **req** $u \notin \mathsf{EncSource} \wedge s_v = \bot$
2 : $\texttt{*gen-key-if-nec}(u)$
3 : **req** $(u, *, \texttt{'hash'}, \mathsf{lbl}) \notin E$
4 : $s_v \leftarrow \mathsf{H}(s_u, \mathsf{lbl})$
5 : $E \mathrel{+}\!\leftarrow (u, v, \texttt{'hash'}, \mathsf{lbl})$

Oracle $\texttt{Join-Hash}(u, u', v, \mathsf{lbl})$

1 : **req** $u, u' \notin \mathsf{EncSource} \wedge s_v = \bot$
2 : $\texttt{*gen-key-if-nec}(u)$
3 : $\texttt{*gen-key-if-nec}(u')$
4 : **req** $((u, u'), *, \texttt{'join-hash'}, \mathsf{lbl}) \notin E$
5 : $s_v \leftarrow \mathsf{H}(s_u, s_{u'}, \mathsf{lbl})$
6 : $\mathsf{WelcomeNode} \mathrel{+}\!\leftarrow v$
7 : $E \mathrel{+}\!\leftarrow ((u, u'), v, \texttt{'join-hash'}, \mathsf{lbl})$

$\texttt{*gen-full-key-if-nec}(u)$

1 : **if** $(s_u, \mathsf{ek}_u, \mathsf{dk}_u) = (\bot, \bot, \bot)$ **then**
2 : $\quad s_u \leftarrow\$ \{0,1\}^\kappa$
3 : $\quad (\mathsf{ek}_u, \mathsf{dk}_u) \leftarrow \mathsf{CmGen}(\mathsf{pp}; \mathsf{H}(s_u))$

$\texttt{*gen-key-if-nec}(u)$

1 : **if** $s_u = \bot$ **then**
2 : $\quad s_u \leftarrow\$ \{0,1\}^\kappa$

gsd-exp(v)

1 : **return** $[v \in \mathsf{Corr}]$
$\quad \vee\, \exists (u, v, *, *) \in E$: **gsd-exp**(u)
$\quad \vee\, \exists ((u, u'), v, *, *) \in E$: **gsd-exp**$(u) \wedge$ **gsd-exp**(u')

FIGURE 4.30: Chained CmPKE conforming GSD game. The adversary $\mathcal{A}$ can access the list of oracles $\mathcal{O} := \{ \texttt{Chall}, \texttt{CmEnc}, \texttt{CmDec}, \texttt{Corr}, \texttt{Hash}, \texttt{Join-Hash}, \texttt{Set-Secret}, \texttt{Set-Full-Secret} \}$ and the random oracle H.

SecurePaths(u, L)

```
1 :  L ++← u
2 :  if ∃v s.t. (v, u, 'hash', *) ∈ E then     // u is connected only from a 'hash' edge
3 :    if gsd-exp(v) = 0 then                  // Such a v is unique if it exists
4 :      SecurePaths(v, L)
5 :  elseif u is not a source then
6 :    (v1, flag1, v2, flag2, S, flag3) ← (⊥, ⊥, ⊥, ⊥, ∅, ⊥)
7 :    if ∃(v, v') s.t. ((v, v'), u, 'join-hash', *) ∈ E then
8 :      (v1, v2) ← (v, v')    // Such (v, v') is unique if it exists
9 :      if gsd-exp(v1) = 0 then
10 :       flag1 ← ⊤
11 :     if gsd-exp(v2) = 0 then
12 :       flag2 ← ⊤
13 :   if ∃v s.t. (v, u, 'enc', *) ∈ E then
14 :     flag3 ← ⊤    // Remains ⊤ if all recipients are uncorrupted
15 :     foreach v s.t. (v, u, 'enc', *) ∈ E do
16 :       S ++← v
17 :       if gsd-exp(v) = 1 then
18 :         flag3 ← ⊥    // v is a corrupted (encryption) source
19 :   if (flag1 = ⊤ ∨ flag2 = ⊤) ∧ flag3 = ⊤ then
20 :     if flag1 = ⊤ then          // u is connected from 'join-hash' and 'enc' edges
21 :       SecurePaths(v1, L)
22 :     if flag2 = ⊤ then
23 :       SecurePaths(v2, L)
24 :     L ++← S
25 :     return L
26 :   elseif (flag1 = ⊤ ∨ flag2 = ⊤) ∧ S = ∅
27 :     if flag1 = ⊤ then          // u is connected only from a 'join-hash' edge
28 :       SecurePaths(v1, L)
29 :     if flag2 = ⊤ then
30 :       SecurePaths(v2, L)
31 :   elseif v1 = ⊥ ∧ flag3 = ⊤ then    // u is connected only from 'enc' edges
32 :     L ++← S
33 :     return L
34 :  elseif gsd-exp(u) = 0 then    // u is a non-corrupted source
35 :    return L
```

FIGURE 4.31: Helper function that outputs all the secure paths leading to the input node u, where $L = \emptyset$ by default.

Proof. We only consider the adaptive setting since downgrading the proof to the selective setting is straightforward. Our proof is in the non-programmable random oracle model.

To aid the proof, we define a helper function SecurePaths in Figure 4.31 which takes as input a node $u \in [M]$ and a list L. At the end of the game, if an adversary $\mathcal{A}$ had chosen a valid challenge node u^*, then $\mathsf{SecurePaths}(u^*, L := \emptyset)$ outputs all the "secure paths" that leads to u^*. To be more concrete, $\mathsf{SecurePaths}(u^*, L := \emptyset)$ outputs a set of lists $D = \{ L_k \}_{k\in[K]}$, where $K \leq M - 1$ and each L_k is either of the form $L_k = (u_1 := u^*, u_2, \cdots, u_{I_k})$ or $L_k = (u_1 := u^*, u_2, \cdots, u_{I_{k-1}}, S)$ for some integer I_k, nodes $u_i \in [M]$ such that $\mathbf{gsd\text{-}exp}(u_i) = 0$, u_{I_k} is a source node, and a set of nodes $S \subset [M]$ such that $\forall u \in S$, $u \in \mathsf{EncSource}$ and $u \notin \mathsf{Corr}$ (which in particular implies $\mathbf{gsd\text{-}exp}(u) = 0$). Intuitively, L_k is a path that consists of uncorrupt nodes (and possibly a set of nodes) that do not trivially leak the secret s^* associated to u^*. Note that it is not enough to simply check if a node is uncorrupted; even if v and v' such that $((v, v'), u, \texttt{'join-hash'}, *) \in E$ are uncorrupted, we must also check that all v'' such that $(v'', u, \texttt{'enc'}, *) \in E$ are uncorrupted as well, since otherwise, u's secret s_u may trivially leak.

Stating the above more formally, we obtain the following lemma.

Lemma 4.7.3. *The challenge sink u^* output by $\mathcal{A}$ is a valid challenge (i.e., $\mathbf{gsd\text{-}exp}(u^*) = 0$) if and only if $\mathsf{SecurePaths}(u^*) \neq \emptyset$.*

Proof. The proof simply consists of checking the conditions. The "only if" part of the proof is trivial since any path that satisfies $\mathbf{gsd\text{-}exp}(u^*) = 0$ is also a path that will be output by SecurePaths. Therefore, let us focus on the "if" part of the proof. Let us assume $\mathsf{SecurePaths}(u^*) \to D = \{ L_k \}_{k\in[K]}$. Consider any L_k of the form $L_k = (u_1 := u^*, \cdots, u_{I_k})$. (The case $L_k = (u_1 := u^*, \cdots, u_{I_{k-1}}, S)$ follows the same argument). By the definition of SecurePaths, it is clear that there is an edge between each adjacent nodes u_i and u_{i+1}. Moreover, we have $\mathbf{gsd\text{-}exp}(u_i) = 0$ for every $i \in [I_k]$. This can be verified by checking the nodes in reverse order; The final output u_{I_k} is a source node that satisfies $\mathbf{gsd\text{-}exp}(u_{I_k}) = 0$. For u_{I_k} to be output, $\mathsf{SecurePaths}(u_{I_k}, L'_k = (u_i)_{i\in[I_k-1]})$ must have been invoked within $\mathsf{SecurePaths}(u_{I_k-1}, L''_k = (u_i)_{i\in[I_k-2]})$. If (u_{I_k}, u_{I_k-1}) is connected by a $\texttt{'hash'}$ edge, then $\mathbf{gsd\text{-}exp}(u_{I_k-1}) = 0$ and we have that u_{I_k} is the only node connected to u_{I_k-1}. If (u_{I_k}, u_{I_k-1}) is connected by an $\texttt{'enc'}$ edge or by a $\texttt{'join-hash'}$ edge (i.e., $\exists$ u such that $((u_{I_k}, u), u_{I_k-1}, \texttt{'join-hash'}, *) \in E$), then all other $\texttt{'enc'}$ edges leading to u_{I_k-1} come from uncorrupted nodes. Therefore, this establishes $\mathbf{gsd\text{-}exp}(u_{I_k-1}) = 0$. We can repeat this argument until we reach $\mathsf{SecurePaths}(u_1 = u^*, L := \emptyset)$ to establish $\mathbf{gsd\text{-}exp}(u_i) = 0$ for every $i \in [I_k]$. This completes the lemma. □

Let $D = \{ L_k \}_{k\in[K]} \leftarrow \mathsf{SecurePaths}(u^*)$. There are two cases:

Case 1: $\exists L_k = (u_1 := u^*, \cdots, u_{I_k}) \in D$ such that for all $i \in [I_k]$, there does not exist v satisfying $(v, u_i, \texttt{'enc'}, *) \in E$.

Case 2: $\forall L_k \in D$, either $L_k = (u_1 := u^*, \cdots, u_{I_k})$ such that there exists $i \in [I_k]$ and v satisfying $(v, u_i, \texttt{'enc'}, *) \in E$ or $L_k = (u_1 := u^*, \cdots, u_{I_k-1}, S)$.

Note that when $L_k = (u_1 := u^*, \cdots, u_{I_k-1}, S)$, we have $S \subseteq \mathsf{EncSource}$ in Figure 4.30; that is, S is the set of nodes that were used to encrypt the secret $s_{u_{I_k-1}}$ associated to node u_{I_k-1}. The following Lemmata 4.7.4 and 4.7.5, each corresponding to Case 1 and Case 2, respectively, completes the proof of the theorem. Note that although the lemma assumes either Case 1 or Case 2 always hold, this is without loss of generality since the reduction can always guess which case we end up in.

Lemma 4.7.4. *If Case 1 occurs, then any adversary $\mathcal{A}$ making at most polynomially many oracle queries has negligible advantage against the* Chained CmPKE *conforming GSD security.*

Proof. When Case 1 occurs, there are no encryption edges coming into any of the nodes along the path $L_k = (u_1 := u^*, \cdots, u_{I_k})$. Therefore, since $\mathbf{gsd\text{-}exp}(u_i) = 0$ for all $i \in [I_k]$, this means all the associated secrets $(s_{u_i})_{i\in[I_k]}$ are information theoretically hidden from the adversary $\mathcal{A}$ until they are queried to the random oracle. This can be argued more formally as follow by induction: Since u_{I_k} is a (non-encryption) source node, $s_{u_{I_k}}$ is information theoretically hidden. In case (u_i, u_{i+1}) is connected by a `'hash'` edge, then if s_{u_i} is hidden, so is $s_{u_{i+1}}$ in the random oracle model. On the other hand, in case (u_i, u_{i+1}) is connected by a `'join-hash'` edge, then even if the other node u such that $((u_i, u), u_{i+1}, \text{`join-hash'}, *) \in E$ is corrupted, $s_{u_{i+1}}$ is hidden as long as s_{u_i} is. Finally, since I_k and the number of random oracle queries $\mathcal{A}$ makes is polynomial, the probability of $\mathcal{A}$ obtaining any information on $(s_{u_i})_{i\in[I_k]}$ is negligible. This in particular implies that $s_{u_1} := s^*$ is uniform random in the view of $\mathcal{A}$. This concludes the lemma, where we note that the adaptivity of $\mathcal{A}$ is irrelevant. □

Lemma 4.7.5. *If Case 2 occurs, then any PPT adversary $\mathcal{A}$ has negligible advantage against the* Chained CmPKE *conforming GSD security assuming the hardness of the* IND-CCA *security with adaptive corruption of* CmPKE*.*

Proof. Let $\mathcal{A}$ be an adversary against the Chained CmPKE conforming GSD security game that triggers Case 2. Consider the following three games where the first game corresponds to the real game depicted in Figure 4.30 and the last game is where no (possibly inefficient) adversary has a winning advantage. We denote S_i as the event that $b = b'$ occurs in the game Game i and show that each adjacent game is indistinguishable, thus establishing the hardness of the real game.

Game 0**.** This is the real game depicted in Figure 4.30.

Game 1**.** The challenger guesses a random challenge sink $u^* \leftarrow\!\$\, [M]$ and a challenge source $v^* \leftarrow\!\$\, [M]$ conditioned on $u^* \neq v^*$. It then proceeds exactly as in the previous game except that it outputs a random bit on behalf of $\mathcal{A}$ if either u^* was not the node $\mathcal{A}$ queries to the challenge oracle or if there does not exist a list $L_k \in D \leftarrow \mathsf{SecurePaths}(u^*)$ such that either $L_k = (u_1 = u^*, \cdots, u_{I_k} = v^*)$ or $L_k = (u_1 = u^*, \cdots, u_{I_k-1}, S)$, where $v^* \in S$. Without loss of generality, we assume there is always an incoming edge to u^*. Then, by Lemma 4.7.3, it is clear that $\Pr[\mathsf{S}_1] \geq \Pr[\mathsf{S}_0]/M^2$.

Game 2**.** This is the same as the previous game except that the challenger answers to oracle `CmEnc` differently for those nodes connected to the challenge source node v^*. More concretely, when $\mathcal{A}$ queries (S, v) to oracle `CmEnc`, the challenger checks whether the following conditions (denoted as SetRand) hold:

- Is $s_v = \perp$ and $v^* \in S$?
- Is $v \in \mathsf{WelcomeNode}$ and does there exist a set of edges in E that connects v^* to v?

If so, the challenger proceeds as in the previous game except that it samples a random message r_v and runs $\mathsf{CmEnc}(\mathsf{pp}, (\mathsf{ek}_u)_{u\in S}, r_v)$ instead of $\mathsf{CmEnc}(\mathsf{pp}, (\mathsf{ek}_u)_{u\in S}, s_v)$ on line 8. Otherwise, it is defined exactly as in the previous game. Intuitively, the challenger modifies all the incoming encryption edges to the secure path L_k to encrypt random values. Note that due to the way oracles `Hash`, `Join-Hash`, and `CmEnc` are defined, an input (S, v) that did not satisfy condition SetRand will remain unsatisfied since such a node v cannot be later connected to the secure path L_k.

Observe that Game 2 now boils down to the argument we made for Case 1 in Lemma 4.7.4 since all the incoming encryption edges to the secure path L_k to encrypt random values. Specifically, there either exists a list $L_k = (u_1 = u^*, \cdots, u_{I_k} = v^*)$ or a list $L_k = (u_1 = u^*, \cdots, u_{I_k-1}, S)$, where $v^* \in S$. In the former case, since all the incoming encryption edges into the nodes in the list L_k are encrypting random values, we can simply ignore them. This is the same for the latter case, where we additionally observe that

$\texttt{CmEnc}(S, u_{I_k-1})$ provides an encryption of a random value. Therefore, following the same argument made in Case 1, $\Pr[\mathsf{S}_2] = \mathsf{negl}(\kappa)$ even for a possible inefficient $\mathcal{A}$ that makes at most polynomially many queries.

To conclude the lemma, it remains to establish the bound between $\Pr[\mathsf{S}_1]$ and $\Pr[\mathsf{S}_2]$. We construct an adversary $\mathcal{B}$ against the IND-CCA security with adaptive corruption game of CmPKE that internally runs $\mathcal{A}$. Without loss of generality, we assume the "multi-challenge-ciphertext" variant where $\mathcal{B}$ can query polynomially-many challenge ciphertexts. By a basic hybrid argument, this variant is secure if the single-challenge version is secure. The description of $\mathcal{B}$ follows:

$\mathcal{B}^{\mathcal{C}(\cdot)}(\mathsf{pp}, (\mathsf{ek}_i)_{i\in[M]})$: Whenever $\mathcal{B}$ needs to generate a new encryption and decryption key pair $(\mathsf{ek}_u, \mathsf{dk}_u)$, it simply uses an unused ek_u provided by its challenge. When $\mathcal{A}$ queries a corruption oracle on u, if dk_u is set, then $\mathcal{B}$ queries u to its corruption oracle and returns the received dk_u. Otherwise, s_u is generated on its own so it simply outputs s_u. When (S, v) queried to oracle $\texttt{CmEnc}$ by $\mathcal{A}$ satisfies condition SetRand, then it samples a random message r_v and queries (s_v, r_v) as its challenge ciphertext. It then uses the provided challenge ciphertext to simulate $\texttt{CmEnc}$. Moreover, all oracle queries to $\texttt{CmDec}$ can be answered by using its decryption oracle. Finally, when $\mathcal{A}$ makes a random oracle query, $\mathcal{B}$ simply relays this to its own random oracle.[38] $\mathcal{B}$ answers all other oracle queries, i.e., $\texttt{Set-Secret}, \texttt{Set-Full-Secret}, \texttt{Hash}, \texttt{Join-Hash}$ on its own.

It can be checked that when $\mathcal{B}$ receives challenge ciphertexts for random messages, then the game it simulates is identical to Game 2. Otherwise, it is identical to Game 1. Therefore, assuming the hardness of the IND-CCA security with adaptive corruption of CmPKE, we have $|\Pr[\mathsf{S}_1] - \Pr[\mathsf{S}_2]| \leq \mathsf{negl}(\kappa)$.

Combining all the bounds, we have $\Pr[\mathsf{S}_0] = \mathsf{negl}(\kappa)$ as desired. This completes the lemma. □

□

Remark 4.7.6 (Adaptive security from CmPKE with no adaptive corruption). In the above proof, if we want to base adaptive security of the Chained CmPKE conforming GSD security from a CmPKE that is only IND-CCA secure (i.e., without adaptive corruption security), then we will incur an exponential reduction loss during the game transition of Game 1 to Game 2. This is because we need to guess correctly all the encryption keys that will not get corrupted from the set $[M]$ in order to simulate the corruption queries. In the worst case, we will lose a factor of $O(2^M)$.

[38] To be precise, we assume the IND-CCA security with adaptive corruption game is defined in the random oracle model. This is without loss of generality.

Chapter 5

MetaData-Hiding Continuous Group Key Agreement[1]

5.1 Introduction

Metadata in SGM. In secure group messaging (SGM) protocols, we can informally divide sensitive information into the following three layers:

$$\left\{\begin{array}{ll} \text{1st layer:} & \text{group secret keys \& messages} \\ \text{2nd layer:} & \text{static, explicit metadata} \\ \text{3rd layer:} & \text{dynamic, implicit metadata} \end{array}\right\} =: \text{“metadata”}$$

Securing the 1st layer is the *default* goal of any SGM protocol; exchanging messages in an E2EE fashion is only possible if secure group secret keys[2] are shared among the group. Since the server is not considered an endpoint of the conversation, state-of-the-art SGM protocols aim at protecting the 1st layer from the server.

The 2nd and 3rd layers together constitute the metadata. Since they help the server to ensure the functionality of the SGM protocol, they are often only encrypted using a transport layer encryption protocol (like TLS or Noise [Per18]) between the server and the participants. In this case, the server has access to this information and may expose it if legally compelled, as discussed earlier.

The 2nd layer captures any *static* metadata that is *explicitly* leaked from the content transmitted over the channel. For instance, the exchanged content may explicitly include the identity of the sender in the clear or the identity of a member being added, as in e.g. vanilla Signal [MP16a] and MLSPlaintext [Bar+22]. Static metadata are also defined as a collection of *sender information* and *handshake messages* in the MLS standard draft [Bar+22, Sec. 10.1].

The 3rd layer captures any *dynamic* metadata that is *implicitly* leaked from the access pattern between group members and the server via the communication channel. For example, in MLSCiphertext that hides up to the 2nd layer, users connect to the server using a non-anonymous protocol such as TLS. Then, since the user implicitly identifies itself to the server via the channel, the server learns the group member's identities (and how many times each member accessed) by observing the users accessing the same group identity. In particular, this happens regardless of protecting the 2nd layer metadata. To hide the user

[1]The contents of this chapter are based on the work presented at ACM CCS 2022 under the title “How to Hide MetaData in MLS-Like Secure Group Messaging: Simple, Modular, and Post-Quantum” [HKP22a]. The full version is available at the IACR Cryptology ePrint Archive. [HKP22b].

[2]There could either be a unique group secret key shared among the entire group as in MLS or multiple group secret keys, where different segments of the keys are shared among different members of the group as in Signal to perform pairwise communications.

TABLE 5.1: SGM protocols and the corresponding layers they protect. "✔✔" (resp. "✘") indicates that there is a (resp. are no) security proof. "✔" indicates that there is a proof capturing the security of the 2nd & 3rd layers but not of the 1st layer.

Layer	1st	1st & 2nd	1st, 2nd & 3rd
Signal	Vanilla Signal	⊥	Private Groups
Security proof	✘	⊥	✔ [CPZ20]
MLS	MLSPlaintext	MLSCiphertext	⊥
Security proof	✔✔ E.g., [AJM22; Alw+21a]	✘	⊥

identity on the channel, users can use anonymous protocols (e.g., Tor [DMS04; Gua]) instead. However, even if an anonymous protocol is used when a user fetches information about a group they belong to, the exact subset of accessed information may be correlated to this user's identity. This may for example be the case for SGM protocols that arrange group members in complex data structures such as trees. Specifically, hiding the metadata only at the 2nd layer while using anonymous channels is insufficient since similar information may be inferred from the 3rd layer, which incorporates all implicit leakages of dynamic metadata.

In this work, only when all three layers are secured do we say that an SGM protocol is *metadata-hiding*.[3]

Existing metadata-hiding SGM. Existing SGM protocols and the level of layers they protect are depicted in Table 5.1. Signal recently proposed a metadata-hiding SGM protocol that we call Private Groups [Sig19]. This is an extension of Sealed Sender [Sig18] — a metadata-hiding *two-user* secure messaging protocol. The main building block of Private Groups is an efficient *MAC-based keyed-verification anonymous credential* (KVAC) [CMZ14] that leverages the specific properties of tools from classical group-based cryptography, such as the ElGamal PKE and Schnorr PoK. While there is no formal security proof for Signal's vanilla SGM, recently Chase, Perrin, and Zaverucha [CPZ20] proposed a new security model to capture exactly the metadata layers (2nd & 3rd layers) and provided a partial security proof of Private Groups.

MLS [Bar+22] comes in two variants: MLSPlaintext and MLSCiphertext, each corresponding to protocols protecting the 1st and 1st & 2nd layers, respectively. The security of MLSPlaintext has been scrutinized over the past few years [BBN19b; AJM22; Alw+20a; Alw+21a; BCK22] and we now have a good understanding of it. However, no formal security proof for MLSCiphertext is known. Moreover, unlike Signal's Private Groups, constructing any variant of MLS that further hides the *dynamic* metadata is unanswered. Considering that dynamic metadata leaks part of, if not all, static metadata, MLSCiphertext may be leaking more metadata in practice than ideally expected.

5.1.1 Goal of This Work

In this work, we focus on *continuous group key agreement* (CGKA) — an abstraction that captures the core protocol underlying the MLS protocol, i.e., TreeKEM [BBR18], and many other MLS-inspired SGM protocols [Coh+18; Kle+21; Alw+20a; AJM22; Alw+21a; Has+21b; Alw+20b; Alw+22c; Alw+22d].[4] In brief, CGKA allows an evolving group of users to agree on a continuous sequence of group secret keys. Other than the simple function of sharing a group secret key, CGKA further models the strong notions of *forward secrecy* (FS) and *post-compromise security* (PCS) [CCG16; Coh+17; ACD19], which allow to greatly limit the scope of a compromise.

[3] Note that there are other types of metadata we can consider such as access timing [Mar+21] and geolocation of users.

[4] To be accurate, MLS was inspired by *asynchronous ratchet trees (ART)* [Coh+18].

FIGURE 5.1: (Left) p and c are proposals and commits. (Right) id_1 uploads a proposal p_1 to the server Sv; id_2 downloads all the stored proposals $\vec{p}$ and uploads a commit c_2 for the next epoch. The single (resp. double, triple) bordered box indicates the 1st (resp. 2nd, 3rd) layer information.

In a nutshell, a CGKA works as follows (see also Figure 5.1):[5] A group member may either (a) add a new member, (b) remove a member, and/or (c) update its keys by sending a *proposal* p. In an arbitrary interval, a group member may download the list of proposals $\vec{p} = \{ p_i \}_i$ from the server and take them into effect by transmitting a *commit* c — this creates a new epoch where the group state is updated according to $\vec{p}$. Importantly, a commit also updates the group secret key to achieve PCS.

A proposal p consists of five elements: (i) a string gid identifying the group; (ii) a counter epoch that specifies the current group state; (iii) the identity id_p of the member creating p; (iv) a string act specifying whether p corresponds to (a), (b), or (c); and (v) other information typically required for authentication. A commit c has a similar structure, where id_c denotes the committer and $\mathsf{ct}_{\mathsf{key}}$ is a ciphertext encrypting a key used to update the group secret.

As depicted in Figure 5.1, key is the 1st layer information, and any other static information included in p and c other than $(\mathsf{gid}, \mathsf{epoch})$ belong to the 2nd layer. Here, $(\mathsf{gid}, \mathsf{epoch})$ needs to be clear so that the receiver can download the appropriate p and c from the server. In the past few years, we have seen several increasingly stronger or different types (e.g., game-based, simulation-based) of security models for CGKA [Alw+20b; AJM22; Alw+20a; Alw+21a; BCK22; Kle+21; Alw+22d; Has+21b; Wei+21; Alw+22c], however these models only capture security at the 1st layer. Although it is straightforward to construct a CGKA that intuitively secures the 2nd layer once a group secret key is established, it is not clear whether this intuition is correct. Indeed, MLSPlaintext has undergone 13 iterations, and formal security analyses of the 1st layer [Alw+20a; AJM22] uncovered some subtle bugs. Thus, our first goal is the following:

(**G1**) *Propose a security model capturing the security of the 1st & 2nd layers and prove the security of existing CGKAs.*

As discussed above, securing the 2nd layer alone is insufficient. At first glance, it is tempting to replace the use of TLS for client-server communication with a client-anonymized authenticated channel (e.g., VPN or an anonymized proxy such as Tor [DMS04; Gua]) in order to hide the 3rd layer. Unfortunately, this introduces another issue since, without any authentication on the client side, any adversary who knows $(\mathsf{gid}, \mathsf{epoch})$ can upload arbitrary garbage proposals and commits to the server, causing a denial of service (DoS) against the group. It could be possible to rely on the efficient MAC-based KVAC used by Signal's Private Groups [Sig19], however, their construction is highly limited to a classical, pre-quantum setting, and the security proof is in the generic group model [Sho97]. Considering the modularity of the vanilla Signal and the MLS protocol, having a generic construction that can be efficiently instantiated from versatile assumptions, including but not limited to *post-quantum* assumptions, is highly desirable. Of independent interest, we note that in the face of a compromise or removal of a group member, Private Groups must restart a new group [CPZ20]. It remains an interesting problem to construct a protocol that offers any (non-trivial) PCS. This brings us to our second goal:

[5] We base the explanation on the most recent iterations of TreeKEM (i.e., after version 8 on MLS) following a "*propose-and-commit*" flow.

(**G2**) *Propose an efficient and generic metadata-hiding CGKA achieving the same level of FS and PCS offered by existing non-metadata-hiding CGKAs.*

Finally, Chase, Perrin, and Zaverucha [CPZ20] proposed a security model capturing the 2nd & 3rd layers of Signal. However, it does not capture the 1st layer of security, i.e., confidentiality and integrity of exchanged messages, nor the notion of PCS. Moreover, this model is tailored to the specific construction of Signal's Private Groups [Sig19] and seems unfit for CGKA. Thus, we arrive at our final goal:

(**G3**) *Propose a security model for metadata-hiding CGKA.*

5.1.2 Contribution of This Work

UC model for the 2nd layer. We propose the first security model of CGKA capturing the security of the 2nd layer, i.e., static metadata. Our security model extends the state-of-the-art universal composability (UC) security model used by Alwen et al. [AJM22] and Hashimoto et al. [Has+21b] to analyze TreeKEM version 10 in MLS and Chained CmPKE, respectively — we denote the ideal functionality as $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$.[6] The main new ingredients we introduce are *leakage functions* that allow us to formally model the leaked static metadata (e.g., sender's identity) from the proposals and commits. Similar to the state-of-the-art ideal functionalities, ours captures a strong model where *active adversaries* can tamper with or inject messages, and *malicious insiders* can invite malicious members to the group and arbitrarily fork the group state. With this formalization effort, we answer the first half of (**G1**), see Section 5.3 for details.

Chained CmPKE$^{\text{ctxt}}$ **UC-realizes** $\mathcal{F}_{\mathbf{CGKA}}^{\text{ctxt}}$. We prove that a ciphertext variant of Chained CmPKE by Hashimoto et al. [Has+21b], coined as Chained CmPKE$^{\text{ctxt}}$, UC-realizes the ideal functionality $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$.[7] Considering the similarity between Chained CmPKE and TreeKEM, we believe the ciphertext variant of TreeKEM can also be proven secure following a similar proof. The reason why we focused on the former is that it is in some sense a generalization of the latter — it allows members to *selectively download* updates from the server, also known as *filtered* CGKA [Alw+22d]. This generalization allows obtaining a concretely efficient CGKA even in the *post-quantum* regime, which otherwise could be quite inefficient [Has+21b]. This security proof addresses the second half of (**G1**), see Section 5.4 for details.

UC model for the 3rd layer. We propose the first UC security model of CGKA capturing the security of the 3rd layer, i.e., group access pattern — we denote the ideal functionality as $\mathcal{F}_{\text{CGKA}}^{\text{mh}}$. Any CGKA that UC-realizes $\mathcal{F}_{\text{CGKA}}^{\text{mh}}$ is a *metadata-hiding* CGKA. The model captures the fact that a group member performing an upload or download remains anonymous and unlinkable from the server, while also restricting non-group members from accessing the group contents. To formalize the latter property, our ideal functionality $\mathcal{F}_{\text{CGKA}}^{\text{mh}}$ captures an *honest-but-curious* server for the first time. All prior models only considered *malicious* servers so it was not possible to define a "correct" behavior of the server, i.e., shutting out non-group members. $\mathcal{F}_{\text{CGKA}}^{\text{mh}}$ allows the adversary to corrupt the server, in which case it becomes identical to $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ defined above. This answers (**G3**), see Section 5.6 for details.

A generic and efficient protocol UC-realizing $\mathcal{F}_{\mathbf{CGKA}}^{\text{mh}}$. We provide a simple and generic wrapper protocol W^{mh} that UC-realizes $\mathcal{F}_{\text{CGKA}}^{\text{mh}}$ in the $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$-hybrid model. Specifically, given an *arbitrary* CGKA Π_{ctxt} that UC-realizes $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$, W^{mh} in composition with Π_{ctxt} UC-realizes $\mathcal{F}_{\text{CGKA}}^{\text{mh}}$. Unlike Signal's Private Groups, we do not rely on complex tools such as a MAC-based KVAC, whose known efficient instantiations require classical group-based assumptions. Our key insight is to leverage the unique group secret key shared

[6] The subscript "ctxt" is inspired by the protocol name MLSCiphertext

[7] This variation mirrors what MLSCiphertext does to MLSPlaintext, in the sense that it encrypts the static metadata by applying a layer of encryption.

among the members (which is non-existing in Signal) to perform a proof of membership to the server. The concrete construction of our wrapper protocol only requires a standard signature scheme, which can be efficiently instantiated using either classical or post-quantum assumptions. Our metadata-hiding CGKA inherits all the FS and PCS properties satisfied by the underlying CGKA satisfying $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$. For instance, using MLSCiphertext as the underlying CGKA, the upload cost of a key update of our metadata-hiding CGKA can be $O(\log N)$, as opposed to $O(N)$ for Private Groups. This provides a theoretic answer to (**G2**), see Section 5.5 for details.

Instantiation and efficiency analysis. We provide concrete instantiations of the proposed protocols under either classical or post-quantum assumptions. We then study the bandwidth impact of W^{mh} when applied to Chained CmPKE$^{\text{ctxt}}$. The impact of W^{mh} is moderate, as it never increases the bandwidth cost of the principal operations ("update", "add" and "remove" proposals, as well as commit or application messages) by more than a factor of two. In practice, the concrete overhead may be even lower. This illustrates that our notion of metadata-hiding CGKA can be realized at a moderate cost. This covers the efficiency aspects of (**G2**), see Section 5.7.2 for details.

Statistical leakage from metadata-hiding CGKA. Our security model allows us to prove that a CGKA UC-realizes an ideal functionality $\mathcal{F}_{\text{CGKA}}^{\text{mh}}$ *with respect to a specific leakage function*, which defines any inherent metadata leakage that cannot be hidden. What the adversary can learn from this leakage function is another question.

We initiate a discussion on the nature and extent of the information that can be inferred from this leakage. We study several CGKAs [AJM22; Kle+21; Has+21b; Alw+22d; Alw+22c] and find that all of them leak information through the size of protocol messages (welcome, proposal and/or commit), sometimes in surprising and indirect ways. At this point, we emphasize that the authors of these protocols never claimed them to be metadata-hiding, so this leakage does not reflect the shortcomings of the designs. We believe that a systematic study of this leakage, as well as proposing countermeasures to provably mitigate it, constitutes a valuable and exciting research direction. For now, this discussion explores the limitations of (**G3**), see Section 5.8 for details.

We would like to clarify the limitations of this work. First, we only consider the CGKA aspect of SGMs. While it is believed that CGKA captures the essence of SGMs — which is supported by the vast amount of research focusing solely on CGKA [Alw+20b; AJM22; Alw+21a; Alw+20a; BCK22; Kle+21; Alw+22d; Alw+22c; Has+21b; Wei+21] — to argue a provably secure metadata-hiding SGM, we would need to extend our security model to cover the message exchanging layer as well. It was only recently that a security model that captures the entire SGM at the 1st layer was proposed [Alw+21a]. Second, our security model and protocol do not prevent an adversary from anonymously registering numerous fake groups on the server. We only prevent an outsider from accessing an existing group. Technically, this seems efficiently solvable using standard anonymous credentials [Cha82] and we leave it as future work to incorporate these into our security model. Finally, metadata outside the scope of our models, such as access timing [Mar+21] and device fingerprinting, may circumvent the privacy guarantees provided by a metadata-hiding CGKA.

5.2 Background about CGKA

We now provide some background on the existing definition of CGKA that models the default security of the 1st layer, i.e., *group secret key*.[8] In this work, we focus on CGKA defined in the UC framework. With the

[8] As mentioned in Section 5.1.2, CGKA only captures the security of the group secret key k. The message, also included in the 1st layer, is handled by a different protocol. Roughly, the messages are sent through E2EE using the established k.

nice composability property, we will be able to construct a metadata-hiding CGKA in a modular manner.

Due to page limitation, we refer the readers to Section 5.3 for some background on the UC framework and a formal definition of the ideal functionality $\mathcal{F}_{\text{CGKA}}$ capturing the UC security of the 1st layer.

5.2.1 Syntax of CGKA

All the group members internally store the group identifier gid, the current epoch of the group, the current group secret key, and information to identify the current members. Moreover, each member (roughly) stores a public key whose associated secret key is known only to themselves. This is called the *key material*.

Members can upload *proposals* p to the server, which can take on three types: addition or removal of some members, and update of key materials. Members can download any subset of these proposals $\vec{p}$ at an arbitrary interval to create a *commit* c. Any members can then *process* this commit to update their group state to the next epoch according to the content of the proposals that were committed. The group secret key is always updated after a process — if every group member's key material was uncompromised at that epoch, then this effectively *heals* the group and offers PCS.

In this work, we follow the syntax of Hashimoto et al. [Has+21b] (which extends the syntax of Alwen et al. [AJM22]) that models *selective downloading*. This allows the members to only download part of the commit required to move to the next epoch. Specifically, a commit c is divided into $(c_0, \vec{c})$, where c_0 is the member-*independent* commit and $\vec{c} = (\widehat{c}_{\mathsf{id}})_{\mathsf{id}}$ is the list of member-*dependent* commit. Member id only needs to download $(c_0, \widehat{c}_{\mathsf{id}})$ from the server to advance its epoch. Here, we assume there is a canonical ordering of the group members, and member id requests its index in the list $\vec{c}$ to the server to download $\vec{c}[\mathsf{index}] = \widehat{c}_{\mathsf{id}}$. Such selective downloading can bring great efficiency gain — especially in the post-quantum regime where asymmetric ciphertexts tend to be much larger than classical ones (see [Has+21b] for further motivations). This formalization was later coined as *filtered* CGKA [Alw+22d].

Informally, CGKA is defined by the following algorithms, where we assume id is the executing party and omit it from the input.

Group Creation $(\texttt{Create})$**:** It initializes a new group state with party id as the only member.[9]

Proposals $(\texttt{Propose}, \mathsf{act}) \to p$ **:** It outputs a proposal p for the *action* act that can take on the value 'add'-id_t, 'rem'-id_t, or 'upd'. The first two actions dictate the adding or removal of id_t. The last updates id's key material.

Commit $(\texttt{Commit}, \vec{p}) \to (c_0, \vec{c}, \vec{w})$**:** It commits a vector of proposals $\vec{p}$ and outputs a commit $(c_0, \vec{c})$. c_0 is a member-independent commit while $\vec{c} = (\widehat{c}_{\mathsf{id}'})_{\mathsf{id}'}$ is a list of member-dependent commits, where $|\vec{c}|$ is equal to the current group size. If $\vec{p}$ contains an add proposal, then it outputs a welcome message $\vec{w} = (\widehat{w}_{\mathsf{id}_t})_{\mathsf{id}_t}$, where id_t denotes the added members.[10]

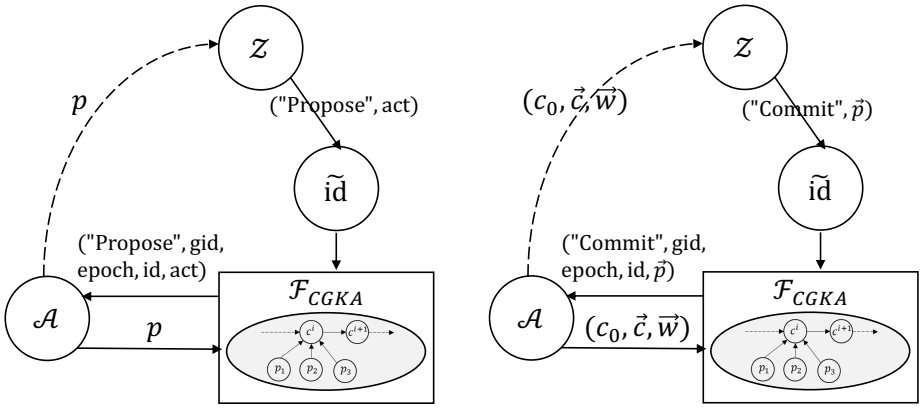
Process $(\texttt{Process}, c_0, \widehat{c}_{\mathsf{id}}, \vec{p})$**:** It processes a commit $(c_0, \widehat{c}_{\mathsf{id}})$ with the associated proposals $\vec{p}$, and advances id's internal group state to the next epoch.

Join $(\texttt{Join}, \widehat{w}_{\mathsf{id}})$**:** It allows id to join the group using the welcome message $\widehat{w}_{\mathsf{id}}$. id's group state is synced with any member who processes the commit made at the same epoch.

Key $(\texttt{Key}) \to \mathsf{k}$**:** It outputs the current group secret key k.

[9] Following prior definitions [AJM22; Alw+20b; Has+21b], we assume Group Creation is run only once.

[10] Although we can also structure $\vec{w}$ to have a party independent w_0 and dependent part, we chose not to do so since it leads to a less secure metadata-hiding protocol. Roughly, by looking at w_0, the server can infer who will be added to the same group.



(A) `Propose` (B) `Commit`

FIGURE 5.2: Group operations in the UC security model. $\mathcal{Z}$ invokes the dummy party $\widetilde{\text{id}}$, and $\mathcal{F}_{\text{CGKA}}$ informs the adversary $\mathcal{A}$ of this invocation. $\mathcal{A}$ simulates the corresponding proposal or commit and sends it to $\mathcal{F}_{\text{CGKA}}$. W.l.o.g., we assume $\mathcal{A}$ sends the same information to $\mathcal{Z}$, denoted by a dashed line. The shaded region denotes the history graph maintained by $\mathcal{F}_{\text{CGKA}}$.

5.2.2 Default UC Security Model of CGKA

All prior works on CGKA capture the rough security notion that the group secret key should remain hidden from the adversary. We follow the state-of-the-art UC security model of [AJM22; Has+21b]. In this model, the malicious server is modeled as an *active adversary* that can tamper with or inject messages. Moreover, a rogue group member is modeled as a *malicious insider* that can invite other malicious members (e.g., server) to the group and arbitrarily fork the group state. The UC security model captures the fact that even in face of such strong adversaries, the group secret key remains indistinguishable from random under certain conditions. Below, we explain the high-level description of the ideal functionality $\mathcal{F}_{\text{CGKA}}$.

History graphs and the safe predicate. The core concept underlying the definition is the so-called *history graph* [AJM22; Alw+21a] maintained by $\mathcal{F}_{\text{CGKA}}$. A history graph is a symbolic representation of the group's evolution, where each node on a graph roughly corresponds to a group state at a particular epoch.[11] It tracks all the generated commits and proposals, and group members' positions on the history graph. To define the security of the group secret key, $\mathcal{F}_{\text{CGKA}}$ is parameterized by a predicate **safe**, which takes the history graph and a node as input, and assigns a random group secret key $\mathsf{k} \leftarrow \mathcal{K}$ to a node where **safe** is true. This formalization allows modeling FS and PCS naturally. For example, an adversary $\mathcal{A}$ can corrupt a member at some epoch, thus making **safe** false at epoch. If the member is *healed* at a later $\text{epoch}' > \text{epoch}$, then **safe** becomes true again at epoch'. Importantly, **safe** is a scheme-specific predicate that can be defined arbitrarily to capture different levels of FS and PCS.

Example: Ideal propose and commit. We depict the ideal `Propose` and `Commit` functions in Figure 5.2, where $\mathcal{Z}$ denotes the environment. For example, $\mathcal{Z}$ can invoke the (dummy) party $\widetilde{\text{id}}$ to execute a commit on an arbitrary list of proposals $\vec{p}$. $\mathcal{F}_{\text{CGKA}}$ then provides to $\mathcal{A}$ the proposals $\vec{p}$ along with all the static metadata $(\mathsf{gid}, \mathsf{epoch}, \mathsf{id})$ included in a commit, where id denotes the identity of the committer. $\mathcal{A}$ interprets the proposals $\vec{p}$ and simulates the commit $(c_0, \vec{c})$, where it further simulates the welcome message $\vec{w}$ if $\vec{p}$ includes an add proposal. The commit and welcome messages are sent to $\mathcal{F}_{\text{CGKA}}$ and are registered in the

[11] The formal definition is made with more care since in case an adversary forks the group state, two different nodes with the same epoch can be created.

history graph as a new node indicating that a new epoch has been created. Here, if $\vec{p}$ was generated by a group member, then $\mathcal{F}_{\text{CGKA}}$ mandates correctness by checking if the commit output by $\mathcal{A}$ was consistent with the actions included in $\vec{p}$. Otherwise, in case some $p \in \vec{p}$ was not generated by a group member, then this implies that the server or some rogue insider injected a *malicious* proposal.

5.3 Static Metadata-Hiding CGKA: Define UC Security Model

As a first step, we propose a UC security model capturing the security of the 1st & 2nd layers (i.e., group secret keys and *static* metadata) by defining a new ideal functionality $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$. This is an extension of the ideal functionality $\mathcal{F}_{\text{CGKA}}$ [Alw+20b; AJM22; Has+21b] that captures the security of the 1st layer. Similarly to the **safe** predicate used in $\mathcal{F}_{\text{CGKA}}$ to control the levels of FS and PCS, $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ comes with five *leakage functions* allowing to control the amount of static metadata leaked from create group, proposal, commit, process, and join. By defining the leakage functions to leak all the static metadata, then $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ (essentially) recovers the prior ideal functionality $\mathcal{F}_{\text{CGKA}}$.

5.3.1 Difference between New Model and Previous Model

$\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ has the same user interface (or syntax) as $\mathcal{F}_{\text{CGKA}}$. The main difference is how the internals of the ideal functionalities are defined. Below, we explain the three main points at which $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ differs from $\mathcal{F}_{\text{CGKA}}$. The full details on $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ is provided in Section 5.3.

Modeling static metadata leakage. Recall how $\mathcal{F}_{\text{CGKA}}$ defined the ideal proposal function (see Figure 5.2). When a party id is invoked on $(\texttt{Propose}, \mathsf{act})$ from the environment $\mathcal{Z}$, $\mathcal{F}_{\text{CGKA}}$ informs the adversary $\mathcal{A}$ with $(\texttt{Propose}, \mathsf{gid}, \mathsf{epoch}, \mathsf{id}, \mathsf{act})$. $\mathcal{A}$ then simulates a proposal p. This models the fact that p in the real world is allowed to leak information on $(\mathsf{gid}, \mathsf{epoch})$, the party id creating p, and the type of proposal $\mathsf{act} \in \{\text{`add'-}\mathsf{id}_t, \text{`rem'-}\mathsf{id}_t, \text{`upd'}\}$ included in p.

We control the amount of such leakage from p by a leakage function ***leak-prop**. Informally, ***leak-prop** takes as input the identity id of the member creating the proposal and the current epoch. In case **safe** is false at epoch, ***leak-prop** outputs all the static metadata since the group secret key is compromised at that epoch. Otherwise, it only outputs the static metadata the CGKA is allowed to leak. For example, to model MLSCiphertext, we define ***leak-prop**$(\mathsf{id}, \mathsf{epoch})$ to only output $(\mathsf{gid}, \mathsf{epoch}, |\mathsf{id}|, |\mathsf{act}|)$ when **safe** is true, where $|\mathsf{act}|$ leaks the size of the action included in p. In case every member identity id_t is encoded in the same bit-length, then this implies for instance that p does not leak who created the p and who was added and removed.

Similarly to ***leak-prop**, we define four more leakage functions ***leak-create**, ***leak-proc**, ***leak-com**, and ***leak-wel**. For instance, the last two controls the amount of static metadata leaked from commits and welcome messages, respectively. Unlike proposals, the static metadata that is inherently leaked from commits and welcome messages differs between CGKAs. Thus, care is required when formally defining them. For instance, in MLSCiphertext, a welcome message for member id_t includes a hash of id_t's key package $\mathsf{kp}_{\mathsf{id}_t}$ (i.e., key materials used by id_t). To model this fact, ***leak-wel** must also output $\mathsf{kp}_{\mathsf{id}_t}$ to the adversary (see Section 5.4.3 for more discussion). Another subtlety is that the size of a commit in MLSCiphertext is dictated by how many blank nodes exist in the tree. In Section 5.8, we discuss the real-world consequences of these inherent leakages of static metadata.

Using semantics for nodes in history graphs. In this work, we update the prior definition of history graphs to use the *semantics* of a transcript to identify the nodes in a history graph. That is, we identify a node by a counter that informally counts the number of group operations leading to the node. In previous

works [AJM22; Has+21b] (which did not capture the security of the static metadata), each node was identified by the *value* of the *non-encrypted* member-independent commit c_0. Since c_0 uniquely defined an epoch and group state, it made intuitive sense to identify each node by c_0.

Unfortunately, this intuition is lost when we try to further secure the static metadata. This is because c_0 is now an *encryption* of the actual commit content. Namely, there can be two distinct c_0 and c_0' that decrypt to the same commit content. While we would like to assign different members processing c_0 and c_0' to the same node in the history graph, it is not clear which c_0 and c_0' to use to identify the node. Such an issue disappears by using the semantics since the group operation defined inside c_0 and c_0' are identical.

Independently, Alwen et al. [Alw+22d] also uses the semantics to define nodes in their security model. This was crucial to capture a more advanced form of (non-metadata-hiding) CGKA coined as *server-aided* CGKA.

Restricted adversary due to commitment problem. In the UC framework, it is typical to restrict the adversary $\mathcal{A}$ from performing corruptions that would cause the so-called *commitment problem*. Informally, this is a type of attack where $\mathcal{A}$ can adaptively choose to corrupt some states *after* being provided with some challenges with respect to the state. While this attack is prohibited by default in any natural game-based definition (e.g., the adversary cannot obtain a secret key after being provided the challenge ciphertext), we need to make this restriction explicit in the UC-based definition.

Compared to prior works in the UC framework [Alw+20b; AJM22; Has+21b], the description of the restriction we require is more strict. Previously, when a new node was created due to a commit, the predicate **safe** was undefined for that node. Roughly, this is because the group secret key inside that node was never explicitly used during the real protocol and $\mathcal{A}$ was given the freedom to adaptively decide whether to corrupt that node at an arbitrary moment of the security game. However, in the static metadata-hiding setting, this freedom of the adversary needs to be restricted. This is because when a new node is created, the group secret key inside this node is explicitly used to encrypt the static metadata of the commit content. When **safe** is true at this node, the ideal functionality $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ (roughly) wants to assign a random ciphertext to model the fact that the commit does not leak any information. Since $\mathcal{A}$ is explicitly given the ciphertext, it cannot later decide to corrupt the node (i.e., **safe** remains true), as otherwise, it could trivially distinguish a valid ciphertext from a random one.

5.3.2 Background

Universal Composable Security. We briefly recall the UC framework. We refer to [Can01; Can+07] for the full descriptions. The UC security is formalized by the indistinguishability of real and ideal protocols. In the real protocol, parties execute a protocol Π, where an *adversary* $\mathcal{A}$ may corrupt some of the parties. In the ideal protocol, the parties are replaced by *dummy* parties that interact with an *ideal functionality* $\mathcal{F}$, where a *simulator* $\mathcal{S}$ may corrupt some of the dummy parties. The dummy parties are defined to be the identity function that simply outputs whatever is fed as input. In addition, there is another entity called the *environment* $\mathcal{Z}$ that tries to distinguish the two protocols. In the real (resp. ideal) protocol, $\mathcal{Z}$ can interact arbitrary with $\mathcal{A}$ (resp. $\mathcal{S}$), and it can also invoke any non-corrupted parties (resp. dummy parties) to honestly run the protocol Π (resp. the ideal functionality $\mathcal{F}$), where the output is always reported back to $\mathcal{Z}$. The goal of UC-security is then, given any adversary $\mathcal{A}$, to construct a simulator $\mathcal{S}$ such that any environment $\mathcal{Z}$ cannot distinguish between the real and ideal protocols. We say the real protocol Π *UC-realizes* the ideal functionality $\mathcal{F}$ if such $\mathcal{S}$ can be constructed. Put differently, whatever $\mathcal{A}$ can learn from the real protocol Π can be simulated using the information provided by the ideal functionality $\mathcal{F}$, which is secure by definition.

We often construct a protocol in a setting where a copy of an ideal functionality $\mathcal{G}$ is available. We call such a model $\mathcal{G}$-hybrid model. If a real protocol Π UC-realizes $\mathcal{F}$ while providing access to an ideal functionality $\mathcal{G}$, then we say Π UC-realizes $\mathcal{F}$ in the *$\mathcal{G}$-hybrid model*. For a real protocol Π in the $\mathcal{G}$-hybrid model, and a real protocol Π' that realizes $\mathcal{G}$ (in the standard model), we can naturally define a composed protocol $\Pi^{\Pi'}$ in the standard model, in which calls for $\mathcal{G}$ from Π are answered by Π' instead of $\mathcal{G}$. Canetti [Can01] proved a *universal composition theorem* stating that, if Π UC-realizes $\mathcal{F}$ in the $\mathcal{G}$-hybrid model and Π' UC-realizes $\mathcal{G}$, then $\Pi^{\Pi'}$ UC realizes $\mathcal{F}$.

The Corruption Model. We define what we mean by "corrupting" a party. In this work, we use the corruption model of continuous state leakage (transient passive corruptions) and adversarially chosen randomness of [Alw+20b]. This is a standard in CGKA literature, but this is non-standard for typically UC-security. This corruption model allows the adversary to repeatedly corrupt parties by sending them two types of corruption messages: (1) a message `Expose` causes the party to send its current state to the adversary (once), (2) a message $(\texttt{CorrRand}, b)$ sets the party's rand-corrupted flag to b. If b is set, the party's randomness-sampling algorithm is replaced by the adversary providing the coins instead. Ideal functionalities are activated upon corruptions and can adjust their behavior accordingly.

Restricted Environments and Adversaries. To avoid the so-called commitment problem, caused by adaptive corruptions in simulation-based frameworks, we restrict the environment (and thus the adversary) not to corrupt parties at certain times. This roughly corresponds to ruling out "trivial attacks" in game-based definitions, e.g., the adversary cannot compromise the secret key after being provided with the challenge ciphertext. In simulation-based frameworks, such attacks are no longer trivial, but security against them requires relatively strong and inefficient cryptographic tools, e.g., non-committing encryption, and is not achieved by most protocols. We follow prior works [JMM19a; Alw+20b; AJM22; Has+21b] and consider a weakened variant of UC-security that only quantifies over a restricted set of so-called admissible environments that do not exhibit the commitment problem. Whether an environment is admissible or not is defined by the ideal functionality $\mathcal{F}$ with statements of the form **restrict** *cond* and an environment is called admissible (for $\mathcal{F}$), if it has negligible probability of violating any such *cond* when interacting with $\mathcal{F}$.

PKI functionality. As in [AJM22; Has+21b], we define our CGKA in a hybrid model where parties can access an ideal functionality that models an (untrusted) PKI. In the real protocol, the parties can interact with the Authentication Service (AS) and Key Service (KS) PKI functionalities. For instance, the environment can instruct the AS (via the party's protocol) to register a new key for a party. As a result, the AS generates a new key pair for the party and hands the public key to the environment, making the secret key available to the party's protocol upon request. We note that the adversary can register arbitrary signature keys for any party to capture an insider adversary.

Authentication Service (AS). The authentication service (AS) certifies the ownership of a signature key. The AS is formalized by the functionality $\mathcal{F}_{\text{AS}}$ defined in Figure 5.3. The definition is identical to that used in [Has+21c]. $\mathcal{F}_{\text{AS}}$ allows parties to register fresh signature key pairs via `register-svk` query and to check whether a verification key svk is registered by a party id' via the `verify-cert` query. On registration, the new key pair for a party id is generated by $\mathcal{F}_{\text{AS}}$ using a genSSK algorithm (whose concrete specification depends on the CGKA). If id's current randomness source is corrupted (i.e., $\mathsf{Rand}[\mathsf{id}] = \text{'bad'}$), $\mathcal{F}_{\text{AS}}$ asks the adversary to provide the randomness. After registration, id receives the new verification key svk. Also, parties can retrieve their signing keys via `get-ssk` query and delete registered signing keys via `del-ssk` query. The adversary can register arbitrary verification keys in the name of any party. When a party is corrupted, all signing keys except for the deleted ones are leaked to the adversary. Security is modeled by the ideal-world variant of $\mathcal{F}_{\text{AS}}$, called $\mathcal{F}_{\text{AS}}^{\text{IW}}$. It marks leaked signing keys by storing them in the $\mathsf{ExposedSvk}$

array (see boxes in Figure 5.3). $\mathcal{F}_{\text{AS}}$ allows the Key Service functionality $\mathcal{F}_{\text{KS}}$ (see below) to signal that a certain ssk is leaked. $\mathcal{F}_{\text{KS}}$ sends this signal when the signature key is leaked due to a compromise of a key package. Finally, $\mathcal{F}_{\text{AS}}^{\text{IW}}$ always leaks all registered signing keys to the simulator.

Key Service (KS). The Key Service (KS) allows parties to upload one-time key packages used to add them to groups while they are offline. The KS is formalized by the functionality $\mathcal{F}_{\text{KS}}$ defined in Figure 5.4. The functionality is identical to that used in [Has+21c] except that KS checks the validity of maliciously registered key packages and parties can check whether the key package is registered to KS. In our definition, when the adversary registers a key package to $\mathcal{F}_{\text{KS}}$, $\mathcal{F}_{\text{KS}}$ checks whether the registering key packages are valid by the `*validate-kp` function. Thus, $\mathcal{F}_{\text{KS}}$ ensures the registered key packages are valid in the sense that the `*validate-kp` function returns `true`. In addition, parties also check the key package is registered to $\mathcal{F}_{\text{KS}}$ via a `has-kp` query. This allows parties to check the validity of a key package via the `has-kp` query since $\mathcal{F}_{\text{KS}}$ ensures the validity of registered key packages. We introduce these functions to make the syntax of an add and update proposal to look more similar. When a party id adds a party id_t, it first fetches an id_t's key package from KS and invokes a CGKA protocol (or $\mathcal{F}_{\text{CGKA}}$) on input $(\texttt{Propose}, \text{`add'-}\mathsf{id}_t\text{-}\mathsf{kp}_t)$. The party (and $\mathcal{F}_{\text{CGKA}}$) can check the validity of $\mathsf{kp}_t$ through the `has-kp` query to $\mathcal{F}_{\text{KS}}$. This is syntactically similar to an update proposal where, the updated signing key svk is validated via the `verify-cert` query.

Other functionalities are identical to that used in [Has+21c]. Similar to $\mathcal{F}_{\text{AS}}$, parties can register key packages via the `register-kp` query. Upon receiving the `register-kp` query, $\mathcal{F}_{\text{AS}}$ generates a new key package using a $\mathsf{genKP}(\mathsf{id}, \mathsf{svk}, \mathsf{ssk})$ algorithm (whose concrete specification depends on the CGKA), which takes a party's identity id and a signature key pair $(\mathsf{svk}, \mathsf{ssk})$ and outputs a key package and the corresponding decryption key. Parties can request another party's key package via `get-kp` query. The returned key package is specified by the adversary. This reflects that the adversary can maliciously inject key packages that were not registered by honest parties. Finally, the ideal-world KS functionality $\mathcal{F}_{\text{KS}}^{\text{IW}}$ always leaks all decryption keys to the simulator.

History Graph. We use a so-called *history graph* [Alw+20b; AJM22; Has+21b; Alw+21a] to define the ideal functionality $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$.

Overview. A history graph is a labeled directed graph that acts as a symbolic representation of a group's evolution. It has two types of nodes: commit and proposal nodes, representing all executed commit and propose operations, respectively. Each party is uniquely assigned to a commit node indicating that a party is in a group of members that processed the commit assigned to that specific commit node. The nodes' labels, furthermore, keep track of all the additional information relevant for defining security. For instance, proposal nodes have a label that stores the proposed action, and commit nodes to have labels that store the epoch's application secret and the set of parties corrupted in the given epoch. Security of the application secrets is then formalized by the functionality of choosing a random and independent key for each commit node whenever security is guaranteed; otherwise, the simulator gets to choose the key. Whether security is guaranteed in a given node, is determined via an explicit safe predicate on the node and the history graph. In addition to the secrecy of the keys, the functionality also formalizes authenticity by appropriately disallowing injections.

Formal Definition. As explained in Section 5.3.1, we deviate from the definition of prior history graphs used to define CGKA in the UC framework. Each node in the history graph is identified by node pointers: $\mathsf{prop\text{-}id} \in \mathbb{N}$ for proposal nodes and $\mathsf{node\text{-}id} \in \{0\} \cup \mathbb{N}$ for commit nodes. In contrast, prior works [Alw+21a; Alw+22d] used concrete (non-encrypted) proposals and commits to identify each node. This formalization was well-defined in prior works since each proposal and commit identified a unique group operation in the real protocol. However, when considering *static metadata-hiding*, two distinct (encrypted) proposals or commits may encrypt the same group operation, in which case, we would like

The functionality is parameterized by a key generation algorithm genSSK().

Initialization

1 : RegisteredSvk $\leftarrow \emptyset$; $\boxed{\text{ExposedSvk} \leftarrow \emptyset}$
2 : SSK$[*, *] \leftarrow \perp$
3 : Rand$[*] \leftarrow$ 'good'

Inputs from a party id

Input (`register-svk`)

1 : **if** Rand[id] = 'good' **then**
2 : (svk, ssk) $\leftarrow$ genSSK()
3 : **else**
4 : Send (`rnd`, id) to the adversary and receive r
5 : (svk, ssk) $\leftarrow$ genSSK(r)
6 : $\boxed{\text{ExposedSvk} \mathrel{+}\!\leftarrow \text{svk}}$
7 : RegisteredSvk $\mathrel{+}\!\leftarrow$ (id, svk)
8 : SSK[id, svk] $\leftarrow$ ssk
9 : Send (`register-svk`, id, svk, $\boxed{\text{ssk}}$) to the adversary
10 : Send svk to the party id

Input (`get-ssk`, svk)

1 : Send SSK[id, svk] to id

Input (`del-ssk`, svk)

1 : SSK[id, svk] $\leftarrow \perp$

Input (`verify-cert`, id′, svk)

1 : Send (id′, svk) $\in$ RegisteredSvk to id

Inputs from the adversary

Input (`register-svk`, id, svk)

1 : $\boxed{\textbf{if } (*, \text{svk}) \notin \text{RegisteredSvk } \textbf{then}}$
2 : $\boxed{\text{ExposedSvk} \mathrel{+}\!\leftarrow \text{svk}}$
3 : RegisteredSvk $\mathrel{+}\!\leftarrow$ (id, svk)

Input (`expose-ssk`, id)

1 : $\boxed{\text{ExposedSvk} \mathrel{+}\!\leftarrow \{\, \text{svk} \mid \text{SSK[id, svk]} \neq \perp \,\}}$
2 : Send SSK$[\text{id}, *]$ to the adversary

Input (`CorrRand`, id, b), $b \in \{$ 'good', 'bad' $\}$

1 : Rand[id] $\leftarrow b$

Inputs from $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ and $\mathcal{F}_{\text{KS}}$

Input (`exposed`, id, svk)

1 : $\boxed{\text{ExposedSvk} \mathrel{+}\!\leftarrow \text{svk}}$
2 : Send SSK[id, svk] to the adversary

Inputs from $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ and $\mathcal{F}_{\text{KS}}$

Input (`has-ssk`, id, svk)

1 : Send SSK[id, svk] $\neq \perp$ to $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$

FIGURE 5.3: The ideal authentication service functionality $\mathcal{F}_{\text{AS}}$ and its variant $\boxed{\mathcal{F}_{\text{AS}}^{\text{IW}}}$ used during the security proof.

The functionality is parameterized by a key-package generation algorithm $\mathsf{genKP}(\mathsf{id}, \mathsf{svk}, \mathsf{ssk})$ and a key package validation function $\texttt{*validate-kp}(\mathsf{kp}, \mathsf{id})$.

Initialization

1 : $\mathsf{RegisteredKp} \leftarrow \emptyset$
2 : $\mathsf{DK}[*,*] \leftarrow \bot; \mathsf{SVK}[*,*] \leftarrow \bot$
3 : $\mathsf{Rand}[*] \leftarrow$ 'good'

Inputs from a party id

Input $(\texttt{register-kp}, \mathsf{svk}, \mathsf{ssk})$

1 : **if** $\mathsf{Rand}[\mathsf{id}] =$ 'good' **then**
2 : $(\mathsf{kp}, \mathsf{dk}) \leftarrow \mathsf{genKP}(\mathsf{id}, \mathsf{svk}, \mathsf{ssk})$
3 : **if** $\mathsf{kp} = \bot$ **then return**
4 : **else**
5 : Send $(\texttt{rnd}, \mathsf{id})$ to the adversary and receive r
6 : $(\mathsf{kp}, \mathsf{dk}) \leftarrow \mathsf{genKP}(\mathsf{id}, \mathsf{svk}, \mathsf{ssk}; \mathsf{r})$
7 : **if** $\mathsf{kp} = \bot$ **then return**
8 : Send $(\texttt{exposed}, \mathsf{id}, \mathsf{svk})$ to $\mathcal{F}_{\mathrm{AS}}$
9 : $\mathsf{RegisteredKp} \mathrel{+}\!\leftarrow (\mathsf{id}, \mathsf{kp})$
10 : $\mathsf{DK}[\mathsf{id}, \mathsf{kp}] \leftarrow \mathsf{dk}; \mathsf{SVK}[\mathsf{id}, \mathsf{kp}] \leftarrow \mathsf{svk}$
11 : Send $(\texttt{register-kp}, \mathsf{id}, \mathsf{svk}, \mathsf{kp}, \boxed{\mathsf{dk}})$ to the adversary
12 : Send kp to the party id

Input $(\texttt{get-dks})$

1 : Send $\{ (\mathsf{kp}, \mathsf{DK}[\mathsf{id}, \mathsf{kp}]) \mid \mathsf{DK}[\mathsf{id}, \mathsf{kp}] \neq \bot \}$ to id

Input $(\texttt{get-kp}, \mathsf{id}')$

1 : Send $(\texttt{get-kp}, \mathsf{id}, \mathsf{id}')$ to the adversary and receive kp'
2 : **try** $\texttt{*validate-kp}(\mathsf{kp}, \mathsf{id})$
3 : $\mathsf{RegisteredKp} \mathrel{+}\!\leftarrow (\mathsf{id}', \mathsf{kp}')$
4 : Send kp' to id

Input $(\texttt{del-kp}, \mathsf{kp})$

1 : $\mathsf{DK}[\mathsf{id}, \mathsf{kp}], \mathsf{SVK}[\mathsf{id}, \mathsf{kp}] \leftarrow \bot$

Inputs from id **and** $\mathcal{F}_{\mathbf{CGKA}}^{\mathrm{ctxt}}$

Input $(\texttt{has-kp}, \mathsf{id}, \mathsf{kp})$

1 : Send $(\mathsf{id}, \mathsf{kp}) \in \mathsf{RegisteredKp}$

Inputs from the adversary

Input $(\texttt{CorrRand}, \mathsf{id}, b)$, $b \in \{$ 'good', 'bad' $\}$

1 : $\mathsf{Rand}[\mathsf{id}] \leftarrow b$

Inputs from the adversary and $\mathcal{F}_{\mathbf{CGKA}}^{\mathrm{ctxt}}$

Input $(\texttt{exposed}, \mathsf{id})$

1 : Send $\mathsf{DK}[\mathsf{id}, *]$ to the adversary
2 : **foreach** $\mathsf{svk} \in \mathsf{SVK}[\mathsf{id}, *]$ **s.t.** $\mathsf{svk} \neq \bot$ **do**
3 : Send $(\texttt{exposed}, \mathsf{id}, \mathsf{svk})$ to $\mathcal{F}_{\mathrm{AS}}$

FIGURE 5.4: The ideal key service functionality $\mathcal{F}_{\mathrm{KS}}$ and its variant $\boxed{\mathcal{F}_{\mathrm{KS}}^{\mathrm{IW}}}$ used during the security proof.

to assign these distinct proposals and commits to the same node. Otherwise, two parties can be in the same group in the real protocol, while they are included in a different commit node in the history graph. Roughly, pointers prop-id and node-id define the *semantics* of a group operation and allow us to assign semantically equivalent proposals and commits to the same node. The pointer values of a proposal or a commit will be arbitrarily assigned by the simulator, and the ideal functionality checks whether the created history graph maintains consistency, authenticity, and confidentiality.

All nodes in the history graph store the following values:

- orig: the identity of the party who created the node, i.e., the message sender.
- par: the parent commit node, representing the sender's current epoch.
- stat $\in \{$ 'good', 'bad', 'adv' $\}$: the flag indicating whether the secrets corresponding to the node are known to the adversary. 'good' means this node is secure, 'bad' means this node is created with adversarial randomness (hence it is well-formed but the adversary knows the secret), and 'adv' means this node is created by the injected message from the adversary.

Proposal nodes further store the following values:

- act $\in \{$ 'upd'-kp, 'add'-id$_t$-kp$_t$, 'rem'-id$_t$ $\}$: the proposal action. 'upd'-kp means the corresponding party updates its key package to kp. 'add'-id$_t$-kp$_t$ means id$_t$ is added with the key package kp$_t$[12].

Commit nodes further store the following values. In this work, history graphs keep gid, epoch, and new variable conthide in addition to the values used in the previous work [AJM22; Has+21c]:

- gid: the group identifier.
- epoch: the current epoch number.
- prop: the ordered list of committed proposals.
- mem: the list of a pair of group member's identity and its key package, which is sorted by dictionary order in identities.
- vcom: the list of party-dependent commitments associated with this node.
- key: the group (application) secret.
- exp: the set keeping track of corrupted parties in this node.
- chall: the flag indicating whether the group secret is challenged. That is, chall $=$ `true` if a random group key was generated for this node, and `false` if the key was set by the adversary (or not generated).
- conthide: the flag indicating whether the static metadata protection is assured at this epoch. That is, conthide $=$ `true` means the messages issued with this epoch's group secret hide metadata, and `false` means the metadata is leaked. This value is initialized when one of proposal/commit/welcome messages is first created at this epoch.

For convenience, we define the following helper function.

- indexOf(id): returns the index of id in the list mem.

[12]The previous models only kept signature keys in kp. To capture metadata-hiding property, we need to manage which key packages are being added/updated.

5.3.3 UC Security Model $\mathcal{F}_{\mathbf{CGKA}}^{\mathsf{ctxt}}$

We propose a new ideal functionality $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ capturing the static metadata-hiding property of CGKAs. $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ is based on the prior ideal functionality $\mathcal{F}_{\mathrm{CGKA}}$ [Has+21b; Has+21c] that only captured the security of group secret keys. The ideal functionality $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ is formally defined in Figures 5.5 to 5.10, along with several helper functions in Figures 5.12 and 5.14 to 5.17 to aid the readability. By setting the flag $\mathtt{flag}_{\mathsf{contHide}}$ to false and $\mathtt{flag}_{\mathsf{selDL}}$ to false (resp. true) in the "Initialization," $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ becomes identical to $\mathcal{F}_{\mathrm{CGKA}}$ used in [Has+21c] (resp. [Has+21b] capturing *selective downloading*).

To specify the ideal functionality $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$, we also need to define the following:

***Safety Predicates*: safe**, **sig-inj-allowed**, and **mac-inj-allowed** specify which epoch secrets are secure and when authenticity is guaranteed,

***Leakage Functions*: *leak-create**, ***leak-prop**, ***leak-com**, ***leak-wel**, and ***leak-proc** specify information leaked from protocol messages.

The safety predicates and leakage functions are protocol specific. For instance, some CGKA may leak the type of proposal, while others may not. Put differently, a specific CGKA UC-realizes the ideal functionality $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ *with respect to* a particular choice of safety predicates and leakage functions. By tuning the choice, $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ allows modeling a wide variety of CGKAs. A concrete choice of such safety predicates and leakage functions is provided in Section 5.4.3, where we prove UC-security of our CGKA $\mathsf{Chained\ CmPKE}^{\mathsf{ctxt}}$.

Below, we provide an overview of the ideal functionality $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$.

States. $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ maintains the history graph. As explained in Section 5.3.2, it identifies proposal nodes by a pointer $\mathsf{prop\text{-}id} \in \{0\} \cup \mathbb{N}$ and commit nodes by a pointer $\mathsf{node\text{-}id} \in \{0\} \cup \mathbb{N}$. We assume one group is created by an honest party (see `Create` in Figure 5.5). This creates a root (commit) node called the *main root* identified by the pointer $\mathsf{node\text{-}id} = 0$. We call the group starting from the main root as the *main group*. Moreover, other roots may be created without a commit message (e.g., when a party processes an injected welcome message that is not directly related to the main group). Such roots are called *detached root*. $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ also stores a pointer $\mathsf{Ptr}[\mathsf{id}]$ for each party id. $\mathsf{Ptr}[\mathsf{id}]$ identifies id's current commit node (i.e., current epoch). If id is not in the group, $\mathsf{Ptr}[\mathsf{id}] = \bot$. $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ switches its functionality according to the flag $\mathtt{flag}_{\mathsf{selDL}}$and $\mathtt{flag}_{\mathsf{contHide}}$: $\mathtt{flag}_{\mathsf{selDL}}$is set to `true`if it performs selective downloading, and $\mathtt{flag}_{\mathsf{contHide}}$is set to `true`if it offers static metadata-hiding. To keep track of party-specific secrets (e.g., CmPKE decryption key), $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ manages the PendDK array (which stores pending secrets of the key package kp) and the CurrDK array (which stores the current secrets of id).

Interface. $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ offers interfaces to create a group, create a proposal, commit to a list of proposals, process a commit, join a group, and retrieve the group secret key. All interfaces except create and join are for group members only (i.e., parties for which $\mathsf{Ptr}[\mathsf{id}] \neq \bot$). We explain each interface in more detail below.

Group creation (See Figure 5.5) $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ allows one main group to be created by a designated party $\mathsf{id}_{\mathsf{creator}}$. Initially, the main group has a single party $\mathsf{id}_{\mathsf{creator}}$, and it can invite additional members by issuing add proposals and committing to them. $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ checks the validity of $\mathsf{id}_{\mathsf{creator}}$'s signature key by `*valid-svk`. Then, $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ informs the adversary $\mathcal{S}$[13] of the creation of a new group by sending the message $(\mathtt{Create})$ to $\mathcal{S}$. (This models the fact that a server knows when a group is created.) If $\mathtt{flag}_{\mathsf{contHide}} = \mathtt{false}$, the adversary also receives the identity and signature key of the group creator. Otherwise, the adversary receives ***leak-create**$(\mathsf{id}, \mathsf{svk})$. The adversary returns the new group's identity gid. Then, $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ generates the initial key package by the `*update-kp` function. The `*update-kp` function generates a new key package

[13] In the UC framework, it is conventional to call $\mathcal{S}$ appearing in the ideal functionality as the "adversary." We use the term "simulator" during the security proof.

by itself if both $\texttt{flag}_{\mathsf{contHide}}$and **safe** are true; otherwise asks the key package to the adversary $\mathcal{S}$. This models, in the ideal metadata-hiding CGKA protocol, an honest party generates a new key package, but it is hidden from the adversary.[14] Then, $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ initializes the root node. Note that the epoch of an initial group is set to 0.

Creating proposals (See Figure 5.6) A party id can be invoked by the environment $\mathcal{Z}$ to create a proposal with a specific action act. $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ informs the adversary $\mathcal{S}$ that a proposal message is being created. $\mathcal{S}$ receives ***leak-prop**$(\mathsf{id}, \mathsf{act})$ and returns a flag *ack*, an ideal proposal p and a node pointer prop-id. If $\texttt{flag}_{\mathsf{contHide}} = \texttt{false}$, then $\mathcal{S}$ obtains all the information $(\mathsf{Ptr}[\mathsf{id}], \mathsf{id}, \mathsf{act}) =$ ***leak-prop**$(\mathsf{id}, \mathsf{act})$ that can be inferred from a non-encrypted proposal. $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ allows $\mathcal{S}$ to send $ack = \texttt{false}$ to report that the protocol fails. If $\mathsf{act} =$ 'upd', $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ updates act with the new updated key package returned from `*update-kp` function. The `*update-kp` function generates a new key package by itself if both $\texttt{flag}_{\mathsf{contHide}}$and **safe** are true; otherwise asks the key package to the adversary $\mathcal{S}$. This models, in the ideal metadata-hiding CGKA protocol, an honest party generates a new key package, but the proposal message hides the key package from the adversary.[15] If the protocol succeeds, and if no node associated with p exists, $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ creates a new proposal node $\mathsf{Prop}[\mathsf{propCtr}]$ and assigns propCtr to p (setting $\mathsf{PropID}[p] \leftarrow \mathsf{propCtr}$). In certain situations, $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ may not create a new proposal node. For example, id proposes to remove the same party twice in the same epoch. Another example is when the adversary $\mathcal{S}$ controls the party's randomness (via setting $\mathsf{Rand} =$ 'bad') and the party proposes to update using the same randomness twice. In these cases, $\mathcal{S}$ can specify to attach the created proposal p to an existing proposal node prop-id. $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ then enforces that the states on the existing proposal node are consistent with the expected one using `*consistent-prop`. $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ marks whether the current epoch is secure or not using `*mark-content-hiden-epoch`. This information is used to determine epochs the adversary is allowed to corrupt (see the **restrict** check run within `Expose` in Figure 5.11). Finally, if all check passes, $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ returns the proposal message p to the calling (dummy) party id, which simply relays it to the environment $\mathcal{Z}$.

Committing to proposals (See Figure 5.7) A party id can be invoked by the environment $\mathcal{Z}$ to create a commit with a list of proposals $\vec{p}$, along with a (possibly fresh) signature verification key svk. $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ informs the adversary $\mathcal{S}$ that a commit message is sent and provides ***leak-com**$(\mathsf{id}, \vec{p}, \mathsf{svk})$. If $\texttt{flag}_{\mathsf{contHide}} = \texttt{false}$, then $\mathcal{S}$ obtains all the information $(\mathsf{Ptr}[\mathsf{id}], \mathsf{id}, \vec{p}, \mathsf{svk}, \mathsf{mem}) =$ ***leak-com**$(\mathsf{id}, \vec{p}, \mathsf{svk})$ that can be inferred from a non-encrypted commit. $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ receives a flag *ack*, a commit node node-id, and a commit message $(c_0, \vec{c})$. Here, $\vec{c}$ is a list of party dependent messages $(\widehat{c}_{\mathsf{id}})_{\mathsf{id}}$; if selective downloading is performed (i.e., $\texttt{flag}_{\mathsf{selDL}}$ is true), then party id only needs to retrieve $(c_0, \widehat{c}_{\mathsf{id}})$ from the server. The adversary $\mathcal{S}$ sets $ack := \texttt{false}$ to report that the protocol fails. $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ then obtains the new updated key package via the `*update-kp` function. This models, in the ideal metadata-hiding CGKA protocol, an honest committer generates a new key package, and it is hidden from the adversary. If the commit protocol succeeds, $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ first asks $\mathcal{S}$ to interpret the injected proposals, i.e., proposals where no node has been created, by calling `*fill-prop`. It then computes the new member set resulting from applying $\vec{p}$ to the current member set by calling `*next-members` (which returns $\perp$if $\vec{p}$ contains invalid proposals).

$\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ then checks the format of $\vec{c}$ specified by $\mathcal{S}$. If $\texttt{flag}_{\mathsf{selDL}}$is true, $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ requires that $\vec{c}$ contains the same number of party-dependent messages as the number of the current members. Else, $\vec{c}$ must be $\perp$.

$\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ then either creates a new commit node or verifies that the existing node is consistent by `*consistent-com`. The adversary $\mathcal{S}$ can specify an existing node-id. This case may happen for example

[14] If the flag $\texttt{flag}_{\mathsf{contHide}}$is false, the key package is asked to the adversary $\mathcal{S}$; this means the key package is known to the adversary because the group creator is known.

[15] If the flag $\texttt{flag}_{\mathsf{contHide}}$is false, the key package is asked to the adversary $\mathcal{S}$; this means the key package is known to the adversary.

when the adversary makes a party process an injected commit message c_0 and then makes another party commit the same c_0 by controlling its randomness. If the specified node Node[node-id] is a detached root, $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ attaches it to id's current node by calling `*attach`. Once the detached root is attached to the main group, the root's tree achieves the same security guarantee as the main tree. Since attaching a detached root changes the topology of the history graph, $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ enforces two invariants: `cons-invariant` enforces the consistency of the graph, and `auth-invariant` enforces the authenticity guarantee.

When add proposals are committed (i.e., addedMem $\neq \perp$), $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ informs the adversary $\mathcal{S}$ that a welcome message is sent and provides ***leak-wel** on input the id's current epoch Ptr[id], the new epoch Node[c_0] and the receiver's identity id_t. If $\texttt{flag}_{\text{contHide}} = \texttt{false}$, then $\mathcal{S}$ obtains all the information that can be inferred from a non-encrypted welcome message. $\mathcal{S}$ returns a simulated welcome message $\widehat{w}$ to $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$. We note that in prior definitions [Has+21b; Has+21c], the commit and welcome messages were simultaneously simulated by $\mathcal{S}$. We consciously divide this process into two. This allows us to model the fact that a welcome message does not necessarily leak information about the group. That is, the server can observe that a party id is invited to some group but will not know which group.[16]. $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ assigns the welcome message to the commit node created above. Finally, $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ marks whether the current epoch is secure or not using `*mark-content-hiden-epoch` and returns $(c_0, \vec{c}, \vec{w})$ to the calling (dummy) party id, which simply relays it to the environment $\mathcal{Z}$

Processing commits (See Figure 5.8) A party id can be invoked by the environment $\mathcal{Z}$ to process a commit message with an associating list of proposals $(c_0, \widehat{c}, \vec{p})$. We explain the case where selective downloading is performed (i.e., $\texttt{flag}_{\text{selDL}}$ is true). $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ first checks that $\widehat{c}$ is the correct id-dependent message associated with c_0, and outputs $\perp$if it is incorrect. (In case $\texttt{flag}_{\text{selDL}}$ is false, $\widehat{c}$ must be $\perp$.) $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ then calls $\mathcal{S}$ on input ***leak-prop**(id) and $(c_0, \widehat{c}, \vec{p})$. $\mathcal{S}$ sets *ack* := `false` to report that the protocol failed. If the process succeeds, $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ first asks the adversary to interpret the injected proposals by calling `*fill-prop`. $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ then either creates a new commit node or verifies that the existing node is consistent. The adversary can specify the existing node-id. If the node corresponding to c_0 does not exist and the adversary does not specify any existing node, $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ checks the validity of $\vec{p}$ and creates a new commit node with the committer identity orig' and its signature key svk' which are $\mathcal{S}$ interprets from $(c_0, \vec{c}, \vec{p})$. Note that the new node holds the same group identity as id's current node and the epoch is incremented. If the commit message was assigned a node (i.e., Node[c_0] $\neq \perp$) or the adversary $\mathcal{S}$ specifies an existing node, $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ checks the validity of the group identity and epoch and enforces that it is a valid successor of id's current node by calling `*valid-successor`. If c_0 is assigned to a detached root, $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ attaches the root to id's current node. If c_0 is not assigned a node, $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ assigns the adversary-specified node-id to c_0.

Finally, if c_0 removes id, $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ sets Ptr[id] $= \perp$. Otherwise, $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ updates id's secrets if necessary and moves Ptr[id] to the new commit node. The calling party receives the committer's identity, the semantics of the applied proposals, and the list of (id, svk)-pair.

Joining a group (See Figure 5.9) A party id can be invoked by the environment $\mathcal{Z}$ to join a group using the welcome message $\widehat{w}$. $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ forwards $(\text{id}, \widehat{w})$ to the adversary $\mathcal{S}$ and receives the interpreted result. As usual, the adversary sets *ack* := `false` to report that the protocol failed. If the process succeeds, $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ identifies the commit node node-id = Wel[id, $\widehat{w}$] corresponding to $\widehat{w}$. If this is the first time $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ sees $\widehat{w}$, i.e., Wel[id, $\widehat{w}$] $= \perp$, $\mathcal{S}$ can specify node-id$'$. If the commit node for node-id$'$ does not exist (i.e., Node[node-id$'$] $= \perp$), $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ creates a new detached root where all the stored values are chosen by $\mathcal{S}$. Finally, $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ updates id's secrets (registered in the key service $\mathcal{F}_{\text{KS}}$) and returns the state of the joining

[16]Note that we can capture the situation where a welcome message leaks the group by defining ***leak-wel** to output the node pointer of the commit message.

group (the committer's identity, the group identity, the epoch, and the list of $(\mathsf{id}, \mathsf{kp})$-pair) to the calling (dummy) party id.

Group keys (See Figure 5.10) Parties can fetch the current group secret via the `Key` query. The returned group secret k is random if the protocol guarantees its confidentiality (identified by the **safe** predicate). Otherwise, k is set by the adversary. Unlike prior definitions [Has+21b; Has+21c], we also prepare two extra group secret keys: the metadata key $\mathsf{k}_{\mathtt{mh}}$ that can be obtained using $\mathtt{Key}_{\mathtt{mh}}$, and the next metadata key $\mathsf{k}_{\mathtt{mh}}{}'$ that can be obtained using $\mathtt{NextKey}_{\mathtt{mh}}$. While $\mathtt{Key}_{\mathtt{mh}}$ is defined identically to `Key`, $\mathtt{NextKey}_{\mathtt{mh}}$ is a function that allows obtaining the metadata key at the next epoch. These keys are explicitly used to hide the *dynamic* metadata. That is, if we only care about hiding static metadata, these two group secret keys can be safely omitted from the definition.

Corruptions (See Figure 5.11) The adversary $\mathcal{S}$ can corrupt a party id using the `Expose` query. When the adversary $\mathcal{S}$ inputs $(\mathtt{Expose}, \mathsf{id})$ to $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$, the ideal functionality records the following leaked information:

- The current group secret keys and id's key materials (e.g., encryption key and signing key). This is recorded by adding id to the exposed set of id's current node (cf. Line 2 in $(\mathtt{Expose}, \mathsf{id})$ query).
- The key materials created by id during an update or a commit at the current epoch. This is recorded by setting the status of all the child commit nodes created by id (i.e., nodes with $\mathsf{par} = \mathsf{Ptr}[\mathsf{id}]$) to 'bad' (cf. `*update-stat-after-exp` function).
- The current signature signing key ssk. This is recorded by signaling to $\mathcal{F}_{\mathrm{AS}}$ that svk is exposed and sends ssk to the adversary (cf. Line 5 in $(\mathtt{Expose}, \mathsf{id})$ query).

Then, the ideal functionality gives id's current epoch $\mathsf{Ptr}[\mathsf{id}]$, the associated information $\mathsf{Node}[\mathsf{Ptr}[\mathsf{id}]]$, and id's current secret keys stored in the CurrDK array. Also, the adversary is allowed to corrupt a non-group member id as well. In such a case, the key packages that id registered to the key service $\mathcal{F}_{\mathrm{KS}}$ are leaked. $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$ signals to $\mathcal{F}_{\mathrm{KS}}$ that key packages (including the signing key) are exposed and send the corresponding decryption keys and signing keys to the adversary (cf. Line 8 in $(\mathtt{Expose}, \mathsf{id})$ query).

If an adversary is allowed to compromise key materials that can be used to compute a group secret key, which $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$ has already assigned random values, then $\mathcal{Z}$ trivially distinguishes a real protocol from an ideal protocol. For instance, if $\mathcal{Z}$ queries `Key` for a commit node where the predicate **safe** is true, then $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$ assigns a random value to the group secret key k. Then, if the adversary at some later point compromises a party id via an $(\mathtt{Expose}, \mathsf{id})$ and can compute the *real* group secret key k' from the compromised key materials, $\mathcal{Z}$ can distinguish the two protocols by checking if $\mathsf{k} = \mathsf{k}'$.

To avoid such trivial attacks, we restrict the environment to not be able to corrupt key materials for those commit nodes with $\mathsf{chall} = \mathtt{true}$ or $\mathsf{conthide} = \mathtt{true}$. The former is identical to those used in prior works [Has+21b; Has+21c]; if a random group secret key was set, then $\mathcal{Z}$ cannot corrupt a party that allows recovering of the real group secret key. The latter is new to this work. For static metadata-hiding, recall that we must encrypt the proposal and commits. This is modeled in $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$ by requiring the adversary $\mathcal{S}$ to create the (encrypted) proposal and commits *without* knowing the message when the predicate **safe** is true. If $\mathcal{Z}$ were to compromise a party that allows recovering the real group secret key, it can try to decrypt the encrypted proposal or commit to trivially distinguish between a real and ideal protocol. $\mathsf{conthide} = \mathtt{true}$ indicates that a commit node created random encryption and restricts $\mathcal{Z}$ from later corrupting it. Note that this implies that the predicate **safe** for honestly generated commit nodes cannot be switched from true to false once created. This is in sharp contrast to previous definitions since the group secret key was never implicitly used as part of the real protocol. We note that this is not a weakness of our security model but rather a natural consequence of considering CGKAs in a larger system.

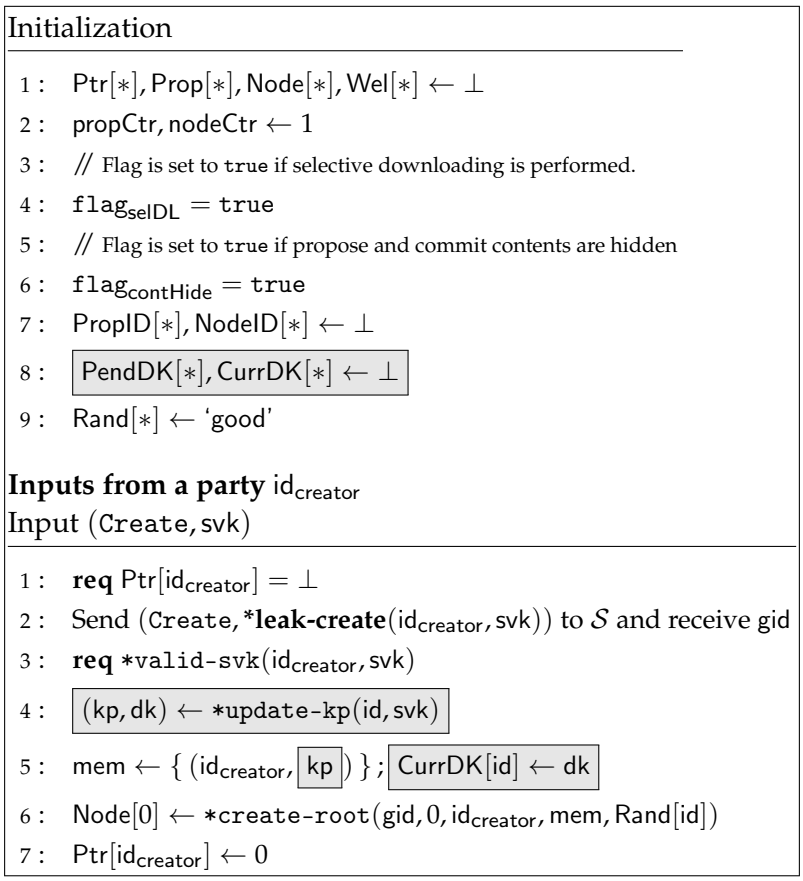


FIGURE 5.5: The ideal static metadata-hiding CGKA functionality $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$: Initialization and `Create` function. The modifications for the functionality to keep track of key packages are highlighted in gray.

Inputs from a party id
Input $(\texttt{Propose}, \mathsf{act})$, $\mathsf{act} \in \{$ 'upd'-svk, 'add'-id_t-kp_t, 'rem'-id_t $\}$

1 : **req** $\mathsf{Ptr}[\mathsf{id}] \neq \bot$
2 : Send $(\texttt{Propose}, \textbf{*leak-prop}(\mathsf{id}, \mathsf{act}))$ to $\mathcal{S}$ and receive $(ack, \text{prop-id}, p)$
3 : **req** *ack*
4 : **if** act = 'upd'-svk **then**
5 : **req** $\texttt{*valid-svk}(\mathsf{id}, \mathsf{svk})$
6 : $(\mathsf{kp}, \mathsf{dk}) \leftarrow \texttt{*update-kp}(\mathsf{id}, \mathsf{svk})$
7 : $\mathsf{act} \leftarrow$ 'upd'-kp
8 : $\mathsf{PendDK}[\mathsf{kp}] \leftarrow \mathsf{dk}$
9 : **if** act = 'add'-id_t-kp_t **then req** $\texttt{*valid-kp}(\mathsf{id}_t, \mathsf{kp}_t)$
10 : **if** $\mathsf{PropID}[p] = \bot \wedge \text{prop-id} = \bot$ **then**
11 : $\mathsf{Prop}[\text{propCtr}] \leftarrow \texttt{*create-prop}(\mathsf{Ptr}[\mathsf{id}], \mathsf{id}, \mathsf{act}, \mathsf{Rand}[\mathsf{id}])$
12 : $\mathsf{PropID}[p] \leftarrow \text{propCtr}; \text{propCtr}{+}{+}$
13 : **else**
14 : **if** $\mathsf{PropID}[p] = \bot$ **then** $(\text{prop-id}', \mathsf{PropID}[p]) \leftarrow (\text{prop-id}, \text{prop-id})$
15 : **else** $\text{prop-id}' \leftarrow \mathsf{PropID}[p]$
16 : $\texttt{*consistent-prop}(\text{prop-id}', \mathsf{id}, \mathsf{act})$
17 : **if** $\mathsf{act} =$ 'upd'-svk $\wedge\ \mathsf{Rand}[\mathsf{id}] =$ 'bad' **then**
18 : Send $(\texttt{exposed}, \mathsf{id}, \mathsf{svk})$ to $\mathcal{F}_{\text{AS}}$
19 : // Mark whether generated messages hide contents (static metadata)
20 : $\texttt{*mark-content-hiden-epoch}(\mathsf{Ptr}[\mathsf{id}])$
21 : **return** p

FIGURE 5.6: The ideal static metadata-hiding CGKA functionality $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$: `Propose` function. The modifications in order for the functionality to keep track of key packages are highlighted in gray.

Input $(\texttt{Commit}, \vec{p}, \mathsf{svk})$

1 : **req** $\mathsf{Ptr}[\mathsf{id}] \neq \bot$
2 : Send $(\texttt{Commit}, \textbf{*leak-com}(\mathsf{id}, \vec{p}, \mathsf{svk}))$ to $\mathcal{S}$ and receive $(ack, \mathsf{node\text{-}id}, c_0, \vec{c})$
3 : **req** $\texttt{*valid-svk}(\mathsf{id}, \mathsf{svk})$
4 : $\boxed{(\mathsf{kp}_{\mathsf{new}}, \mathsf{dk}_{\mathsf{new}}) \leftarrow \texttt{*update-kp}(\mathsf{id}, \mathsf{svk})}$; $\boxed{\mathsf{PendDK}[\mathsf{kp}_{\mathsf{new}}] \leftarrow \mathsf{dk}_{\mathsf{new}}}$
5 : **req** $\texttt{*succeed-com}(\mathsf{id}, \vec{p}, \mathsf{kp}_{\mathsf{new}}) \vee ack$
6 : $\texttt{*fill-prop}(\vec{p})$
7 : $(\mathsf{mem}', *) \leftarrow \texttt{*next-members}(\mathsf{Ptr}[\mathsf{id}], \mathsf{id}, \vec{p}, \boxed{\mathsf{kp}_{\mathsf{new}}})$
8 : **assert** $\mathsf{mem}' \neq \bot \wedge (\mathsf{id}, \boxed{\mathsf{kp}_{\mathsf{new}}}) \in \mathsf{mem}'$
9 : // If selective downloading is performed, then member specific $\vec{c}$ has the same size as the current member. Otherwise, $\vec{c}$ is $\bot$
10 : **if** $\texttt{flag}_{\mathsf{selDL}}$ **then**
11 : **assert** $|\vec{c}| = |\mathsf{Node}[\mathsf{Ptr}[\mathsf{id}]].\mathsf{mem}|$
12 : **else**
13 : **assert** $\vec{c} = \bot$
14 : **if** $\mathsf{NodeID}[c_0] = \bot \wedge \mathsf{node\text{-}id} = \bot$ **then**
15 : $\mathsf{Node}[\mathsf{nodeCtr}] \leftarrow \texttt{*create-child}(\mathsf{Ptr}[\mathsf{id}], \mathsf{id}, \vec{p}, \vec{c}, \mathsf{mem}', \mathsf{Rand}[\mathsf{id}])$
16 : $\mathsf{NodeID}[c_0] \leftarrow \mathsf{nodeCtr}; \mathsf{nodeCtr}{+}{+}$
17 : **else**
18 : **if** $\mathsf{NodeID}[c_0] = \bot$ **then** $(\mathsf{node\text{-}id}', \mathsf{NodeID}[c_0]) \leftarrow (\mathsf{node\text{-}id}, \mathsf{node\text{-}id})$
19 : **else** $\mathsf{node\text{-}id}' \leftarrow \mathsf{NodeID}[c_0]$
20 : $\texttt{*consistent-com}(\mathsf{node\text{-}id}', \mathsf{id}, \vec{p}, \mathsf{mem})$
21 : **if** $\mathsf{Node}[\mathsf{node\text{-}id}'].\mathsf{par} = \bot$ **then**
22 : $\texttt{*attach}(\mathsf{node\text{-}id}', \mathsf{id}, \vec{p})$ // Create welcome message for added members
23 : $\mathsf{addedMem} \leftarrow \mathsf{Node}[\mathsf{NodeID}[c_0]].\mathsf{mem} \setminus \mathsf{Node}[\mathsf{Ptr}[\mathsf{id}]].\mathsf{mem}$
24 : $\vec{w} \leftarrow \emptyset$
25 : **foreach** $(\mathsf{id}_t, *) \in \mathsf{addedMem}$ **do**
26 : Send $(\texttt{Welcome}, \textbf{*leak-wel}(\mathsf{Ptr}[\mathsf{id}], \mathsf{NodeID}[c_0], \mathsf{id}_t))$ to $\mathcal{S}$ and receive $(ack, \widehat{w})$
27 : **req** ack
28 : **parse** $(\mathsf{id}_t, *) \leftarrow \widehat{w}$
29 : **assert** $\mathsf{Wel}[\mathsf{id}_t, \widehat{w}] \in \{\bot, \mathsf{NodeID}[c_0]\}$
30 : $\mathsf{Wel}[\mathsf{id}_t, \widehat{w}] \leftarrow \mathsf{NodeID}[c_0]$
31 : $\vec{w} \mathrel{+}\!\!\leftarrow \widehat{w}$
32 : **assert** $\texttt{cons-invariant} \wedge \texttt{auth-invariant}$
33 : **if** $\mathsf{Rand}[\mathsf{id}] =$ 'bad' **then**
34 : Send $(\texttt{exposed}, \mathsf{id}, \mathsf{svk})$ to $\mathcal{F}_{\mathrm{AS}}$
35 : $\texttt{*mark-content-hiden-epoch}(\mathsf{Ptr}[\mathsf{id}])$ // Mark whether generated messages hide contents (static metadata)
36 : **if** $\vec{w} \neq \emptyset$ **then** $(\mathsf{NodeID}[c_0])$
37 : **return** $(c_0, \vec{c}, \vec{w})$

FIGURE 5.7: The ideal static metadata-hiding CGKA functionality $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$: `Commit` function. The modifications in order for the functionality to keep track of key packages are indicated by the box.

Input $(\texttt{Process}, c_0, \widehat{c}, \vec{p})$

1 : **req** $\mathsf{Ptr}[\mathsf{id}] \neq \bot$
2 : // If c_0 and $\vec{p}$ were generated honestly, then use the existing node-id $\neq \bot$.
// Otherwise, the group secret must be exposed and $\mathcal{S}$ decides where to attach $\mathsf{Ptr}[\mathsf{id}]$.
3 : Send $(\texttt{Process}, \textbf{*leak-proc}(\mathsf{id}), c_0, \widehat{c}, \vec{p})$ to $\mathcal{S}$ and receive $(ack, \mathsf{node\text{-}id}, \mathsf{orig}', \boxed{\mathsf{kp}'})$
4 : **req** $\texttt{*succeed-proc}(\mathsf{id}, c_0, \widehat{c}, \vec{p}) \vee ack$
5 : $\texttt{*fill-prop}(\vec{p})$
6 : // If selective downloading is performed, then only the member-specific commitment $\widehat{c}$ is accepted when c_0 is honestly generated.
if $\mathtt{flag}_{\mathsf{selDL}}$ **then**
7 : $\mathsf{node\text{-}id} \leftarrow \mathsf{NodeID}[c_0]$
8 : **if** $\mathsf{node\text{-}id} \neq \bot \wedge \mathsf{Node}[\mathsf{node\text{-}id}].\mathsf{stat} = \text{`good'}$ **then**
9 : $\mathsf{index}_{\mathsf{id}} \leftarrow \mathsf{Node}[\mathsf{Ptr}[\mathsf{id}]].\mathsf{indexOf}(\mathsf{id})$
10 : **req** $\widehat{c} = \mathsf{Node}[\mathsf{node\text{-}id}].\mathsf{vcom}[\mathsf{index}_{\mathsf{id}}]$
11 : **else assert** $\widehat{c} = \bot$ // Otherwise, $\widehat{c}$ is $\bot$.
12 : **if** $\mathsf{NodeID}[c_0] = \bot \wedge \mathsf{node\text{-}id} = \bot$ **then** // If node-id $= \bot$, create a new node
13 : **try** $(\mathsf{mem}', *) \leftarrow \texttt{*next-members}(\mathsf{Ptr}[\mathsf{id}], \mathsf{orig}', \vec{p}, \boxed{\mathsf{kp}'})$
14 : **assert** $\mathsf{mem}' \neq \bot$
15 : $\mathsf{Node}[\mathsf{nodeCtr}] \leftarrow \texttt{*create-child}(\mathsf{Ptr}[\mathsf{id}], \mathsf{orig}', \vec{p}, \mathsf{mem}', \text{`adv'})$
16 : $(\mathsf{NodeID}[c_0], \mathsf{node\text{-}id}) \leftarrow (\mathsf{nodeCtr}, \mathsf{nodeCtr})$
17 : $\mathsf{nodeCtr}{+}{+}$
18 : **else** // If node-id $\neq \bot$, check consistency with existing node.
19 : **if** $\mathsf{NodeID}[c_0] = \bot$ **then** $(\mathsf{node\text{-}id}', \mathsf{NodeID}[c_0]) \leftarrow (\mathsf{node\text{-}id}, \mathsf{node\text{-}id})$
20 : **else** $\mathsf{node\text{-}id}' \leftarrow \mathsf{NodeID}[c_0]$
21 : // After processing, require gid to remain the same and epoch to be incremented by 1.
22 : **assert** $\mathsf{Node}[\mathsf{Ptr}[\mathsf{id}]].\mathsf{gid} = \mathsf{Node}[\mathsf{node\text{-}id}'].\mathsf{gid}$
23 : **assert** $\mathsf{Node}[\mathsf{Ptr}[\mathsf{id}]].\mathsf{epoch} = \mathsf{Node}[\mathsf{node\text{-}id}'].\mathsf{epoch} + 1$
24 : $\mathsf{id}_c \leftarrow \mathsf{Node}[\mathsf{node\text{-}id}'].\mathsf{orig}; \mathsf{kp}_c \leftarrow \mathsf{Node}[\mathsf{node\text{-}id}'].\mathsf{mem}[\mathsf{id}_c]$
25 : $(\mathsf{mem}', *) \leftarrow \texttt{*next-members}(\mathsf{Ptr}[\mathsf{id}], \mathsf{id}_c, \vec{p}, \mathsf{kp}_c)$
26 : **assert** $\mathsf{mem}' \neq \bot$
27 : $\texttt{*valid-successor}(\mathsf{node\text{-}id}', \mathsf{id}_c, \vec{p}, \mathsf{mem}')$
28 : // Mark whether generated messages hide contents (static metadata)
29 : **if** $\mathsf{Node}[\mathsf{node\text{-}id}'].\mathsf{par} = \bot$ **then** $\texttt{*attach}(\mathsf{node\text{-}id}', \mathsf{id}, \vec{p})\texttt{*mark-content-hiden-epoch}(\mathsf{Ptr}[\mathsf{id}])$
30 : **if** $\exists p \in \vec{p} : \mathsf{Prop}[p].\mathsf{act} = \text{`rem'-id}$ **then** $\mathsf{Ptr}[\mathsf{id}] \leftarrow \bot$
31 : **else**
32 : **assert** $(\mathsf{id}, *) \in \mathsf{Node}[\mathsf{NodeID}[c_0]].\mathsf{mem}$
33 : **if** $\mathsf{Node}[\mathsf{Ptr}[\mathsf{id}]].\mathsf{mem}[\mathsf{id}] \neq \mathsf{Node}[\mathsf{NodeID}[c_0]].\mathsf{mem}[\mathsf{id}]$ **then** // Fetch new dk if key package is updated
34 : $\boxed{\mathsf{kp} \leftarrow \mathsf{Node}[\mathsf{NodeID}[c_0]].\mathsf{mem}[\mathsf{id}]}; \boxed{\mathsf{CurrDK}[\mathsf{id}] \leftarrow \mathsf{PendDK}[\mathsf{kp}]}$
35 : $\mathsf{Ptr}[\mathsf{id}] \leftarrow \mathsf{NodeID}[c_0]$
36 : **assert** $\texttt{cons-invariant} \wedge \texttt{auth-invariant}$
37 : **return** $\texttt{*output-proc}(\mathsf{node\text{-}id}')$

FIGURE 5.8: The ideal static metadata-hiding CGKA functionality $\mathcal{F}_{\mathrm{CGKA}}^{\mathrm{ctxt}}$: `Process` function. The modifications in order for the functionality to keep track of key packages are indicated by the box.

Input $(\texttt{Join}, \mathsf{id}, \widehat{w})$

1 : **req** $\mathsf{Ptr}[\mathsf{id}] = \bot$
2 : // If $\widehat{w}$ was generated honestly, then use the existing node-id $\neq \bot$.
3 : Send $(\texttt{Join}, \mathsf{id}, \widehat{w})$ to $\mathcal{S}$ and receive $(ack, \mathsf{node\text{-}id}', \mathsf{gid}', \mathsf{epoch}', \mathsf{orig}', \mathsf{mem}')$
4 : **req** $*\texttt{succeed-wel}(\mathsf{id}, \widehat{w}) \vee ack$
5 : $\mathsf{node\text{-}id} \leftarrow \mathsf{Wel}[\mathsf{id}, \widehat{w}]$
6 : **if** $\mathsf{node\text{-}id} = \bot$ **then**
7 : **if** $\mathsf{Node}[\mathsf{node\text{-}id}'] \neq \bot$ **then** $\mathsf{node\text{-}id} \leftarrow \mathsf{node\text{-}id}'$
8 : **else**
9 : $\mathsf{Node}[\mathsf{nodeCtr}] \leftarrow *\texttt{create-root}(\mathsf{gid}', \mathsf{epoch}', \mathsf{orig}', \mathsf{mem}', \text{'adv'})$
10 : $\mathsf{node\text{-}id} \leftarrow \mathsf{nodeCtr}$
11 : $\mathsf{nodeCtr}{+}{+}$
12 : $\mathsf{Wel}[\mathsf{id}, \widehat{w}] \leftarrow \mathsf{node\text{-}id}$
13 : $\mathsf{kp} \leftarrow \mathsf{Node}[\mathsf{node\text{-}id}].\mathsf{mem}[\mathsf{id}]$
14 : $\boxed{\mathsf{CurrDK}[\mathsf{id}] \leftarrow \mathsf{DK}[\mathsf{id}, \mathsf{kp}]}$ // Fetch the registered dk used to join
15 : $\mathsf{Ptr}[\mathsf{id}] \leftarrow \mathsf{node\text{-}id}$
16 : **assert** $(\mathsf{id}, *) \in \mathsf{Node}[\mathsf{node\text{-}id}].\mathsf{mem}$
17 : **assert** $\texttt{cons-invariant} \wedge \texttt{auth-invariant}$
18 : **return** $*\texttt{output-join}(\mathsf{node\text{-}id})$

FIGURE 5.9: The ideal static metadata-hiding CGKA functionality $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$: `Join` function. The modifications in order for the functionality to keep track of key packages are indicated by the box.

Input $(\texttt{Key})$

1 : **req** $\mathsf{Ptr}[\mathsf{id}] \neq \bot$
2 : **if** $\mathsf{Node}[\mathsf{Ptr}[\mathsf{id}]].\mathsf{key} = \bot$ **then**
3 : $*\texttt{set-key}(\mathsf{Ptr}[\mathsf{id}])$
4 : **return** $\mathsf{Node}[\mathsf{Ptr}[\mathsf{id}]].\mathsf{key}$

Input $(\texttt{Key}_{\texttt{mh}})$

1 : **req** $\mathsf{Ptr}[\mathsf{id}] \neq \bot$
2 : **if** $\mathsf{Node}[\mathsf{Ptr}[\mathsf{id}]].\mathsf{k}_{\mathsf{mh}} = \bot$ **then**
3 : $*\texttt{set-key}(\mathsf{Ptr}[\mathsf{id}])$
4 : **return** $\mathsf{Node}[\mathsf{Ptr}[\mathsf{id}]].\mathsf{k}_{\mathsf{mh}}$

Input $(\texttt{NextKey}_{\texttt{mh}}, c_0)$

1 : **req** $\mathsf{Ptr}[\mathsf{id}] \neq \bot \wedge \mathsf{NodeID}[c_0] \neq \bot$
2 : **req** $\mathsf{Node}[\mathsf{NodeID}[c_0]].\mathsf{par} = \mathsf{Ptr}[\mathsf{id}] \wedge \mathsf{Node}[\mathsf{NodeID}[c_0]].\mathsf{orig} = \mathsf{id}$
3 : **if** $\mathsf{Node}[\mathsf{NodeID}[c_0]].\mathsf{k}_{\mathsf{mh}} = \bot$ **then**
4 : $*\texttt{set-key}(\mathsf{NodeID}[c_0])$
5 : **return** $\mathsf{Node}[\mathsf{NodeID}[c_0]].\mathsf{k}_{\mathsf{mh}}$

FIGURE 5.10: The ideal static metadata-hiding CGKA functionality $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$: `Key`, $\texttt{Key}_{\texttt{mh}}$, and $\texttt{NextKey}_{\texttt{mh}}$ functions. The last two $\texttt{Key}_{\texttt{mh}}$ and $\texttt{NextKey}_{\texttt{mh}}$ are to be used in a higher layer protocol. The modifications in order for the functionality to keep track of key packages are indicated by the box.

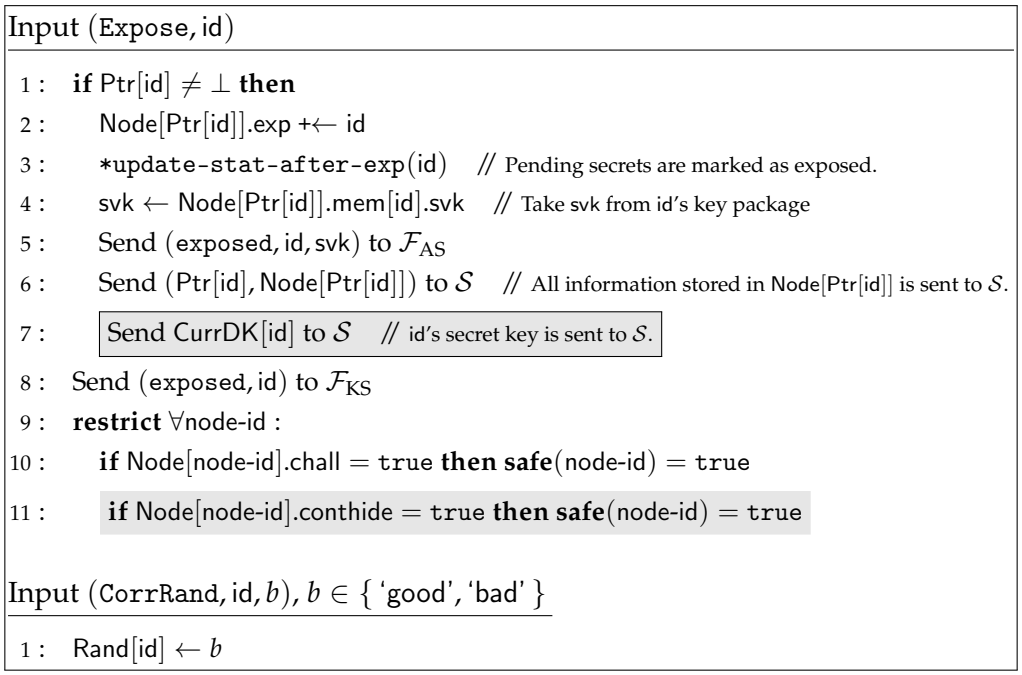


FIGURE 5.11: The static metadata-hiding CGKA functionality $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$: Corruptions from the adversary $\mathcal{S}$. The difference between those of $\mathcal{F}_{\mathrm{CGKA}}$ [Has+21c] is highlighted in gray. The modifications in order for the functionality to keep track of key packages are indicated by the box.

*create-root(gid, epoch, id, mem, stat)

1 : **return** new node with par ← ⊥, orig ← id, gid ← gid
epoch ← epoch, prop ← ⊥, mem ← mem, stat ← stat.

*create-child(node-id, id, $\vec{p}$, $\vec{c}$, mem, stat)

1 : (gid, epoch) ← (Node[Ptr[id]].gid, Node[Ptr[id]].epoch)
2 : // Create a new node with an incremented epoch.
3 : **return** new node with par ← node-id, orig ← id,
epoch ← epoch + 1, prop ← $\vec{p}$, vcom ← $\vec{c}$,
mem ← mem, stat ← stat.

*create-prop(node-id, id, act, stat)

1 : **return** new node with par ← node-id, orig ← id,
act ← act, stat ← stat.

*fill-prop($\vec{p}$)

1 : **foreach** $p \in \vec{p}$ **s.t.** PropID[p] = ⊥ **do**
2 : Send (Propose, Ptr[id], p) to $\mathcal{S}$ and
receive (prop-id, orig, act)
3 : **if** prop-id = ⊥ **then**
4 : Prop[propCtr] ← *create-prop(Ptr[id], orig, act, 'adv')
5 : PropID[p] ← propCtr
6 : propCtr++
7 : // If $\texttt{flag}_{\mathsf{contHide}}$ = true and p includes the same plain content,
8 : // then $\mathcal{S}$ outputs prop-id ≠ ⊥.
// In this case, check consistency with the exiting node.
9 : **else**
10 : *consistent-prop(prop-id, orig, act)
11 : PropID[p] ← prop-id

*set-key(node-id)

1 : **if safe**(node-id) **then**
2 : Node[node-id].key ←$ $\mathcal{K}$
3 : Node[node-id].chall ← true
4 : **else**
5 : Send (Key, id) to $\mathcal{S}$ and
receive k
6 : Node[node-id].key ← k
7 : Node[node-id].chall ← false

*mark-content-hiden-epoch(node-id)

1 : **if safe**(node-id) **then**
2 : Node[node-id].conthide ← true
3 : **else**
4 : Node[node-id].conthide ← false

*update-stat-after-exp(id)

1 : **foreach** prop-id **s.t.** Prop[prop-id] ≠ ⊥
∧ Prop[prop-id].par = Ptr[id]
∧ Prop[prop-id].orig = id
∧ Prop[prop-id].act = 'upd'-∗ **do**
2 : Prop[prop-id].stat ← 'bad'
3 : **foreach** node-id **s.t.** Node[node-id] ≠ ⊥
∧ Node[node-id].par = Ptr[id]
∧ Node[node-id].orig = id **do**
4 : Node[node-id].stat ← 'bad'

FIGURE 5.12: The helper functions for $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$: Creating and maintaining the history graph.

*valid-kp(id, kp)

1 : *ack* ← query (has-kp, id, kp) to $\mathcal{F}_{\text{KS}}$
2 : **return** *ack*

*valid-svk(id, svk′)

1 : **if** Ptr[id] ≠ ⊥ **then**
2 : svk ← Node[Ptr[id]].mem[id]
3 : **if** svk′ ≠ ⊥ ∧ svk = svk′ **then**
4 : **return** true
5 : *ack* ← query (has-ssk, id, svk′) to $\mathcal{F}_{\text{AS}}$
6 : **return** *ack*

*update-kp (id, svk)

1 : **if** $\texttt{flag}_{\mathsf{contHide}}$ ∧ **safe**(Ptr[id]) **then**
2 : ssk ← query (get-ssk, svk) to $\mathcal{F}_{\text{AS}}$ on behalf of id
3 : **if** Rand[id] = 'good' **then**
4 : (kp, dk) ← genKP(id, svk, ssk)
5 : **else**
6 : Send (rnd, id) to the adversary and receive r
7 : (kp, dk) ← genKP(id, svk, ssk; r)
8 : **else**
9 : Receive (kp, dk) from $\mathcal{S}$
10 : **return** (kp, dk)

FIGURE 5.13: The helper functions for $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$: Related to keys. The *update-kp function highlighted by the box is newly introduced to manage key packages in the functionality.

*output-proc(node-id)

1 : id_c ← Node[node-id].orig
2 : kp_c ← Node[node-id].mem[id_c]
3 : (∗, propSem) ← *next-members(node-id, id_c, Node[node-id].prop, kp_c)
4 : **return** (Node[node-id].orig, propSem, Node[node-id].mem)

*output-join(node-id)

1 : gid ← Node[node-id].gid
2 : epoch ← Node[node-id].epoch
3 : mem ← Node[node-id].mem
4 : id_c ← Node[node-id].orig
5 : **return** (id_c, gid, epoch, mem)

FIGURE 5.14: The helper functions for $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$: Defining output of process and join protocols.

`*next-members`$(\text{node-id}, \text{id}_c, \vec{p}, \text{kp}_c)$

1 : **if** Node[node-id] $\neq \bot \wedge (\text{id}_c, *) \in$ Node[node-id].mem
$\wedge \forall p \in \vec{p} : \text{Prop}[p] \neq \bot \wedge \text{Prop}[p].\text{par} = \text{node-id}$ **then**
2 : $\vec{p}_{\text{'upd'}} \| \vec{p}_{\text{'rem'}} \| \vec{p}_{\text{'add'}} \leftarrow$ **`*sort-proposals`**$(\vec{p})$
3 : mem $\leftarrow$ Node[node-id].mem
4 : mem $-\!\leftarrow (\text{id}_c, *)$; mem $+\!\leftarrow (\text{id}_c, \text{kp}_c)$
5 : $L \leftarrow \{ \text{id}_c \}$ // set of updated parties
6 : **foreach** $p \in \vec{p}_{\text{'upd'}}$ **do**
7 : $(\text{id}_s, \text{'upd'-kp}) \leftarrow (\text{Prop}[p].\text{orig}, \text{Prop}[p].\text{act})$
8 : **if** $\neg((\text{id}_s, *) \in \text{mem} \wedge \text{id}_s \notin L)$
9 : **then return** $(\bot, \bot)$
10 : mem $-\!\leftarrow (\text{id}_s, *)$; mem $+\!\leftarrow (\text{id}_s, \text{kp})$
11 : $L +\!\leftarrow \text{id}_s$
12 : **foreach** $p \in \vec{p}_{\text{'rem'}}$ **do**
13 : $(\text{id}_s, \text{'rem'-id}_t) \leftarrow (\text{Prop}[p].\text{orig}, \text{Prop}[p].\text{act})$
14 : **if** $\neg((\text{id}_s, *) \in \text{mem} \wedge (\text{id}_t \in \text{mem} \wedge \text{id}_t \notin L))$
15 : **then return** $(\bot, \bot)$
16 : mem $-\!\leftarrow (\text{id}_t, *)$
17 : **foreach** $p \in \vec{p}_{\text{'add'}}$ **do**
18 : $(\text{id}_s, \text{'add'-id}_t\text{-kp}_t) \leftarrow (\text{Prop}[p].\text{orig}, \text{Prop}[p].\text{act})$
19 : **if** $\neg((\text{id}_s, *) \in \text{mem} \wedge (\text{id}_t, *) \notin \text{mem})$
20 : **then return** $(\bot, \bot)$
21 : mem $+\!\leftarrow (\text{id}_t, \text{kp}_t)$
22 : $P \leftarrow ()$
23 : **foreach** $p \in \vec{p}_{\text{'upd'}} \| \vec{p}_{\text{'rem'}} \| \vec{p}_{\text{'add'}}$ **do**
24 : $P \mathbin{+\!\!+\!\leftarrow} (\text{Prop}[\text{PropID}[p]].\text{orig}, \text{Prop}[\text{PropID}[p]].\text{act})$
25 : **return** (mem, P)
26 : **else**
27 : **return** $(\bot, \bot)$

`*sort-proposals`$(\vec{p})$

1 : $\vec{p}_{\text{'upd'}}, \vec{p}_{\text{'rem'}}, \vec{p}_{\text{'add'}} \leftarrow ()$
2 : **foreach** $p \in \vec{p}$ **do**
3 : $\text{act}_p \leftarrow \text{Prop}[\text{PropID}[p]].\text{act}$
4 : **if** $\text{act}_p =$ 'upd'- $*$ **then**
5 : $\vec{p}_{\text{'upd'}} \mathbin{+\!\!+\!\leftarrow} p$
6 : **elseif** $\text{act}_p =$ 'rem'- $*$ **then**
7 : $\vec{p}_{\text{'rem'}} \mathbin{+\!\!+\!\leftarrow} p$
8 : **if** $\text{act}_p =$ 'add'- $*$ **then**
9 : $\vec{p}_{\text{'add'}} \mathbin{+\!\!+\!\leftarrow} p$
10 : **return** $\vec{p}_{\text{'upd'}} \| \vec{p}_{\text{'rem'}} \| \vec{p}_{\text{'add'}}$

FIGURE 5.15: The helper functions for $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$: Determining the group state after applying a commit. **`*sort-proposals`**$(\vec{p})$ orders applying proposals.

*consistent-prop(prop-id, id, act)

1 : **assert** Prop[prop-id].par = Ptr[id]
∧ Prop[prop-id].orig = id
∧ Prop[prop-id].act = act

*consistent-com(node-id, id, $\vec{p}$, mem)

1 : *valid-successor(node-id, id, $\vec{p}$, mem)
2 : **assert** Rand[id] = 'bad' ∧ Node[node-id].orig = id

*valid-successor(node-id, id, $\vec{p}$, mem)

1 : **assert** Node[node-id] ≠ ⊥
∧ Node[node-id].mem = mem
∧ Node[node-id].prop ∈ { ⊥, $\vec{p}$ }
∧ Node[node-id].par ∈ { ⊥, Ptr[id] }

*attach(node-id, id, $\vec{p}$)

1 : // Cannot attach to the original honest root node-id = 0
2 : **assert** node-id ≠ 0
3 : Node[node-id].par ← Ptr[id]
4 : Node[node-id].prop ← $\vec{p}$

*succeed-com(id, $\vec{p}$, kp)

1 : **return** *next-members(Ptr[id], id, $\vec{p}$, kp) ≠ (⊥, ⊥)
∀$p \in \vec{p}$:
prop-id := PropID[p] ≠ ⊥
∧ Prop[prop-id].stat ≠ 'adv'

*succeed-proc(id, c_0, $\widehat{c}$, $\vec{p}$)

1 : node-id ← NodeID[c_0]
2 : index_{id} ← Node[node-id].indexOf(id)
3 : **return** node-id ≠ ⊥ ∧ Node[node-id] ≠ ⊥
∧ Node[node-id].par = Ptr[id]
∧ Node[node-id].prop = $\vec{p}$
∧ Node[node-id].stat ≠ 'adv'
∧ ∀$p \in \vec{p}$: Prop[PropID[p]].stat ≠ 'adv'
∧ Node[node-id].vcom[index_{id}] = $\widehat{c}$

*succeed-wel(id, $\widehat{w}$)

1 : node-id ← Wel[id, $\widehat{w}$]
2 : c ← Node[node-id]
3 : addedMem ← (c.mem \ Node[c.par].mem)
4 : **return** node-id ≠ ⊥
∧ $c \neq \bot \wedge c$.stat ≠ 'adv'
∧ (id, ∗) ∈ addedMem

FIGURE 5.16: The helper functions for $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$: Checking consistency and correctness.

`auth-invariant`

return `true` **iff**

(a) $\forall$node-id with node-id$_p$:= Node[node-id].par, node-id$_p \neq \perp$ and id := Node[node-id].orig :
if Node[node-id].stat = 'adv' **then sig-inj-allowed**(node-id$_p$, id) $\wedge$ **mac-inj-allowed**(node-id$_p$) and

(b) $\forall p$ with node-id$_p$:= Prop[p].par and id := Prop[p].orig :
if Prop[p].stat = 'adv' **then sig-inj-allowed**(node-id$_p$, id) $\wedge$ **mac-inj-allowed**(node-id$_p$) and

(c) $\forall$node-id with Node[node-id].par $= \perp$ and id := Node[node-id].orig : **sig-inj-allowed**(node-id, id)

`cons-invariant`

return `true` **iff**

(a) $\forall$node-id **s.t.** Node[node-id].par $\neq \perp$: Node[node-id].prop $\neq \perp \wedge$
$\forall p \in$ Node[node-id].prop : Prop[PropID[p]].par = Node[node-id].par and

(b) $\forall$id **s.t.** Ptr[id] $\neq \perp$: (id, $*$) $\in$ Node[Ptr[id]].mem and

(c) the history graph contains no cycle

FIGURE 5.17: The history graph invariants for $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$.

5.4 Static Metadata-Hiding CGKA: Construction and Security Proof

In this section, we provide a static-metadata CGKA protocol. To do so, we modify Chained CmPKE of Hashimoto et al. [Has+21b] into a protocol, which we call Chained CmPKE$^{\text{ctxt}}$, that further secures the static metadata. We then prove that Chained CmPKE$^{\text{ctxt}}$ UC-realizes the ideal functionality $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$, where the leakage functions are defined to only leak the minimal static metadata matching our intuition. This results in the first CGKA that provably secures the 2nd layer.

5.4.1 Construction of Static Metadata-Hiding CGKA Chained CmPKE$^{\text{ctxt}}$

We provide the description of Chained CmPKE$^{\text{ctxt}}$. The protocol state and the related helper method are shown in Tables 5.2 to 5.5. The main protocol is depicted in Figures 5.19 to 5.23 and 5.27, and the associated helper functions are depicted in Figures 5.24 to 5.26 and 5.28 to 5.30.

Chained CmPKE$^{\text{ctxt}}$ is almost identical to Chained CmPKE [Has+21c]. The main differences are explained below and highlighted in gray in the figures. For a detailed description of the common parts of the protocols, we refer the readers to [Has+21c].

- Chained CmPKE$^{\text{ctxt}}$ additionally generates an *encryption secret* encSecret and a *welcome secret* welcomeSecret from the joiner secret.[17] encSecret is used to encrypt the contents in the proposal and commit messages, excluding the group identifier gid, epoch, and the message type, which are necessary for message delivery (cf. `*enc-prop` and `*enc-commit` functions in Figure 5.29). welcomeSecret is used to encrypt the group information and the signature in the welcome message (cf. `*enc-welcome` function in Figure 5.30).
- Chained CmPKE$^{\text{ctxt}}$ creates a separate welcome message for each new member. In contrast, Chained CmPKE included a member-independent welcome message w_0 that is transmitted to every new

[17] An encryption secret and welcome secret are also generated in MLS [Bar+22, Table 3] used to secure the static metadata.

group member (see Footnote 10). However, this will allow the server to infer that the recipients with the same w_0 will join a (possibly unknown) group.

- In Chained CmPKE$^{\text{ctxt}}$, parties explicitly sort received proposals. In contrast, Chained CmPKE and MLS's TreeKEM implicitly assume that when parties fetch the proposals, the server sorts the proposals according to predetermined rules[18]. However, since proposals are encrypted in Chained CmPKE$^{\text{ctxt}}$, the server can no longer sort them. Therefore, we make parties sort fetched proposals via `*dec-and-sort-proposals` shown in Figure 5.29 before they commit or process the proposals.
- (Optional for hiding the dynamic metadata) Chained CmPKE$^{\text{ctxt}}$ additionally generates an *metadata secret* metaKey from the joiner secret. This is used when hiding the *dynamic* metadata (see Section 5.5 for more detail).

TABLE 5.2: The protocol state of Chained CmPKE$^{\text{ctxt}}$. The additional component from [Has+21c] are highlighted in gray.

G.gid	The identifier of the group.
G.epoch	The current epoch number.
G.confTransHash	The confirmed transcript hash.
G.confTransHash-w.o-'id_c'	The confirmed transcript hash without the committer identity.
G.interimTransHash	The interim transcript hash for the next epoch.
G.member$[*]$	A mapping associating party id with its state.
G.memberHash	A hash of the public part of G.member$[*]$.
G.certSvks$[*]$	A mapping associating the set of validated signature verification keys to each party.
G.pendUpd$[*]$	A mapping associating the secret keys for each pending update proposal issued by id.
G.pendCom$[*]$	A mapping associating the new group state for each pending commit issued by id.
G.id	The identity of the party.
G.ssk	The current signing key.
G.appSecret	The current epoch's shared key.
G.membKey	The key used to MAC proposal packages.
G.encSecret	The key used to encrypt proposal and commit messages.
G.metaKey	The key used to hide the dynamic metadata in a higher level protocol.
G.initSecret	The next epoch's init secret.

[18] MLS stipulates that proposals are to be applied in the order Update, Remove, Add [Bar+22, Sec. 13.2.2].

TABLE 5.3: The party id's state stored in $G.\mathsf{member}[\mathsf{id}]$ and helper method.

id	The identity of the party.
ek	The encryption key of a CmPKE scheme.
dk	The corresponding decryption key.
svk	The signature verification key of a signature scheme.
sig	The signature for $(\mathsf{id}, \mathsf{ek}, \mathsf{svk})$ under the signature singing key corresponding to svk.
$\mathsf{kp}()$	Returns $(\mathsf{id}, \mathsf{ek}, \mathsf{svk}, \mathsf{sig})$ (if $G.\mathsf{member}[\mathsf{id}] \neq \perp$).

TABLE 5.4: The helper methods on the protocol state. The additional method from [Has+21c] are highlighted in gray.

$G.\mathsf{clone}()$	Returns (independent) copy of G.
$G.\mathsf{memberIDs}()$	Returns the list of party ids sorted by dictionary order.
$G.\mathsf{memberIDsvks}()$	Returns the list of party ids and its associating svk sorted by dictionary order in the ids.
$G.\mathsf{memberPublicInfo}()$	Returns the public part of $G.\mathsf{member}[*]$.
$G.\mathsf{groupCont}()$	Returns $(G.\mathsf{gid}, G.\mathsf{epoch}, G.\mathsf{memberHash}, G.\mathsf{confTransHash})$.
$G.\mathsf{indexOf}(\mathsf{id})$	Returns the index of id in the sorted member list returned by $G.\mathsf{memberIDs}()$.

TABLE 5.5: The protocol state maintained only during the proof. The additional component from [Has+21c] are highlighted in gray.

$G.\mathsf{joinerSecret}$	The current epoch's joiner secret.
$G.\mathsf{comSecret}$	The current epoch's commit secret.
$G.\mathsf{confKey}$	The key used to MAC for commit and welcome messages.
$G.\mathsf{welcomeSecret}$	The key used to encrypt welcome messages.
$G.\mathsf{confTag}$	The MAC tag included either in the commit or welcome message.
$G.\mathsf{membTags}$	The set of MAC tags included in the proposal messages.

$\mathsf{genSSK}()$

1 : $(\mathsf{svk}, \mathsf{ssk}) \leftarrow \mathsf{SIG.KeyGen}(\mathsf{pp}_{\mathsf{SIG}})$
2 : **return** $(\mathsf{svk}, \mathsf{ssk})$

$\mathsf{genKP}(\mathsf{id}, \mathsf{svk}, \mathsf{ssk})$

1 : $(\mathsf{ek}, \mathsf{dk}) \leftarrow \mathsf{CmGen}(\mathsf{pp}_{\mathsf{CmPKE}})$
2 : $\mathsf{sig} \leftarrow \mathsf{SIG.Sign}(\mathsf{pp}_{\mathsf{SIG}}, \mathsf{ssk}, (\mathsf{id}, \mathsf{ek}, \mathsf{svk}))$
3 : $\mathsf{kp} \leftarrow (\mathsf{id}, \mathsf{ek}, \mathsf{svk}, \mathsf{sig})$
4 : **return** $(\mathsf{kp}, \mathsf{dk})$

FIGURE 5.18: Key generation algorithms of Chained $\mathsf{CmPKE}^{\mathsf{ctxt}}$.

Input $(\texttt{Create}, \mathsf{svk})$

1 : **req** $G = \bot \wedge \mathsf{id} = \mathsf{id}_{\mathsf{creator}}$
2 : $G.\mathsf{gid} \leftarrow\!\!\$\ \{0,1\}^\kappa$; $G.\mathsf{joinerSecret} \leftarrow\!\!\$\ \{0,1\}^\kappa$
3 : $G.\mathsf{epoch} \leftarrow 0$
4 : $G.\mathsf{member}[*] \leftarrow \bot$; $G.\mathsf{memberHash} \leftarrow \bot$
5 : $G.\mathsf{confTransHash\text{-}w.o\text{-}`id_c'} \leftarrow \bot$
6 : $G.\mathsf{confTransHash} \leftarrow \bot$
7 : $G.\mathsf{certSvks}[*] \leftarrow \emptyset$
8 : $G.\mathsf{pendUpd}[*] \leftarrow \bot$; $G.\mathsf{pendCom}[*] \leftarrow \bot$
9 : $G.\mathsf{id} \leftarrow \mathsf{id}$
10 : **try** $\mathsf{ssk} \leftarrow \texttt{*fetch-ssk-if-nec}(G, \mathsf{svk})$
11 : $(\mathsf{kp}, \mathsf{dk}) \leftarrow \mathsf{genKP}(\mathsf{id}, \mathsf{svk}, \mathsf{ssk})$
12 : $G \leftarrow \texttt{*assign-kp}(G, \mathsf{id}, \mathsf{kp})$
13 : $G.\mathsf{member}[\mathsf{id}].\mathsf{dk} \leftarrow \mathsf{dk}$
14 : $G.\mathsf{ssk} \leftarrow \mathsf{ssk}$
15 : $G.\mathsf{memberHash} \leftarrow \texttt{*derive-member-hash}(G)$
16 : $(G, \mathsf{confKey}) \leftarrow \texttt{*derive-epoch-keys}(G, G.\mathsf{joinerSecret})$
17 : $\mathsf{confTag} \leftarrow \texttt{*gen-conf-tag}(G, \mathsf{confKey})$
18 : $G \leftarrow \texttt{*set-interim-trans-hash}(G, \mathsf{confTag})$

Input $(\texttt{Propose}, \text{`upd'}, \mathsf{svk})$

1 : **req** $G \neq \bot$
2 : **try** $\mathsf{ssk} \leftarrow \texttt{*fetch-ssk-if-nec}(G, \mathsf{svk})$
3 : $(\mathsf{kp}, \mathsf{dk}) \leftarrow \mathsf{genKP}(\mathsf{id}, \mathsf{svk}, \mathsf{ssk})$
4 : $P \leftarrow (\text{`upd'}, \mathsf{kp})$
5 : $p \leftarrow \texttt{*frame-prop}(G, P)$
6 : $G.\mathsf{pendUpd}[p] \leftarrow (\mathsf{ssk}, \mathsf{dk})$
7 : $p^{\mathsf{ctxt}} \leftarrow \texttt{*enc-prop}(G.\mathsf{encSecret}, p)$
8 : **return** p^{ctxt}

Input $(\texttt{Propose}, \text{`add'}, \mathsf{id}_t, \mathsf{kp}_t)$

1 : **req** $G \neq \bot \wedge \mathsf{id}_t \notin G.\mathsf{memberIDs}()$
2 : **req** $\mathsf{kp}_t \neq \bot$
3 : Send $(\texttt{has-kp}, \mathsf{id}_t, \mathsf{kp}_t)$ to $\mathcal{F}_{\mathrm{KS}}$ and receive *ack*
4 : // *ack* = `true` implies $\texttt{*validate-kp}(\mathsf{kp}_t, \mathsf{id}_t) = \texttt{true}$
5 : **req** *ack*
6 : $P \leftarrow (\text{`add'}, \mathsf{kp}_t)$
7 : $p \leftarrow \texttt{*frame-prop}(G, P)$
8 : $p^{\mathsf{ctxt}} \leftarrow \texttt{*enc-prop}(G.\mathsf{encSecret}, p)$
9 : **return** p^{ctxt}

Input $(\texttt{Propose}, \text{`rem'}, \mathsf{id}_t)$

1 : **req** $G \neq \bot \wedge \mathsf{id}_t \in G.\mathsf{memberIDs}()$
2 : $P \leftarrow (\text{`rem'}, \mathsf{id}_t)$
3 : $p \leftarrow \texttt{*frame-prop}(G, P)$
4 : $p^{\mathsf{ctxt}} \leftarrow \texttt{*enc-prop}(G.\mathsf{encSecret}, p)$
5 : **return** p^{ctxt}

FIGURE 5.19: Static metadata-hiding CGKA protocol Chained $\mathsf{CmPKE}^{\mathsf{ctxt}}$: `Create` and `Propose`. The major changes from [Has+21c] are highlighted in gray.

Input $(\texttt{Commit}, \vec{p}^{\,\mathsf{ctxt}}, \mathsf{svk})$

1 : **req** $G \neq \bot$
2 : **try** $\vec{p} \leftarrow$ `*dec-and-sort-proposals`$(G.\mathsf{encSecret}, \vec{p}^{\,\mathsf{ctxt}})$
3 : $G' \leftarrow$ `*init-epoch`(G)
4 : **try** $(G', upd, rem, add) \leftarrow$ `*apply-props`$(G, G', \vec{p})$
5 : **req** $(*, \text{`rem'-id}) \notin rem \wedge (\mathsf{id}, *) \notin upd$
6 : $\mathsf{addedMem} \leftarrow \{\, \mathsf{id}_t \mid (*, \text{`add'-id}_t\text{-}*) \in add \,\}$ // Recipients of the welcome message
7 : $\mathsf{receivers} \leftarrow G'.\mathsf{memberIDs}() \setminus \mathsf{addedMem}$ // Recipients of the new commit secret
8 : **try** $(G', \mathsf{comSecret}, \mathsf{kp}, \mathsf{ct}_0, \vec{\mathsf{ct}} = (\widehat{\mathsf{ct}}_{\mathsf{id}})_{\mathsf{id} \in \mathsf{receivers}}) \leftarrow$ `*rekey`$(G', \mathsf{receivers}, \mathsf{id}, \mathsf{svk})$
9 : $G' \leftarrow$ `*set-member-hash`(G')
10 : $\mathsf{propIDs} \leftarrow ()$
11 : **foreach** $p \in \vec{p}$ **do** $\mathsf{propIDs} \mathrel{+}\!\!\leftarrow \mathsf{H}(p)$
12 : $C_0 \leftarrow (\mathsf{propIDs}, \mathsf{kp}, \mathsf{ct}_0)$
13 : $\mathsf{sig} \leftarrow$ `*sign-commit`(G, C_0)
14 : $G' \leftarrow$ `*set-conf-trans-hash`$(G, G', \mathsf{id}, C_0, \mathsf{sig})$
15 : $(G', \mathsf{confKey}, \mathsf{joinerSecret}) \leftarrow$ `*derive-keys`$(G, G', \mathsf{comSecret})$
16 : $\mathsf{confTag} \leftarrow$ `*gen-conf-tag`$(G', \mathsf{confKey})$
17 : $c_0 \leftarrow$ `*frame-commit`$(G, C_0, \mathsf{sig}, \mathsf{confTag})$
18 : $G' \leftarrow$ `*set-interim-trans-hash`$(G', \mathsf{confTag})$
19 : $\vec{c} \leftarrow \emptyset$
20 : **foreach** $\mathsf{id} \in G.\mathsf{memberIDs}()$ **do**
21 : **if** $\mathsf{id} \in \mathsf{receivers}$ **then** $\vec{c} \mathrel{+}\!\!\leftarrow (\mathsf{id}, \widehat{\mathsf{ct}}_{\mathsf{id}})$)
22 : **else** $\vec{c} \mathrel{+}\!\!\leftarrow \widehat{c}_{\mathsf{id}} := (\mathsf{id}, \bot)$)
23 : **if** $add \neq ()$ **then**
24 : $(G', w_0, \vec{w}) \leftarrow$ `*welcome-msg`$(G', \mathsf{addedMem}, \mathsf{joinerSecret}, \mathsf{confTag})$
25 : **else**
26 : $w_0 \leftarrow \bot; \vec{w} \leftarrow \emptyset$
27 : $G.\mathsf{pendCom}[c_0] \leftarrow (G', \vec{p}, upd, rem, add)$
28 : // Encrypt messages
29 : $(c_0^{\mathsf{ctxt}}, \vec{c}^{\,\mathsf{ctxt}}) \leftarrow$ `*enc-commit`$(G.c_0, \vec{c})$
30 : $\vec{w}^{\,\mathsf{ctxt}} \leftarrow \emptyset$
31 : $\mathsf{welcomeSecret} \leftarrow \mathsf{HKDF.Expand}(\mathsf{joinerSecret}, \text{`wel'})$
32 : **foreach** $\widehat{w} \in \vec{w}$ **then**
33 : $\widehat{w}^{\mathsf{ctxt}} \leftarrow$ `*enc-welcome`$(\mathsf{welcomeSecret}, w_0, \widehat{w})$
34 : $\vec{w}^{\,\mathsf{ctxt}} \mathrel{+}\!\!\leftarrow \widehat{w}^{\mathsf{ctxt}}$
35 : **return** $(c_0^{\mathsf{ctxt}}, \vec{c}^{\,\mathsf{ctxt}}, \vec{w}^{\,\mathsf{ctxt}})$

FIGURE 5.20: Static metadata-hiding CGKA protocol Chained $\mathsf{CmPKE}^{\mathsf{ctxt}}$: `Commit`. The major changes from [Has+21c] are highlighted in gray.

Input $(\texttt{Process}, c_0^{\text{ctxt}}, \widehat{c}^{\text{ctxt}}, \vec{p}^{\text{ctxt}})$

1 : **req** $G \neq \bot$
2 : **try** $(c_0, \widehat{c}) \leftarrow$ `*dec-commit`$(G.\mathsf{encSecret}, c_0^{\text{ctxt}}, \widehat{c}^{\text{ctxt}})$
3 : **try** $\vec{p} \leftarrow$ `*dec-and-sort-proposals`$(G.\mathsf{encSecret}, \vec{p}^{\text{ctxt}})$
4 : $(\mathsf{id}_c, C_0, \mathsf{sig}, \mathsf{confTag}) \leftarrow$ `*unframe-commit`(G, c_0)
5 : **if** $\mathsf{id}_c = \mathsf{id}$ **then**
6 : **parse** $(G', \vec{p}', upd, rem, add) \leftarrow G.\mathsf{pendCom}[c_0]$
7 : **req** $\vec{p} = \vec{p}'$
8 : **return** $(\mathsf{id}_c, upd \| rem \| add, G'.\mathsf{memberIDsvks}())$
9 : **parse** $(\mathsf{propIDs}, \mathsf{kp}_c, \mathsf{ct}_0) \leftarrow C_0$
10 : **parse** $(\mathsf{id}', \widehat{\mathsf{ct}}_{\mathsf{id}'}) \leftarrow \widehat{c}'$
11 : **req** $G.\mathsf{id} = \mathsf{id}'$
12 : **for** $i \in 1, \ldots, |\vec{p}|$ **do**
13 : **req** $\mathsf{H}(\vec{p}[i]) = \mathsf{propIDs}[i]$
14 : $G' \leftarrow$ `*init-epoch`(G)
15 : **try** $(G', upd, rem, add) \leftarrow$ `*apply-props`$(G, G', \vec{p})$
16 : **req** $(*, \mathsf{id}_c) \notin rem \wedge (\mathsf{id}_c, *) \notin upd$
17 : **if** $(*, \text{'rem'-id}) \in rem$ **then**
18 : $G' \leftarrow \bot$
19 : **return** $(\mathsf{id}_c, upd \| rem \| add, \bot)$
20 : **else**
21 : $G' \leftarrow$ `*set-conf-trans-hash`$(G, G', \mathsf{id}_c, C_0, \mathsf{sig})$
22 : $(G', \mathsf{comSecret}) \leftarrow$ `*apply-rekey`$(G', \mathsf{id}_c, \mathsf{kp}_c, \mathsf{ct}_0, \widehat{\mathsf{ct}}_{\mathsf{id}})$
23 : $G' \leftarrow$ `*set-member-hash`(G')
24 : $(G', \mathsf{confKey}, \mathsf{joinerSecret}) \leftarrow$ `*derive-keys`$(G, G', \mathsf{comSecret})$
25 : **req** `*vrf-conf-tag`$(G', \mathsf{confKey}, \mathsf{confTag})$
26 : $G' \leftarrow$ `*set-interim-trans-hash`$(G', \mathsf{confTag})$
27 : **return** $(\mathsf{id}_c, upd \| rem \| add, G'.\mathsf{memberIDsvks}())$

FIGURE 5.21: Static metadata-hiding CGKA protocol Chained $\mathsf{CmPKE}^{\text{ctxt}}$: `Process`. The major changes from [Has+21c] are highlighted in gray.

Input $(\texttt{Join}, \widehat{w}^{\mathsf{ctxt}})$

req $G = \bot$
$(w_0, \widehat{w}) \leftarrow$ `*dec-welcome`$(\widehat{w}^{\mathsf{ctxt}})$
parse $(\mathsf{ct}_0, \mathsf{groupInfo}, \mathsf{sig}) \leftarrow w_0$
parse $(\mathsf{id}', \mathsf{kphash}, \widehat{\mathsf{ct}}_{\mathsf{id}'}) \leftarrow \widehat{w}$
req $\mathsf{id} = \mathsf{id}'$
try $(G, \mathsf{confTag}, \mathsf{id}_c) \leftarrow$ `*initialize-group`$(G, \mathsf{id}, \mathsf{groupInfo})$
req $G.\mathsf{confTransHash} = \mathsf{H}(G.\mathsf{confTransHash}\text{-w.o-`}\mathsf{id}_c\text{'}, \mathsf{id}_c)$
req $G.\mathsf{interimTransHash} = \mathsf{H}(G.\mathsf{confTransHash}, \mathsf{confTag})$
req $\mathsf{SIG.Verify}(G.\mathsf{member}[\mathsf{id}_c].\mathsf{svk}, \mathsf{sig}, (\mathsf{ct}_0, \widehat{\mathsf{ct}}, \mathsf{groupInfo}))$
try $G \leftarrow$ `*vrf-group-state`(G)
$G.\mathsf{id} \leftarrow \mathsf{id}$
$\mathsf{svk} \leftarrow G.\mathsf{member}[\mathsf{id}].\mathsf{svk}$
Send $(\texttt{get-ssk}, \mathsf{svk})$ to $\mathcal{F}_{\mathrm{AS}}$ and receive ssk
$G.\mathsf{ssk} \leftarrow \mathsf{ssk}$
Send $(\texttt{get-dks})$ to $\mathcal{F}_{\mathrm{KS}}$ and receive kbs
$\mathsf{joinerSecret} \leftarrow \bot$
foreach $(\mathsf{kp}, \mathsf{dk}) \in \mathsf{kbs}$ **do**
  if $\mathsf{H}(\mathsf{kp}) = \mathsf{kphash}$ **then**
    req $G.\mathsf{member}[\mathsf{id}].\mathsf{kp}() = \mathsf{kp}$
    $G.\mathsf{member}[\mathsf{id}].\mathsf{dk} \leftarrow \mathsf{dk}$
    $\mathsf{joinerSecret} \leftarrow \mathsf{CmDec}(\mathsf{dk}, \mathsf{ct}_0, \widehat{\mathsf{ct}})$
req $\mathsf{joinerSecret} \neq \bot$
$(G, \mathsf{confKey}) \leftarrow$ `*derive-epoch-keys`$(G, \mathsf{joinerSecret})$
req `*vrf-conf-tag`$(G, \mathsf{confKey}, \mathsf{confTag})$
return $(\mathsf{id}_c, G.\mathsf{memberIDsvks}())$

FIGURE 5.22: Static metadata-hiding CGKA protocol Chained $\mathsf{CmPKE}^{\mathsf{ctxt}}$: `Join`. The major changes from [Has+21c] are highlighted in gray.

Input $(\texttt{Key})$

req $G \neq \bot$
$\mathsf{k} \leftarrow G.\mathsf{appSecret}$
return k

Input $(\texttt{Key}_{\mathtt{mh}}, c_0)$

req $G \neq \bot$
$\mathsf{k}_{\mathsf{mh}} \leftarrow G.\mathsf{metaKey}$
return k_{mh}

Input $(\texttt{NextKey}_{\mathtt{mh}}, c_0)$

// Returns the pending $\mathsf{k}_{\mathsf{mh}}'$
// for the next epoch
parse $(G', *) \leftarrow G.\mathsf{pendCom}[c_0]$
$\mathsf{k}_{\mathsf{mh}}' \leftarrow G'.\mathsf{metaKey}$
return $\mathsf{k}_{\mathsf{mh}}'$

FIGURE 5.23: Static metadata-hiding CGKA protocol Chained $\mathsf{CmPKE}^{\mathsf{ctxt}}$: Retrieve group secret key. The major changes from [Has+21c] are highlighted in gray. Note that $\texttt{Key}_{\mathtt{mh}}$ and $\texttt{NextKey}_{\mathtt{mh}}$ are used in the higher-level metadata-hiding protocol.

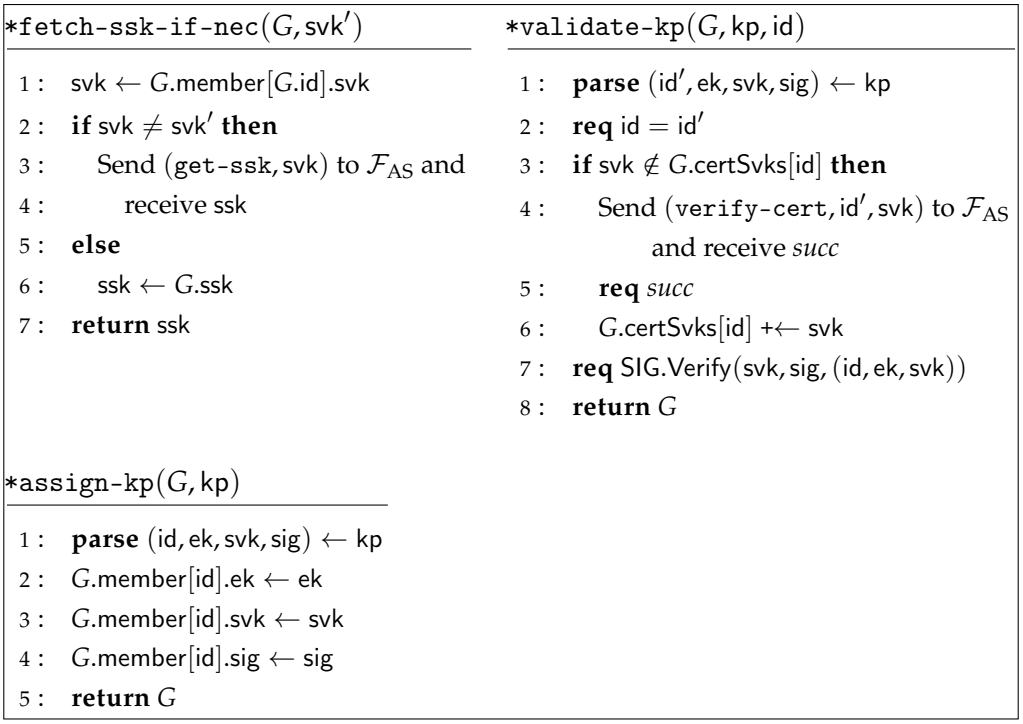


FIGURE 5.24: Helper functions of Chained CmPKE^{ctxt}: Key material related.

```
*init-epoch(G)
1 : G' ← G.clone()
2 : G'.epoch ← G.epoch + 1
3 : G'.pendUpd[*], G'.pendCom[*] ← ⊥
4 : return G'

*rekey(G', receivers, id, svk)
1 : try ssk ← *fetch-ssk-if-nec(G', svk)
2 : (kp, dk) ← genKP(id, svk, ssk)
3 : G' ← *assign-kp(G', kp)
4 : G'.ssk ← ssk
5 : G'.member[id].dk ← dk
6 : comSecret ←$ {0,1}^κ
7 : ek⃗ ← (G.member[id'].ek)_{id'∈receivers}
8 : (ct_0, ct⃗) ← CmEnc(pp_CmPKE, ek⃗, comSecret)
9 : return (G', comSecret, kp, ct_0, ct⃗)

*apply-rekey(G', id_c, kp_c, ct_0, ĉt)
1 : dk ← G'.member[G'.id].dk
2 : comSecret ← CmDec(dk, ct_0, ĉt)
3 : try G' ← *validate-kp(G', kp_c, id_c)
4 : G' ← *assign-kp(G', kp_c)
5 : return (G', comSecret)
```

```
*apply-props(G, G', p⃗)
1 : upd, rem, add ← ()
2 : foreach p ∈ p⃗ do
3 :   try (id_s, P) ← *unframe-prop(G, p)
4 :   parse (type, val) ← P
5 :   if type = 'upd' then
6 :     req id_s ∈ G.memberIDs()
7 :     req (id_s, *) ∉ upd ∧ rem = () ∧ add = ()
8 :     try G' ← *validate-kp(G', val, id_s)
9 :     G' ← *assign-kp(G', val)
10 :    if id_s = G.id then
11 :      parse (ssk, dk) ← G.pendUpd[p]
12 :      G'.ssk ← ssk
13 :      G'.member[G.id].dk ← dk
14 :    svk ← G'.member[id_s].svk
15 :    upd +← (id_s, 'upd'-svk)
16 :  elseif type = 'rem' then
17 :    parse id_t ← val
18 :    req id_t ≠ id_s ∧ id_t ∈ G.memberIDs()
19 :    req (id_t, *) ∉ upd ∧ add = ()
20 :    G'.member[id_t] ← ⊥
21 :    rem +← (id_s, 'rem'-id_t)
22 :  elseif type = 'add' then
23 :    (id_t, *, *, *) ← val
24 :    req id_t ∉ G.memberIDs()
25 :    try G' ← *validate-kp(G', val, id_t)
26 :    G' ← *assign-kp(G', val)
27 :    add +← (id_s, 'add'-val)
28 :  else
29 :    return ⊥
30 : return (G', upd, rem, add)
```

FIGURE 5.25: Helper functions of Chained CmPKE^{ctxt}: `Commit` and `Process` related.

*welcome-msg$(G', \text{addedMem}, \text{joinerSecret}, \text{confTag})$

1 : $\vec{\text{ek}} \leftarrow (G'.\text{member}[\text{id}_t].\text{ek})_{\text{id}_t \in \text{addedMem}}$ **do**
2 : $(\text{ct}_0, \vec{\text{ct}} = (\widehat{\text{ct}}_{\text{id}_t})_{\text{id}_t \in \text{addedMem}}) \leftarrow \text{CmEnc}(\text{pp}_{\text{CmPKE}}, \vec{\text{ek}}, \text{joinerSecret})$
3 : $\text{groupInfo} \leftarrow (G'.\text{gid}, G'.\text{epoch}, G'.\text{memberPublicInfo}(), G'.\text{memberHash}, G'.\text{confTransHash-w.o-'id}_c\text{'},$
$G'.\text{confTransHash}, G'.\text{interimTransHash}, \text{confTag}, G'.\text{id})$
4 : $\text{sig} \leftarrow \text{SIG.Sign}(\text{pp}_{\text{SIG}}, G'.\text{ssk}, (\text{ct}_0, \text{groupInfo}))$
5 : $w_0 \leftarrow (\text{ct}_0, \text{groupInfo}, \text{sig})$
6 : $\vec{w} \leftarrow \emptyset$
7 : **foreach** $\text{id}_t \in \text{addedMem}$ **do**
8 : $\text{kphash}_t \leftarrow \text{H}(G'.\text{member}[\text{id}_t].\text{kp}())$
9 : $\vec{w} \mathrel{+}\!\leftarrow \widehat{w}_{\text{id}_t} = (\text{id}_t, \text{kphash}_t, \widehat{\text{ct}}_{\text{id}_t})$
10 : **return** $(G', w_0, \vec{w})$

*initialize-group$(G, \text{id}, \text{groupInfo})$

1 : **parse** $(\text{gid}, \text{epoch}, \text{member}, \text{memberHash}, \text{confTransHash-w.o-'id}_c\text{'}, \text{confTransHash}, \text{interimTransHash}, \text{confTag}, \text{id}_c)$
$\leftarrow \text{groupInfo}$
2 : $(G.\text{gid}, G.\text{epoch}, G.\text{member}, G.\text{memberHash}, G.\text{confTransHash-w.o-'id}_c\text{'}, G.\text{confTransHash}, G.\text{interimTransHash})$
$\leftarrow (\text{gid}, \text{epoch}, \text{member}, \text{memberHash}, \text{confTransHash-w.o-'id}_c\text{'}, \text{confTransHash}, \text{interimTransHash})$
3 : $G.\text{certSvks}[*] \leftarrow \emptyset$
4 : $G.\text{pendUpd}[*] \leftarrow \perp; G.\text{pendCom}[*] \leftarrow \perp$
5 : $G.\text{id} \leftarrow \text{id}$
6 : **return** $(G, \text{confTag}, \text{id}_c)$

*vrf-group-state(G)

1 : **req** $G.\text{memberHash} =$ *derive-member-hash(G)
2 : $\text{mem} \leftarrow G.\text{memberIDs}()$
3 : **foreach** $\text{id} \in \text{mem}$ **do**
4 : $\text{kp} \leftarrow G.\text{member}[\text{id}].\text{kp}()$
5 : **try** $G \leftarrow$ *validate-kp$(G, \text{kp}, \text{id})$
6 : **return** G

FIGURE 5.26: Helper functions of Chained CmPKE$^{\text{ctxt}}$: Join related.

*gen-conf-tag(G, confKey)

1 : confTag ← MAC.Gen(confKey, G.confTransHash)
2 : **return** confTag

*vrf-conf-tag(G, confKey, confTag)

1 : $\succ$ ← MAC.Verify(confKey, confTag, G.confTransHash)
2 : **return** $\succ$

*set-member-hash (G)

1 : G.memberHash ← *derive-member-hash(G)
2 : **return** G

*derive-member-hash (G)

1 : // mem is sorted by dictionary order
2 : KP ← (); mem ← G.memberIDs()
3 : **foreach** id ∈ mem **do**
4 : KP +← G.member[id].kp()
5 : **return** H(KP)

*set-conf-trans-hash($G, G', \mathsf{id}_c, C_0$, sig)

1 : comCont ← (G.gid, G.epoch, 'commit', C_0, sig)
2 : G'.confTransHash-w.o-'id_c' ← H(G.interimTransHash, comCont)
3 : G'.confTransHash ← H(G'.confTransHash-w.o-'id_c', id_c)
4 : **return** G'

*set-interim-trans-hash(G', confTag)

1 : G'.interimTransHash ← H(G'.confTransHash, confTag)
2 : **return** G'

*derive-keys(G, G', comSecret)

1 : s ← HKDF.Extract(G.initSecret, comSecret)
2 : joinerSecret ← HKDF.Expand(s, 'joi')
3 : (G', confKey) ← *derive-epoch-keys(G', joinerSecret)
4 : **return** (G', confKey, joinerSecret)

*derive-epoch-keys(G', joinerSecret)

1 : confKey ← HKDF.Expand(joinerSecret, G'.groupCont()‖'conf')
2 : G'.appSecret ← HKDF.Expand(joinerSecret, G'.groupCont()‖'app')
3 : G'.membKey ← HKDF.Expand(joinerSecret, G'.groupCont()‖'memb')
4 : G'.encSecret ← HKDF.Expand(joinerSecret, G'.groupCont()‖'enc')
5 : G'.metaKey ← HKDF.Expand(joinerSecret, G'.groupCont()‖'meta')
6 : G'.initSecret ← HKDF.Expand(joinerSecret, G'.groupCont()‖'init')
7 : **return** (G', confKey)

FIGURE 5.27: Helper functions of Chained CmPKE$^{\mathsf{ctxt}}$: computing tags, hash values, group secrets and framing and unframing packets. The major changes from [Has+21c] are highlighted in gray.

*frame-prop(G, P)

1 : propCont $\leftarrow$ $(G.\mathsf{groupCont}(), G.\mathsf{id}, \text{'proposal'}, P)$
2 : sig $\leftarrow$ $\mathsf{SIG.Sign}(\mathsf{pp}_{\mathsf{SIG}}, G.\mathsf{ssk}, \mathsf{propCont})$
3 : membTag $\leftarrow$ $\mathsf{MAC.Gen}(G.\mathsf{membKey}, (\mathsf{propCont}, \mathsf{sig}))$
4 : $\overline{p} \leftarrow (G.\mathsf{id}, P, \mathsf{sig}, \mathsf{membTag})$
5 : **return** $(G.\mathsf{gid}, G.\mathsf{epoch}, \text{'proposal'}, \overline{p})$

*unframe-prop(G, p)

1 : **parse** $(\mathsf{gid}, \mathsf{epoch}, \mathsf{contType}, \overline{p}) \leftarrow p$
2 : **parse** $(\mathsf{id}_s, P, \mathsf{sig}, \mathsf{membTag}) \leftarrow \overline{p}$
3 : **req** $\mathsf{contType} = \text{'proposal'}$
$\wedge\ \mathsf{gid} = G.\mathsf{gid} \wedge \mathsf{epoch} = G.\mathsf{epoch}$
4 : propCont $\leftarrow$ $(G.\mathsf{groupCont}(), \mathsf{id}_s, \text{'proposal'}, P)$
5 : svk $\leftarrow$ $G.\mathsf{member}[\mathsf{id}_s].\mathsf{svk}$
6 : **req** $G.\mathsf{member}[\mathsf{id}_s] \neq \bot$
$\wedge\ \mathsf{SIG.Verify}(\mathsf{svk}, \mathsf{sig}, \mathsf{propCont})$
$\wedge\ \mathsf{MAC.Verify}(G.\mathsf{membKey}, \mathsf{membTag}, (\mathsf{propCont}, \mathsf{sig}))$
7 : **return** (id_s, P)

*sign-commit(G, C_0)

1 : comCont $\leftarrow$ $(G.\mathsf{groupCont}(), G.\mathsf{id}, \text{'commit'}, C_0)$
2 : sig $\leftarrow$ $\mathsf{SIG.Sign}(\mathsf{pp}_{\mathsf{SIG}}, G.\mathsf{ssk}, \mathsf{comCont})$
3 : **return** sig

*frame-commit$(G, C_0, \mathsf{sig}, \mathsf{confTag})$

1 : $\overline{c}_0 \leftarrow (G.\mathsf{id}, C_0, \mathsf{sig}, \mathsf{confTag})$
2 : **return** $(G.\mathsf{gid}, G.\mathsf{epoch}, \text{'commit'}, \overline{c}_0)$

*unframe-commit(G, c_0)

1 : **parse** $(G.\mathsf{gid}, G.\mathsf{epoch}, \text{'commit'}, \mathsf{CT}_{\overline{c}_0}) \leftarrow c_0$
2 : **parse** $(\mathsf{id}_c, C_0, \mathsf{sig}, \mathsf{confTag}) \leftarrow \overline{c}_0$
3 : **req** $\mathsf{contType} = \text{'commit'}$
$\wedge\ \mathsf{gid} = G.\mathsf{gid} \wedge \mathsf{epoch} = G.\mathsf{epoch}$
4 : comCont $\leftarrow$ $(G.\mathsf{groupCont}(), \mathsf{id}_c, \text{'commit'}, C_0)$
5 : $\mathsf{svk}_c \leftarrow G.\mathsf{member}[\mathsf{id}_c].\mathsf{svk}$
6 : **req** $G.\mathsf{member}[\mathsf{id}_c] \neq \bot$
$\wedge\ \mathsf{SIG.Verify}(\mathsf{svk}_c, \mathsf{sig}, \mathsf{comCont})$
7 : **return** $(\mathsf{id}_c, C_0, \mathsf{sig}, \mathsf{confTag})$

FIGURE 5.28: Helper functions of Chained CmPKE$^{\mathsf{ctxt}}$: Frame and unframe packets.

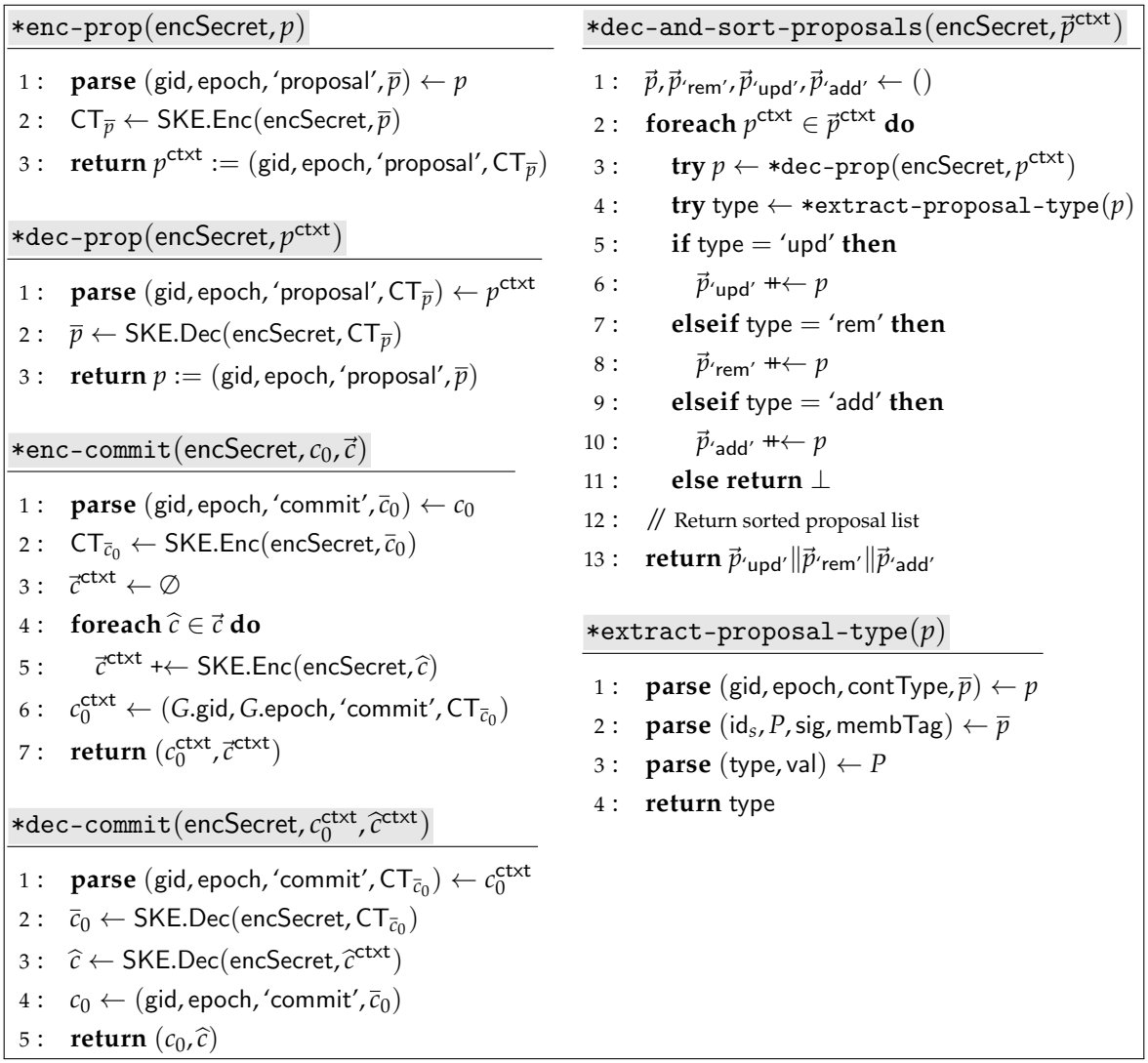


FIGURE 5.29: Helper functions unique to Chained CmPKE$^{\mathsf{ctxt}}$: Encrypt and decrypt proposals and commits. The major changes from [Has+21c] are highlighted in gray. `*sort-proposals` decrypts proposals and sort them following the order of MLS specification [Bar+22, Sec. 13.2.2].

`*enc-welcome`$(\mathsf{welcomeSecret}, w_0, \widehat{w})$

1 : **parse** $(\mathsf{id}_t, \mathsf{kphash}_t, \widehat{\mathsf{ct}}) \leftarrow w_0$
2 : **parse** $(\mathsf{ct}_0, \mathsf{groupInfo}, \mathsf{sig}) \leftarrow \widehat{w}$
3 : $\mathsf{welcomeSecret}_{\mathsf{id}_t}$
$\quad \leftarrow \mathsf{HKDF.Expand}(\mathsf{welcomeSecret}, \mathsf{id}_t)$
4 : $\mathsf{CT} \leftarrow \mathsf{SKE.Enc}(\mathsf{welcomeSecret}_{\mathsf{id}_t}, (\mathsf{groupInfo}, \mathsf{sig}))$
5 : **return** $\widehat{w}^{\mathsf{ctxt}} := (\mathsf{id}_t, (\mathsf{kphash}_t, \mathsf{ct}_0, \widehat{\mathsf{ct}}), \mathsf{CT})$

`*dec-welcome`$(\widehat{w}^{\mathsf{ctxt}})$

1 : **parse** $(\mathsf{id}_t, (\mathsf{kphash}, \mathsf{ct}_0, \widehat{\mathsf{ct}}), \mathsf{CT}) \leftarrow \widehat{w}^{\mathsf{ctxt}}$
2 : Send `(get-dks)` to $\mathcal{F}_{\mathrm{KS}}$ and receive kbs
3 : $\mathsf{joinerSecret}, \mathsf{kp}_{\mathsf{id}}, \mathsf{dk}_{\mathsf{id}} \leftarrow \bot$
4 : **foreach** $(\mathsf{kp}, \mathsf{dk}) \in \mathsf{kbs}$ **do**
5 : **if** $\mathsf{H}(\mathsf{kp}) = \mathsf{kphash}$ **then**
6 : $(\mathsf{kp}_{\mathsf{id}}, \mathsf{dk}_{\mathsf{id}}) \leftarrow (\mathsf{kp}, \mathsf{dk})$
7 : $\mathsf{joinerSecret} \leftarrow \mathsf{CmDec}(\mathsf{dk}, \mathsf{ct}_0, \widehat{\mathsf{ct}})$
8 : **break**
9 : **req** $\mathsf{joinerSecret} \neq \bot$
10 : $\mathsf{welcomeSecret}$
$\quad \leftarrow \mathsf{HKDF.Expand}(\mathsf{joinerSecret}, \text{`wel'})$
11 : $\mathsf{welcomeSecret}_{\mathsf{id}_t}$
$\quad \leftarrow \mathsf{HKDF.Expand}(\mathsf{welcomeSecret}, \mathsf{id}_t)$
12 : $(\mathsf{groupInfo}, \mathsf{sig})$
$\quad \leftarrow \mathsf{SKE.Dec}(\mathsf{welcomeSecret}_{\mathsf{id}_t}, \mathsf{CT})$
13 : $w_0 \leftarrow (\mathsf{ct}_0, \mathsf{groupInfo}, \mathsf{sig})$
14 : $\widehat{w} \leftarrow (\mathsf{id}_t, \mathsf{kphash}_t, \widehat{\mathsf{ct}})$
15 : **return** $(w_0, \widehat{w})$

FIGURE 5.30: Helper functions unique to Chained $\mathsf{CmPKE}^{\mathsf{ctxt}}$: Encrypt and decrypt welcome messages. The major changes from [Has+21c] are highlighted in gray.

5.4.2 Security Proof of Static Metadata-Hiding CGKA

Hashimoto et al. [Has+21b] proposed Chained CmPKE and showed that it UC-realizes $\mathcal{F}_{\text{CGKA}}$. We show that the ciphertext variant of Chained CmPKE, which we call Chained CmPKE$^{\text{ctxt}}$, UC-realizes $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$. The construction itself is a variation mirroring what MLSCiphertext does to MLSPlaintext. Thus, the technical novelty of this section is the first formal security proof that models static metadata-hiding. Considering that the overall structure of Chained CmPKE is similar to TreeKEM, except that the former further supports selective downloading, we expect our proof techniques to naturally translate to the ciphertext variant of TreeKEM. We leave it as an important future work to formally validate this. Below, we provide the core insights of our security proof. The full detail of the proof is provided in Section 5.4.4.

Recycling previous proofs. As explained earlier, the ideal functionality $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ *almost* degenerates to $\mathcal{F}_{\text{CGKA}}$ when the leakage functions are defined to leak all static metadata. Our proof takes advantage of this fact. Recall that to prove security in the UC framework, we informally need to construct a simulator $\mathcal{S}$ that simulates the real world adversary $\mathcal{A}$. We thus try to construct a simulator $\mathcal{S}^{\text{ctxt}}$ for $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ that internally executes $\mathcal{S}$ for $\mathcal{F}_{\text{CGKA}}$. Using the description of $\mathcal{S}$ provided in Hashimoto et al. [Has+21b], the hope is to recycle all the proofs provided by them (which spanned 30 pages!). At a high level, during the hybrids where the leakage functions do not hide any static metadata, $\mathcal{S}^{\text{ctxt}}$ can use knowledge of the group encryption key to encrypt whatever non-encrypted proposals and commits $\mathcal{S}$ outputs, and decrypt whatever $\mathcal{S}$ inputs to perform a process. This allows us to recycle all the proofs that ignore the security of the static metadata. We then gradually modify the definition of the leakage functions to arrive at our ideal functionality $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$. In the final hybrid, $\mathcal{S}^{\text{ctxt}}$ no longer knows the group encryption key for those epochs where **safe** is true, and $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ (roughly) takes care of generating random encryption of a proposal and commit.

While the high-level idea of recycling the proof of Hashimoto et al. [Has+21b] sounds straightforward, it turns out to be easier said than done. The non-triviality comes from the fact that the history graph created by $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ while using the null-leakage function is *not* identical to those created by $\mathcal{F}_{\text{CGKA}}$ — it is only *almost* identical. As explained earlier, our new history graph uses the semantics to identify the nodes and captures settings unique to $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$. Since all the security arguments boil down to how the predicate **safe** is defined over the history graph constructed during the security proof, it is not clear how to relate the proof of Hashimoto et al. to ours. To this end, we define the notion of *isomorphisms* of history graphs so that a security proof translates within the same class of history graphs. At a high level, two history graphs are isomorphism if the symbolic representation of the group evolution is identical. We prove that the two history graphs created in $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ and $\mathcal{F}_{\text{CGKA}}$ are isomorphic.

Polynomial security loss. We like to point out that our security proof only admits a *polynomial* security loss. This is in contrast to Hashimoto et al. [Has+21b] that admitted an *exponential* security loss.[19]

The main reason for this disparity is because Chained CmPKE$^{\text{ctxt}}$ captures a larger part of a CGKA compared to Chained CmPKE. As explained earlier, Chained CmPKE does not use the group secret key to encrypt proposals and commits, while Chained CmPKE$^{\text{ctxt}}$ does. Effectively, in the former, an adversary was able to *adaptively* corrupt the group secret key while not trivially winning the security game. However, once the group secret key is used in a higher-level protocol as in Chained CmPKE$^{\text{ctxt}}$, this is no longer the case. An adversary capable of corrupting the group secret key *after* seeing the ciphertext can trivially win the security game.[20] Since the exponential security loss in Hashimoto et al. [Has+21b] was due to

[19] We note that they were able to achieve a polynomial security loss relying on a stronger form of multi-recipient PKE secure against adaptive corruptions.

[20] We note that this problem may theoretically be solved using *non-committing* encryptions [Can+96], but we did not consider them as a viable option as it will add a noticeable overhead in efficiency.

the adaptivity of the adversary, we can remove this loss by naturally restricting such adversaries in our UC-security model.

Remark 5.4.1 (Adversary-controlled randomness). In case of corruption, [AJM22; Has+21b] allow the adversary $\mathcal{A}$ in $\mathcal{F}_{\text{CGKA}}$ to set the randomness to be used by the corrupted member id by altering the value $\mathsf{Rand}[\mathsf{id}]$. While our ideal functionality $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ models such strong adversaries, we restrict the adversary in our security proof to never alter the randomness for simplicity and better readability. Recent definitions [Alw+22d; Alw+21a] also intentionally disregard this type of attack.[21] We leave it as future work to incorporate adversary-controlled randomness into the proof.

5.4.3 Safety Predicates and Leakage Functions for Chained CmPKE$^{\text{ctxt}}$

As explained in Section 5.3.3, to formally prove that Chained CmPKE$^{\text{ctxt}}$ UC-realizes the ideal functionality $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$, we must first specify the safety predicates **safe**, **mac-inj-allowed**, and **sig-inj-allowed**, and the leakage functions ***leak-create**, ***leak-prop**, ***leak-com**, ***leak-wel**, and ***leak-proc**.

These are formally defined in Figures 5.31 and 5.32. The definition of the safety predicates **safe**, **mac-inj-allowed** and **sig-inj-allowed** is identical to those considered by Chained CmPKE [Has+21c, Fig. 28], modulo the syntactical difference in the definition of node pointers (see Section 5.3.1).

The leakage functions ***leak-create**, ***leak-prop**, ***leak-com**, ***leak-wel**, and ***leak-proc** are new to the definition of $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ and dictate the amount of static metadata leaked to the server. Since the concrete leakage information depends on the concrete choice of CGKA, our choice only captures the leakage of Chained CmPKE$^{\text{ctxt}}$. Namely, other protocols such as the ciphertext variant of TreeKEM used in MLSCiphertext may require a slightly more complex leakage function. We provide some discussion in Section 5.8. Below, we explain the specific choice of our leakage functions in more detail.

***leak-create**$(\mathsf{id}, \mathsf{svk})$: This defines the information leaked when a new group is created. In the previous (non-metadata-hiding) Chained CmPKE, the simulator was given the identity of the party id creating the group and its corresponding signature verification key svk to simulate the group identifier gid. In other words, gid leaked information on $(\mathsf{id}, \mathsf{svk})$. In Chained CmPKE$^{\text{ctxt}}$, the simulator is asked to simulate gid without giving any other information. This models the fact that when a new group is created, the server cannot guess who created it. As mentioned in the introduction, this could potentially open the door to a DoS attack on the server. One way to solve this would be to use standard anonymous credentials [Cha82]. We leave such direction of research as an important future work.

***leak-prop**$(\mathsf{id}, \mathsf{act})$: This defines the information that is leaked from proposals p. In Chained CmPKE, the simulator was given the identity of the party id creating the proposal, the commit node $\mathsf{Ptr}[\mathsf{id}]$ that locates id in the history graph, and the action act of the proposal to simulate the proposal p. Here, when there is no fork in the main group, then $\mathsf{Ptr}[\mathsf{id}]$ essentially pinpoints the specific epoch of the group. In case there is a fork, then epoch alone is not sufficient to identify where party id is located, in which case we need the exact proposal node $\mathsf{Ptr}[\mathsf{id}]$. In other words, proposal p leaked the proposing party id and the action act, where note that p must always include $\mathsf{Ptr}[\mathsf{id}]$ (which is $(\mathsf{gid}, \mathsf{epoch})$ essentially) for the server to deliver p to the appropriate member in a specific epoch.

In Chained CmPKE$^{\text{ctxt}}$, we define ***leak-prop**$(\mathsf{id}, \mathsf{act})$ to only output $\mathsf{Ptr}[\mathsf{id}]$ and the length of $|\mathsf{id}|$ and $|\mathsf{act}|$. This models the fact that proposals p only leak the specific group by which p is supposed to be processed. Moreover, in case the size of each action act is different, this leaks the action type. However, considering that the addition or removal of a group member is noticeable from the server (since it changes the size of commit messages), hiding the type of action may not add as much security as one expects.

[21] Alwen et al.[Alw+21a] does not allow adversaries to manipulate randomness used for symmetric key encryption.

***leak-com**$(\mathsf{id}, \vec{p}, \mathsf{svk})$: This defines the information that is leaked from commits $(c_0, \vec{c} = (\widehat{c}_{\mathsf{id}'})_{\mathsf{id}'})$. In Chained CmPKE, the simulator was given the identity of the party id creating the commit; the commit node Ptr[id] that locates id in the history graph; the list of proposals $\vec{p}$ that is going to be committed; the new signing key svk of id; and the list of current members mem in the group to simulate the commit $(c_0, \vec{c})$. In other words, a commit leaks the committer identity id, its signing key svk, and the current list of members mem.

In Chained CmPKE$^{\mathsf{ctxt}}$, we define ***leak-com**$(\mathsf{id}, \vec{p}, \mathsf{svk})$ to only output $(\mathsf{Ptr}[\mathsf{id}], |\mathsf{id}|, \vec{p}, |\mathsf{svk}|, |\mathsf{mem}|)$. Note that similarly to proposals, Ptr[id] cannot be hidden. Moreover, $\vec{p}$ is guaranteed to hide the static metadata from above, we can provide this to the simulator. Finally, the size of the current member |mem| would necessarily leak from commit messages when using Chained CmPKE since $\vec{c}$ scales linearly with the group size. While we could apply some padding to hide the group size from the commit, the server can infer the group size when selective downloading is performed; if N distinct $\widehat{c}$ were downloaded out of $M \geq N$ commits (that includes the padded garbage commits), then the server can guess that the group size was N.

***leak-wel**$(\mathsf{node\text{-}id}_{\mathsf{cur}}, \mathsf{node\text{-}id}_{\mathsf{next}}, \mathsf{id}_t)$: This defines the information that is leaked from welcome messages $\widehat{w}$. In Chained CmPKE, the simulator was given the key package kp_t of the added member id_t; the commit node of the next epoch $\mathsf{node\text{-}id}_{\mathsf{next}}$; and all the information stored on the commit node $\mathsf{Node}[\mathsf{node\text{-}id}_{\mathsf{next}}]$ to simulate the welcome message $\widehat{w}$. In other words, welcome messages leak the group identifier gid, the next epoch, the identity id of the party who created the welcome message, the size of the member of the next epoch, and so on.

In Chained CmPKE$^{\mathsf{ctxt}}$, we define ***leak-wel**$(\mathsf{node\text{-}id}_{\mathsf{cur}}, \mathsf{node\text{-}id}_{\mathsf{next}}, \mathsf{id}_t)$ to only output $(\mathsf{kp}_t, |\mathsf{gid}|, |\mathsf{epoch}|, |\mathsf{id}|, |\mathsf{mem}|)$. Here, we include kp_t in clear since Chained CmPKE$^{\mathsf{ctxt}}$ (and MLSCiphertext) includes the hash of the key package of id_t in the welcome message in the clear. Namely, in case id_t creates many key packages, the simulator must know which kp_t is used to simulate the welcome message.

We note that the reason why the proposed protocol and MLSCiphertext includes a hash kphash_t of kp_t in the welcome message is for efficiency. The added party, which may have created many key packages for different groups, can check which key package to use by simply finding the key package that equals the hash value kphash_t. Otherwise, the party must go through all the decryption keys associated with each key package it has and try to see whether the welcome message can be decrypted correctly. We also note that id_t is always leaked from the welcome message since otherwise, the server will not know the destination of the welcome message, i.e., it cannot deliver it.

***leak-proc**(id): This defines the information that is leaked from a party-dependent commit $(c_0, \widehat{c})$ and a list of proposals $\vec{p}$. In Chained CmPKE, the simulator was given the identity of the party id processing the commit and the commit node Ptr[id] that locates id in the history graph to decide whether $(c_0, \widehat{c}, \vec{p})$ should be processed correctly by a party id.

In Chained CmPKE$^{\mathsf{ctxt}}$, we define ***leak-proc**(id) to only output $(\mathsf{Ptr}[\mathsf{id}], \mathsf{index}_{\mathsf{id}})$. As explained above, Ptr[id] cannot be hidden since this information is required for id to retrieve the uploaded commit and proposals from the server. Moreover, since selective downloading is performed, $\widehat{c}$ must leak the position of the party id in the group (assuming that id can process $(c_0, \widehat{c})$ correctly).

5.4.4 Security of Chained CmPKE$^{\mathsf{ctxt}}$

The following theorem establishes that Chained CmPKE$^{\mathsf{ctxt}}$ UC-realizes the ideal functionality $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$.

Theorem 5.4.2. *Assuming that* CmPKE *is* IND-CCA *secure,* SIG *is* sEUF-CMA *secure, and* SKE *is* IND-CCA *secure, our CGKA protocol* Chained CmPKE$^{\mathsf{ctxt}}$ *adaptively and securely realizes the ideal functionality* $\mathcal{F}_{CGKA}^{\mathsf{ctxt}}$ *in the* $(\mathcal{F}_{AS}, \mathcal{F}_{KS}, \mathcal{G}_{RO})$*-hybrid model.*

Knowledge of party's secrets.

know(node-id, id) $\iff$
(a) id $\in$ Node[node-id].exp$\vee$
(b) ***secrets-injected**(node-id, id)$\vee$
(c) (Node[node-id].par $\neq \perp \wedge$ **know**(Node[node-id].par, id)) $\wedge \neg$***secrets-replaced**(node-id, id)$\vee$
(d) $\exists$node-id$'$: (Node[node-id$'$].par = node-id $\wedge$ **know**(node-id$'$, id) $\wedge \neg$***secrets-replaced**(node-id$'$, id))
***secrets-injected**(node-id, id) $\iff$
(a) (Node[node-id].orig = id $\wedge$ Node[node-id].stat $\neq$ 'good')$\vee$
(b) $\exists p \in$ Node[node-id].prop with prop-id := PropID$[p]$:
(Prop[prop-id].act = 'upd'-$*\wedge$Prop[prop-id].orig = id $\wedge$ Prop[prop-id].stat $\neq$ 'good')$\vee$
(c) $\exists p \in$ Node[node-id].prop with prop-id := PropID$[p]$: (Prop[prop-id].act = 'add'-id-$*$-svk-$*\wedge$svk $\in$ ExposedSvk)
***secrets-replaced**(node-id, id) $\iff$
Node[node-id].orig = id$\vee$
$\exists p \in$ Node[node-id].prop with prop-id := PropID$[p]$: Prop[prop-id].act $\in$ { 'add'-id-$*$,'rem'-id } $\vee$
$\exists p \in$ Node[node-id].prop with prop-id := PropID$[p]$: (Prop[prop-id].act = 'upd'-$*\wedge$Prop[prop-id].orig = id)

Knowledge of epoch secrets.

know(node-id, 'epoch') $\iff$ Node[node-id].exp $\neq \emptyset \vee$ ***can-traverse**(node-id)
***can-traverse**(node-id) $\iff$
(a) $\exists p \in$ Node[node-id].prop : (Prop[PropID$[p]$].act = 'add'-id-$*$-svk-$*\wedge$svk $\in$ ExposedSvk)
(b) ***reused-welcome-key-leaks**(node-id)$\vee$
(c) Node[node-id].stat = 'bad' $\wedge \exists p \in$ Node[node-id].prop : Prop[PropID$[p]$].act = 'add'-$*\vee$
(d) (Node[node-id].par $= \perp \vee$ **know**(Node[node-id].par, 'epoch'))$\wedge$
$\exists$(id, $*$) $\in$ Node[node-id].mem : **know**(node-id, id)
***reused-welcome-key-leaks**(node-id) $\iff$
$\exists$id, $p \in$ Node[node-id].prop : Prop$[p]$.act = 'add'-id-$*\wedge$
$\exists$node-id$_d$: node-id$_d$ is a descendant of node-id $\wedge$ id $\in$ Node[node-id$_d$].exp$\wedge$
no node node-id$_h$ exists on node-id–node-id$_d$ path s.t. ***secrets-replaced**(node-id$_h$, id) = `true`

Safe and can-inject.

safe(node-id) $\iff$ **know**(node-id, 'epoch')
sig-inj-allowed(node-id, id) $\iff$ Node[node-id].mem[id] $\in$ ExposedSvk
mac-inj-allowed(node-id) $\iff$ **know**(node-id, 'epoch')

FIGURE 5.31: The safety predicate for Chained CmPKE$^{\text{ctxt}}$. This is identical to [Has+21c, Fig. 28] modulo the syntactical difference in the definition of node pointers.

***leak-create**(id, svk)

1 : **if** $\texttt{flag}_{\mathsf{contHide}}$ **then**
2 : **return** $\perp$
3 : **else**
4 : **return** (id, svk)

***leak-prop**(id, act)

1 : **if** $\texttt{flag}_{\mathsf{contHide}} \wedge$ **safe**(Ptr[id]) **then**
2 : **return** (Ptr[id], |id| , |act|)
3 : **else**
4 : **return** (Ptr[id], id, act)

***leak-com**(id, $\vec{p}$, svk)

1 : mem $\leftarrow$ Node[Ptr[id]].mem
2 : **if** $\texttt{flag}_{\mathsf{contHide}} \wedge$ **safe**(Ptr[id]) **then**
3 : **return** (Ptr[id], |id| , $\vec{p}$, |svk| , |mem|)
4 : **else**
5 : **return** (Ptr[id], id, $\vec{p}$, svk, mem)

***leak-proc**(id)

1 : $\mathsf{index}_{\mathsf{id}} \leftarrow$ Node[Ptr[id]].indexOf(id)
2 : **if** $\texttt{flag}_{\mathsf{contHide}} \wedge$ **safe**(Ptr[id]) **then**
3 : **return** (Ptr[id], $\mathsf{index}_{\mathsf{id}}$)
4 : **else**
5 : **return** (Ptr[id], id)

***leak-wel**($\mathsf{node\text{-}id}_{\mathsf{cur}}$, $\mathsf{node\text{-}id}_{\mathsf{next}}$, id_t)

1 : gid $\leftarrow$ Node[$\mathsf{node\text{-}id}_{\mathsf{next}}$].gid
2 : epoch $\leftarrow$ Node[$\mathsf{node\text{-}id}_{\mathsf{next}}$].epoch
3 : $\mathsf{id}_c \leftarrow$ Node[$\mathsf{node\text{-}id}_{\mathsf{next}}$].orig
4 : mem $\leftarrow$ Node[$\mathsf{node\text{-}id}_{\mathsf{next}}$].mem
5 : **if** $\texttt{flag}_{\mathsf{contHide}} \wedge$ **safe**($\mathsf{node\text{-}id}_{\mathsf{next}}$) **then**
6 : **return** (kp_t, |gid| , |epoch| , $|\mathsf{id}_c|$, |mem|)
7 : **else**
8 : **return** (kp_t, $\mathsf{node\text{-}id}_{\mathsf{next}}$, Node[$\mathsf{node\text{-}id}_{\mathsf{next}}$])

FIGURE 5.32: Leakage functions for Chained $\mathsf{CmPKE}^{\mathsf{ctxt}}$.

Here, $\mathcal{F}^{\mathsf{ctxt}}_{CGKA}$ is defined with respect to the safety predicates and leakage functions in Figures 5.31 and 5.32. Moreover, calls to the hash function H, HKDF, *and* MAC *are replaced by calls to the global random oracle $\mathcal{G}_{RO}$.*

Proof Overview. The proof consists of a sequence of hybrids where we gradually modify the protocol Chained CmPKE$^{\mathsf{ctxt}}$ into the ideal functionality $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$. As explained in Section 5.4.2, with some work, we can reuse a great part of the proof by Hashimoto et al. [Has+21b].

The key observation is that when the leakage functions ***leak-create**, ***leak-prop**, ***leak-com**, ***leak-wel** and ***leak-proc** are defined to leak all the static metadata (i.e., $\mathtt{flag}_{\mathsf{contHide}} = \mathtt{false}$), then $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ is *almost* identical to the ideal functionality $\mathcal{F}_{\mathrm{CGKA}}$ that does not hide the static metadata. The only difference is that the proposals and commits in $\mathcal{F}_{\mathrm{CGKA}}$ leak the static metadata, while $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ encrypts them. To this end, using the fact that the simulator $\mathcal{S}$ knows the encryption key (which follows from $\mathtt{flag}_{\mathsf{contHide}} = \mathtt{false}$), the simulator $\mathcal{S}$ for $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ will internally run the simulator $\widetilde{\mathcal{S}}$ for $\mathcal{F}_{\mathrm{CGKA}}$ defined by Hashimoto et al. [Has+21b].

In a bit more detail, our proof goes through the same Hybrids 1 to 6 as in [Has+21b, Theorem E.1]. At Hybrid 6, the application keys for the nodes with **safe** $=$ `true` will become indistinguishable from random — this is the place where [Has+21b, Theorem E.1] ends. We then continue to complete the proof by further adding a new Hybrid 7, where $\mathtt{flag}_{\mathsf{contHide}}$ is switched to `true`. This hybrid is part of our security proof that takes care of the static metadata. In each Hybrid i for $i \in [6]$, we define the simulator $\mathcal{S}_i$ that internally runs $\widetilde{\mathcal{S}}_i$ defined in [Has+21b, Theorem E.1]. Informally, whenever $\widetilde{\mathcal{S}}_i$ takes as input a (non-static metadata-hiding) proposal, commit, or a welcome message, $\mathcal{S}_i$ first decrypts the (static metadata-hiding) proposal, commit, and welcome message before feeding it into $\widetilde{\mathcal{S}}_i$. On the other hand, if $\widetilde{\mathcal{S}}_i$ outputs a (non-static metadata-hiding) content, then $\mathcal{S}_i$ encrypts them before outputting them to $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ or the environment $\mathcal{Z}$. Here, since $\mathtt{flag}_{\mathsf{contHide}} = \mathtt{false}$, we can assume $\mathcal{S}_i$ knows all the encryption/decryption key to perform the above procedure.

While the high-level idea of piggybacking on the prior proof of Hashimoto et al. [Has+21b] sounds straightforward, several technical issues make the above non-trivial. This non-triviality is caused by the difference of the *semantics* of the nodes in the history graph in our new ideal functionality $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ and the prior ideal functionality $\mathcal{F}_{\mathrm{CGKA}}$. For instance, when the environment $\mathcal{Z}$ invokes id on input $(\mathtt{Commit}, \vec{p}, \mathsf{svk})$, the simulator in [Has+21b] (roughly) outputs $(ack, rt, c_0, \vec{c})$, where $rt \in \mathbb{N}$. Our simulator must reinterpret this and output an appropriate node-id such that the history graph defined with respect to c_0 remains consistent with the history graph defined with respect to node-id. In particular, the most non-trivial part of reusing the proof by Hashimoto et al. [Has+21b] is to check that the proof moves from Hybrid 2 to 3 in [Has+21b, Theorem E.1], which concerns the consistency of the history graph, translates to our setting.

We also note that while the proof of Hashimoto et al. [Has+21b] required an CmPKE that is IND-CCA secure *with adaptive corruptions*, we only require a standard CmPKE. As explained in Section 5.4.2, this is due to the added restriction on the adversary (see Figure 5.11 and Section 5.3.3), which is necessary for any natural static metadata-hiding CGKA in order not to trivially win the security game. Specifically, the exponential loss appearing in the proof of moving from Hybrid 5 to Hybrid 6 in [Has+21b] disappears by considering the restricted adversary defined in $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$.

Finally, we prove the indistinguishability of the real and ideal protocols assuming that the adversary *cannot* inject bad randomness. That is, Rand is always set to 'good' in the ideal functionality $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$. To be more precise, our proof up till Hybrid 6 allows the adversary to inject bad randomness. We only restrict the adversary when moving to Hybrid 7 — this is where we explicitly use the group secret to perform symmetric key encryption to hide the static metadata. This restriction is also done in recent work by Alwen et al. [Alw+21a], where they disallow the adversary to manipulate the randomness used for the symmetric key encryption. We leave security analysis of such types of attacks as future work.

Proof of Theorem 5.4.2. We now provide the full proof of Theorem 5.4.2.

Proof. We consider the following sequence of hybrids. While the environment $\mathcal{Z}$ interacts with Chained $\mathsf{CmPKE}^{\mathsf{ctxt}}$ in Hybrid 1, it interacts with the ideal functionality $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ in Hybrid 7. As explained above, Hybrids 1 to 6 are defined identically to [Has+21b], where the only difference is in the definition of the simulator $\mathcal{S}$ in each hybrid. Below, we first define all the hybrids and then explain how the simulators are defined.

Hybrid 1. This is the real world, where we make a syntactic change. We consider a simulator $\mathcal{S}_1$ that interacts with a dummy functionality $\mathcal{F}_{\mathrm{dummy}}$. $\mathcal{F}_{\mathrm{dummy}}$ sits between the environment $\mathcal{Z}$ and $\mathcal{S}_1$, and relays any message from $\mathcal{Z}$ to $\mathcal{S}_1$ without modifying them. $\mathcal{S}_1$ internally simulates the real-world parties and adversary $\mathcal{A}$ by using the messages sent from $\mathcal{F}_{\mathrm{dummy}}$. From $\mathcal{A}$'s point of view, $\mathcal{S}_1$ is the environment $\mathcal{Z}$.

Hybrid 2. This hybrid concerns the authentication service and key service. We replace $(\mathcal{F}_{\mathrm{AS}}, \mathcal{F}_{\mathrm{KS}})$ with these ideal version $(\mathcal{F}_{\mathrm{AS}}^{\mathrm{IW}}, \mathcal{F}_{\mathrm{KS}}^{\mathrm{IW}})$. Since these functions are not accessible by $\mathcal{Z}$, this modification is undetectable by $\mathcal{Z}$. Thus, the view of $\mathcal{Z}$ in Hybrid 1 and Hybrid 2 are identical.

Hybrid 3. This hybrid concerns the correctness and consistency guarantees. We replace $\mathcal{F}_{\mathrm{dummy}}$ with a variant of $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$, denoted as $\mathcal{F}_{\mathrm{CGKA},3}^{\mathsf{ctxt}}$, where **safe** (resp. **sig-inj-allowed** and **mac-inj-allowed**) always returns false (resp. true) and $\mathtt{flag}_{\mathsf{contHide}}$ is set to false. In other words, all application secrets are set by the simulator, injections are always allowed, and the confidentiality of messages is not considered. The simulator $\mathcal{S}_3$ is identical to $\mathcal{S}_2$.

Hybrid 4. This hybrid concerns the security of the signature scheme. We modify $\mathcal{F}_{\mathrm{CGKA},3}^{\mathsf{ctxt}}$ to use the original **sig-inj-allowed** predicate, denoted as $\mathcal{F}_{\mathrm{CGKA},4}^{\mathsf{ctxt}}$. $\mathcal{F}_{\mathrm{CGKA},4}^{\mathsf{ctxt}}$ halts if a proposal, a commit, or a welcome message is injected when the sender's signing key is not exposed. The simulator $\mathcal{S}_4$ is identical to $\mathcal{S}_3$.

Hybrid 5. This hybrid concerns the security of the MAC scheme. We modify $\mathcal{F}_{\mathrm{CGKA},4}^{\mathsf{ctxt}}$ to use the original **mac-inj-allowed** predicate, denoted as $\mathcal{F}_{\mathrm{CGKA},5}^{\mathsf{ctxt}}$. $\mathcal{F}_{\mathrm{CGKA},5}^{\mathsf{ctxt}}$ halts if a proposal or a commit are injected when the corresponding MAC key is not exposed. The simulator $\mathcal{S}_5$ is identical to $\mathcal{S}_4$.

Hybrid 6. This hybrid concerns the confidentiality of the application secrets. We modify $\mathcal{F}_{\mathrm{CGKA},5}^{\mathsf{ctxt}}$ where it uses the original **safe** predicate, denoted as $\mathcal{F}_{\mathrm{CGKA},6}^{\mathsf{ctxt}}$. The simulator $\mathcal{S}_6$ is identical to $\mathcal{S}_5$ except that it sets only those application secrets for which **safe** is false. (Note that this functionality roughly corresponds to the previous ideal functionality $\mathcal{F}_{\mathrm{CGKA}}$ where the security of the static metadata is not considered.)

Hybrid 7. This hybrid concerns the confidentiality of proposal, commit and welcome messages. We modify $\mathcal{F}_{\mathrm{CGKA},6}^{\mathsf{ctxt}}$ so that $\mathtt{flag}_{\mathsf{contHide}}$ is set to true, denoted as $\mathcal{F}_{\mathrm{CGKA},7}^{\mathsf{ctxt}}$. The simulator $\mathcal{S}_7$ is identical to $\mathcal{S}_6$ except that it simulates the protocol executions with the leakage information defined by ***leak-create**, ***leak-prop**, ***leak-com**, ***leak-wel**, and ***leak-proc**. The functionality $\mathcal{F}_{\mathrm{CGKA},7}^{\mathsf{ctxt}}$ corresponds to the ideal functionality $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$.

We show the indistinguishability of Hybrids 2 to 7 in Lemmata 5.4.3 and 5.4.8. This completes the proof of the main theorem. □

From Hybrid 2 to 6: Lemma 5.4.3. We first prove the indistinguishability of hybrid 2 to Hybrid 6.

Lemma 5.4.3. *Hybrid 2 and Hybrid 6 are indistinguishable assuming the* IND-CCA *security of* CmPKE, sEUF-CMA *security of* SIG, *and the key-committing property of* SKE.

Moreover, we assume the adversary can inject bad randomness throughout these hybrids, i.e., Rand *can be set to 'bad'.*

Proof. The proof consists of five parts. We first explain how the simulator simulates the protocols with the simulator defined in [Has+21b]; second, we explain how to prove the correspondence of the history graphs maintained by $\mathcal{F}_{\mathsf{CGKA},i}$ and $\mathcal{F}^{\mathsf{ctxt}}_{\mathsf{CGKA},i}$, which is the goal of this proof; third, we explain the detailed description of the simulator; fourth, we prove the correspondence between the two history graphs; finally, we prove the supporting propositions to finish the proof of this lemma.

Part 1: Preparation. As explained in the overview, $\mathcal{S}_i$ internally executes the simulator $\widetilde{\mathcal{S}}_i$ presented in the proof by Hashimoto et al. [Has+21b]. To be more precise, $\mathcal{S}_i$ outsources the simulation of Chained $\mathsf{CmPKE}^{\mathsf{ctxt}}$ to $\widetilde{\mathcal{S}}_i$ that simulates Chained CmPKE by appropriately modifying the inputs and outputs of $\widetilde{\mathcal{S}}_i$.

To formalize the description of $\mathcal{S}_i$, we make one modification to $\widetilde{\mathcal{S}}_i$. Without loss of generality, while $\widetilde{\mathcal{S}}_i$ is internally simulating Chained CmPKE, we assume it executes and stores the following two secrets whenever the parties are invoked on a `Commit` or a `Join`:

- $G.\mathsf{encSecret} \leftarrow \mathsf{HKDF.Expand}(\mathsf{joinerSecret}, G.\mathsf{groupCont}() \| \text{'enc'})$, and
- $\mathsf{welcomeSecret} \leftarrow \mathsf{HKDF.Expand}(\mathsf{joinerSecret}, \text{'wel'})$.

Here, the inputs to HKDF are states of the parties simulated by $\widetilde{\mathcal{S}}_i$. $\widetilde{\mathcal{S}}_i$ then outputs these stored secrets to $\mathcal{S}_i$. $\mathcal{S}_i$ uses these secrets to either encrypt proposal and commit messages output by $\widetilde{\mathcal{S}}_i$ or decrypt proposal, commit, and welcome messages sent by $\mathcal{F}^{\mathsf{ctxt}}_{\mathsf{CGKA},i}$. Note that these secret keys are never used in the original Chained CmPKE — these are encryption keys used only to secure the static metadata included in proposal, commit, and welcome messages of Chained $\mathsf{CmPKE}^{\mathsf{ctxt}}$. It can be checked that the proof in [Has+21b] still holds even if we consider this slightly modified simulator $\widetilde{\mathcal{S}}_i$ as above since encSecret and welcomeSecret leak no information on joinerSecret in the global random oracle model. We also modify $\widetilde{\mathcal{S}}_i$ so that it outputs commit query $(\texttt{Commit}, \ldots)$ and welcome query $(\texttt{Welcome}, \ldots)$ separately like $\mathcal{S}_i$. That is, $\widetilde{\mathcal{S}}_i$ first receives $(\texttt{Commit}, \mathsf{id}, \widetilde{\vec{p}}, \mathsf{svk})$ and simulates the commit $\widetilde{c}_0$; then it receives $(\texttt{Welcome}, \mathsf{kp}_t, \mathsf{Ptr}[\mathsf{id}], \widetilde{c}_0)$ and simulates the welcome message, where kp_t is the receiver's key package, $\mathsf{Ptr}[\mathsf{id}]$ is the committer's current node, and $\widetilde{c}_0$ is the new epoch associated with the welcome message. The above modification in the proof in [Has+21b] is without loss of generality since the information $\widetilde{\mathcal{S}}_i$ receives to simulate the welcome message corresponds exactly to the information required to invoke the simulated party id to output a welcome message.

Part 2: Goal of Proof. We are now ready to explain the description of $\mathcal{S}_i$ that internally executes $\widetilde{\mathcal{S}}_i$. Looking ahead, the main goal of the proof is to relate the history graph created within $\widetilde{\mathcal{S}}_i$ to those maintained by $\mathcal{F}^{\mathsf{ctxt}}_{\mathsf{CGKA},i}$. Then, since [Has+21b] proved that the history graph made within $\widetilde{\mathcal{S}}_i$ and those maintained by the ideal functionality $\mathcal{F}_{\mathsf{CGKA}}$ in Hybrid i (i.e., $\mathcal{F}_{\mathsf{CGKA},i}$) are identical, we will be able to indirectly prove that the history graph made within $\mathcal{S}_i$ and those maintained by the ideal functionality $\mathcal{F}^{\mathsf{ctxt}}_{\mathsf{CGKA},i}$ are identical as well. Indistinguishability between each adjacent hybrid then follows straightforwardly from the prior proof and the correspondence of the history graphs maintained by $\mathcal{F}_{\mathsf{CGKA},i}$ and $\mathcal{F}^{\mathsf{ctxt}}_{\mathsf{CGKA},i}$.

Below, we put tilde on top, e.g., $\widetilde{\mathsf{Ptr}}[*]$, $\widetilde{\mathsf{Node}}[*]$, $\widetilde{\mathsf{Wel}}[*]$, to denote the components created by the history graph within $\widetilde{\mathcal{S}}_i$ (which is also known to $\mathcal{S}_i$). As explained above, the history graphs created within $\widetilde{\mathcal{S}}_i$ and maintained by $\mathcal{F}^{\mathsf{ctxt}}_{\mathsf{CGKA},i}$ are identical so we may use them interchangeably. We put tilde on proposals $\widetilde{p}$ and commits $\widetilde{c}_0$ to denote that they are *non-encrypted* variants used by $\widetilde{\mathcal{S}}_i$.

Using these terminologies, we can restate our goal as to create a one-to-one correspondence between the nodes $\widetilde{\mathsf{Node}}[\widetilde{c}]$, $\widetilde{\mathsf{Prop}}[\widetilde{p}]$, and $\widetilde{\mathsf{Wel}}[\widetilde{w}]$ maintained by the history graph in $\mathcal{F}_{\mathrm{CGKA},i}$ and the nodes $\mathsf{Node}[\mathsf{node\text{-}id}]$, $\mathsf{Prop}[\mathsf{prop\text{-}id}]$, and $\mathsf{Wel}[\mathsf{node\text{-}id}]$ maintained by the history graph in $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA},i}$. We will make the meaning of "one-to-one correspondence" more clear later, but we informally mean that the graph topology and the syntactical information stored in each node are consistent with each other. We assume $\mathcal{S}_i$ maintains lists L_p and L_c that (roughly) assign unique counters to proposal and commit nodes created within $\mathcal{S}_i$, respectively. This allows us to relate which node in $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA},i}$ relates to $\mathcal{F}_{\mathrm{CGKA},i}$.

Part 3: Description of Simulator $\mathcal{S}_i$. For a detailed motivation and description of $\widetilde{\mathcal{S}}_i$, we refer the readers to the original proof by Hashimoto et al. [Has+21b]. We only use their result proving that the history graphs created within $\widetilde{\mathcal{S}}_i$ and maintained by $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA},i}$ are identical.

Below, we explain how $\mathcal{S}_i$ answers each queries made by the ideal functionality $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA},i}$ for $i \in [3:6]$, where we omit the case $i = 2$ since it is already defined (see the definiton of Hybrids 1 and 2). Here, keep in mind that for $i \in [3:6]$ we have $\mathtt{flag}_{\mathsf{contHide}} = \mathtt{false}$, i.e., the leakage functions leak all the static metadata. Since the description of $\widetilde{\mathcal{S}}_i$ in [Has+21b] is identical for each $i \in [3:6]$, we omit the subscript i below for simplicity. In other words, we only need to check that the history graph maintained by $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA},3}$ has a one-to-one correspondence with $\mathcal{F}_{\mathrm{CGKA},3}$. As a result, the indistinguishability of Hybrids 2 to 6 then inherits from [Has+21b].

(1) Create query from $\mathsf{id}_{\mathsf{creator}}$. This concerns the case when $\mathcal{Z}$ queries $(\mathtt{Create}, \mathsf{svk})$ to $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$. If $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ outputs $(\mathtt{Create}, \mathsf{id}_{\mathsf{creator}}, \mathsf{svk})$ to $\mathcal{S}$, $\mathcal{S}$ invokes $\widetilde{\mathcal{S}}$ on the same input. From $\widetilde{\mathcal{S}}$'s point of view, $\mathcal{S}$ acts identically to the ideal functionality $\mathcal{F}_{\mathrm{CGKA}}$. $\widetilde{\mathcal{S}}$ simply runs the simulated party $\mathsf{id}_{\mathsf{creator}}$ on input $(\mathtt{Create}, \mathsf{svk})$.[22] $\mathcal{S}$ then samples a random group identifier $\mathsf{gid} \leftarrow\!\!\$\ \{0,1\}^\kappa$ and outputs it to $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$. This creates the node $\mathsf{Node}[0]$ within the history graph maintained by $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$. Finally, since $\widetilde{\mathcal{S}}$ creates a root node $\widetilde{\mathsf{Node}}[\mathsf{root}_0]$, $\mathcal{S}$ records $L_c[\mathsf{root}_0] \leftarrow 0$.

(2) Propose query from id. This concerns the case when $\mathcal{Z}$ queries $(\mathtt{Propose}, \mathsf{act})$ for some $\mathsf{act} \in \{\text{'upd'-svk}, \text{'add'-id}_t\text{-kp}_t, \text{'rem'-id}_t\}$ to $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$. If $\mathsf{Ptr}[\mathsf{id}] \neq \bot$, then $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ outputs $(\mathtt{Propose}, \mathsf{node\text{-}id} := \mathsf{Ptr}[\mathsf{id}], \mathsf{id}, \mathsf{act})$ to $\mathcal{S}$, and $\mathcal{S}$ invokes $\widetilde{\mathcal{S}}$ on the same input. From $\widetilde{\mathcal{S}}$'s point of view, $\mathcal{S}$ acts identically to the ideal functionality $\mathcal{F}_{\mathrm{CGKA}}$. $\widetilde{\mathcal{S}}$ then runs the simulated party id on input $(\mathtt{Propose}, \mathsf{act})$. If $\mathsf{Rand}[\mathsf{id}] = \text{'bad'}$ and $\mathsf{act} = \text{'upd'-svk}$, it asks $\mathcal{S}$ to provide the randomness to run party id. $\mathcal{S}$ then queries this request to its own adversary $\mathcal{A}$ to receive the adversarially chosen randomness. Here, recall that randomness is only used by $\widetilde{\mathcal{S}}$ to generate an update key package kp. If party id, internally simulated by $\widetilde{\mathcal{S}}$, returns $\bot$, then $\widetilde{\mathcal{S}}$ returns $(\mathit{ack} := \mathtt{false}, \bot)$ to $\mathcal{S}$. $\mathcal{S}$ then outputs $(\mathit{ack} := \mathtt{false}, \bot, \bot)$ to $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$.[23]

Otherwise, if id returns $\widetilde{p}$, then $\widetilde{\mathcal{S}}$ returns $(\mathit{ack} := \mathtt{true}, \widetilde{p})$. $\mathcal{S}$ encrypts $\widetilde{p}$ as $p^{\mathsf{ctxt}} \leftarrow \mathtt{*enc\text{-}prop}(G.\mathsf{encSecret}, \widetilde{p})$ using $G.\mathsf{encSecret}$ of id (provided by $\widetilde{\mathcal{S}}$, see *Part 1*). It runs the encryption on the adversarially controlled randomness if $\mathsf{Rand}[\mathsf{id}] = \text{'bad'}$. There are two cases to consider.

Case 1. If $\widetilde{\mathsf{Prop}}[\widetilde{p}] = \bot$, i.e., it did not exist in the history graph maintained by $\widetilde{\mathcal{S}}$, then $\mathcal{S}$ outputs $(\mathit{ack} := \mathtt{true}, \bot, p^{\mathsf{ctxt}})$.

Case 2. If $\widetilde{\mathsf{Prop}}[\widetilde{p}] \neq \bot$, i.e., it exists in the history graph maintained by $\widetilde{\mathcal{S}}$, then $\mathcal{S}$ searches for any $p^{\mathsf{ctxt}\prime}$ such that $\widetilde{p} \leftarrow \mathtt{*dec\text{-}prop}(G.\mathsf{encSecret}, p^{\mathsf{ctxt}\prime})$ and $\mathsf{prop\text{-}id} = \mathsf{PropID}[p^{\mathsf{ctxt}\prime}] \neq \bot$. It then chooses any such $p^{\mathsf{ctxt}\prime}$ and outputs $(\mathit{ack} := \mathtt{true}, \mathsf{prop\text{-}id}, p^{\mathsf{ctxt}})$. By Proposition 5.4.6, Item 1, which we prove later, such $p^{\mathsf{ctxt}\prime}$ exists and $\mathsf{prop\text{-}id}$ is unique regardless of the choice of $p^{\mathsf{ctxt}\prime}$.

[22] We note that in the previous definition of $\mathcal{F}_{\mathrm{CGKA}}$, the group identifies gid was omitted. However, this can be amended to $\mathcal{F}_{\mathrm{CGKA}}$ without loss of generality.

[23] Recall that due to our modification in the definition of the key service, $\widetilde{\mathcal{S}}$ does not need to output svk_t (see Section 5.3.2).

If act = 'upd', $\widetilde{\mathcal{S}}$ outputs the updated key package kp[24]. $\mathcal{S}$ passes it to $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$.

(3) Commit query from id**.** This concerns the case when $\mathcal{Z}$ queries $(\texttt{Commit}, \vec{p}^{\,\text{ctxt}}, \mathsf{svk})$ to $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$. If $\mathsf{Ptr}[\mathsf{id}] \neq \bot$, then $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ outputs $(\texttt{Commit}, \mathsf{node\text{-}id} := \mathsf{Ptr}[\mathsf{id}], \mathsf{id}, \vec{p}^{\,\text{ctxt}}, \mathsf{svk})$ to $\mathcal{S}$. $\mathcal{S}$ then decrypts the proposals $\widetilde{\vec{p}} := \texttt{*dec-and-sort-proposals}(G.\mathsf{encSecret}, \vec{p}^{\,\text{ctxt}})$ using G.encSecret of id. If decryption fails, $\mathcal{S}$ outputs $(ack := \texttt{false}, \bot, \bot, \bot)$. Otherwise, $\mathcal{S}$ invokes $\widetilde{\mathcal{S}}$ on input $(\texttt{Commit}, \mathsf{id}, \widetilde{\vec{p}}, \mathsf{svk})$. $\widetilde{\mathcal{S}}$ then runs the simulated party id on input $(\texttt{Commit}, \widetilde{\vec{p}}, \mathsf{svk})$, where it asks $\mathcal{S}$ to provide the randomness to run party id if $\mathsf{Rand}[\mathsf{id}] =$ 'bad'. $\mathcal{S}$ then queries this request to its own adversary $\mathcal{A}$ to receive the adversarially chosen randomness. If party id returns $\bot$, then $\widetilde{\mathcal{S}}$ outputs $(ack := \texttt{false}, \bot, \bot\bot)$ to $\mathcal{S}$, and $\mathcal{S}$ outputs $(ack := \texttt{false}, \bot, \bot, \bot)$ to $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$.

Otherwise, if party id returns $(\widetilde{c}_0, \widetilde{\vec{c}}, \widetilde{w}_0, \widetilde{\vec{w}})$, $\widetilde{\mathcal{S}}$ decides what to output depending on the following checks (which are identical to the checks in [Has+21b]). We also explain what $\mathcal{S}$ does when given the output of $\widetilde{\mathcal{S}}$.

Case 1. If $\widetilde{\mathsf{Node}}[\widetilde{c}_0] = \bot$, $w_0 \neq \bot$, and if there exists some $rt' \in \mathbb{N}$ and $\widetilde{w}_0'$ such that $\widetilde{\mathsf{Wel}}[w_0'] = \mathsf{root}_{rt'}$ and $\widetilde{w}_0'$ includes the same confTag as $\widetilde{w}_0$, then $\widetilde{\mathcal{S}}$ chooses any such $(\widetilde{w}_0', rt')$ (guaranteed to exist uniquely from [Has+21b, Proposition E.6]) and returns $(ack := \texttt{true}, rt := rt' \neq \bot, \widetilde{c}_0, \widetilde{\vec{c}})$ to $\mathcal{S}$. $\mathcal{S}$ then encrypts the (non-encrypted) commit message as $(c_0^{\text{ctxt}}, \vec{c}^{\,\text{ctxt}}) \leftarrow \texttt{*enc-commit}(G.\mathsf{encSecret}, \widetilde{c}_0, \widetilde{\vec{c}})$ using G.encSecret of id (provided by $\widetilde{\mathcal{S}}$, see *Part 1*). By Proposition 5.4.7, Item 3 (which we prove later), there exists a unique node-id such that $\mathsf{Node}[\mathsf{node\text{-}id}] = \widetilde{\mathsf{Node}}[\mathsf{root}_{rt}]$. That is, the syntactical information stored on the node, e.g., gid, epoch, prop, orig, and so on, are identical. $\mathcal{S}$ then outputs $(ack := \texttt{true}, \mathsf{node\text{-}id}, c_0^{\text{ctxt}}, \vec{c}^{\,\text{ctxt}})$ to $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$. Afterwards, when $\mathcal{S}$ is invoked on $(\texttt{Welcome}, \mathsf{kp}_t, \mathsf{node\text{-}id} := \mathsf{Ptr}[\mathsf{id}], \widetilde{c}_0)$ by $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$, it invokes $\widetilde{\mathcal{S}}$ on the same input. $\widetilde{\mathcal{S}}$ then returns $(ack := \texttt{true}, \widetilde{w}_0, \widetilde{\widehat{w}})$ to $\mathcal{S}$, where $\widetilde{\widehat{w}}$ is the welcome message for added member $\mathsf{id}_t \in \mathsf{mem}$ with kp_t. Finally, $\mathcal{S}$ encrypts $(\widetilde{w}_0, \widetilde{\widehat{w}})$ as $\widehat{w}^{\text{ctxt}}$ using G.welcomeSecret of id and returns $(ack := \texttt{true}, \widehat{w}^{\text{ctxt}})$ to $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$.

Case 2. Otherwise, if $\widetilde{\mathsf{Node}}[\widetilde{c}_0] = \bot$ and either $\widetilde{w}_0 = \bot$ or there does not exist $\widetilde{w}_0'$ such that $\widetilde{\mathsf{Wel}}[\widetilde{w}_0'] = \mathsf{root}_{rt'}$ for some $rt' \in \mathbb{N}$ and $\widetilde{w}_0'$ includes the same confTag as $\widetilde{w}_0$, then $\widetilde{\mathcal{S}}$ returns $(ack := \texttt{true}, rt := \bot, \widetilde{c}_0, \widetilde{\vec{c}})$ to $\mathcal{S}$. $\mathcal{S}$ then encrypts the commit as above and returns $(ack := \texttt{true}, \bot, c_0^{\text{ctxt}}, \vec{c}^{\,\text{ctxt}})$ to $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$. The welcome message is handled similarly to Case 1.

Case 3. Finally, if $\widetilde{\mathsf{Node}}[\widetilde{c}_0] \neq \bot$, then $\widetilde{\mathcal{S}}$ returns $(ack := \texttt{true}, rt := \bot, \widetilde{c}_0, \widetilde{\vec{c}})$ to $\mathcal{S}$. $\mathcal{S}$ then searches for any $c_0^{\text{ctxt}'}$ such that $\widetilde{c}_0 \leftarrow \texttt{*dec-commit}_0(G.\mathsf{encSecret}, c_0^{\text{ctxt}'})$ and $\mathsf{node\text{-}id} = \mathsf{NodeID}[c_0^{\text{ctxt}'}] \neq \bot$ (see Proposition 5.4.7 for the definition of $\texttt{*dec-commit}_0$). It then chooses any such $c_0^{\text{ctxt}'}$ and outputs $(ack := \texttt{true}, \mathsf{node\text{-}id}, c^{\text{ctxt}}, \vec{c}^{\,\text{ctxt}})$. By Proposition 5.4.7, Item 1 (which we prove later), such $c_0^{\text{ctxt}'}$ exists and node-id is unique regardless of the choice of $c_0^{\text{ctxt}'}$.

If act = 'upd', $\widetilde{\mathcal{S}}$ outputs the committer's updated key package kp, and $\mathcal{S}$ passes it to $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ accordingly.

Finally, when $\mathcal{F}_{\text{CGKA}}$ queries $(\texttt{Propose}, \mathsf{Ptr}[\mathsf{id}], p^{\text{ctxt}})$ to $\mathcal{S}$ during the $\texttt{*fill-prop}$ check, it must first decrypt p^{ctxt} to obtain the (non-encrypted) proposal $\widetilde{p}$. We use [Has+21b, Proposition E.8] that establishes that any member at the same node $\widetilde{\mathsf{Ptr}}[\mathsf{id}]$ stores the same secret key G.encSecret. Namely, $\mathcal{S}$ retrieves the unique G.encSecret assigned to the node $\widetilde{\mathsf{Ptr}}[\mathsf{id}]$ to decrypt p^{ctxt}. If $\widetilde{\mathsf{Prop}}[\widetilde{p}] = \bot$, then $\mathcal{S}$ invokes $\widetilde{\mathcal{S}}$ on input $(\texttt{Propose}, \widetilde{p})$ and receives $(\mathsf{orig}, \mathsf{act})$[25]. It then outputs $(\bot, \mathsf{orig}, \mathsf{act})$ to $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$. Otherwise, if $\widetilde{\mathsf{Prop}}[\widetilde{p}] \neq \bot$,

[24] Recall that due to our modification in the definition of $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$, $\widetilde{\mathcal{S}}$ needs to output kp if act = 'upd'. However, this can be amended to $\widetilde{\mathcal{S}}$ without loss of generality.

[25] Without loss of generality, we assume $\widetilde{\mathcal{S}}$ returns $\mathsf{act} \in \{$ 'upd'-kp, 'add'-id_t-kp_t, 'rem'-id_t $\}$

then $\mathcal{S}$ searches for any $p^{\mathsf{ctxt}'}$ such that $\widetilde{p} \leftarrow \texttt{*dec-prop}(G.\mathsf{encSecret}, p^{\mathsf{ctxt}'})$ and $\mathsf{prop\text{-}id} = \mathsf{PropID}[p^{\mathsf{ctxt}'}] \neq \bot$. It then choses any such $p^{\mathsf{ctxt}'}$ and outputs $(ack := \texttt{true}, \mathsf{prop\text{-}id}, p^{\mathsf{ctxt}})$. By Proposition 5.4.6, Item 1, such $p^{\mathsf{ctxt}'}$ exists and prop-id is unique regardless of the choice of $p^{\mathsf{ctxt}'}$. It retrieves $(\mathsf{orig}, \mathsf{act}) \leftarrow (\widetilde{\mathsf{Prop}}[\widetilde{p}].\mathsf{orig}, \widetilde{\mathsf{Prop}}[\widetilde{p}].\mathsf{act})$ and returns $(\mathsf{prop\text{-}id}, \mathsf{orig}, \mathsf{act})$ to $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$.

(4) Process query from id**.** This concerns the case when $\mathcal{Z}$ queries $(\texttt{Process}, c_0^{\mathsf{ctxt}}, \widehat{c}^{\mathsf{ctxt}}, \vec{p}^{\mathsf{ctxt}})$ to $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$. If $\mathsf{Ptr}[\mathsf{id}] \neq \bot$, then $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ outputs $(\texttt{Process}, \mathsf{node\text{-}id}, \mathsf{id}, c_0^{\mathsf{ctxt}}, \widehat{c}^{\mathsf{ctxt}}, \vec{p}^{\mathsf{ctxt}})$ to $\mathcal{S}$. $\mathcal{S}$ then decrypts the input messages as $(\widetilde{c}_0, \widetilde{\widehat{c}}) := \texttt{*dec-commit}(G.\mathsf{encSecret}, c_0^{\mathsf{ctxt}}, \widehat{c}^{\mathsf{ctxt}})$ and $\widetilde{\vec{p}} := \texttt{*dec-and-sort-proposals}(G.\mathsf{encSecret}, \vec{p}^{\mathsf{ctxt}})$ as in the real protocol. If decryption fails, $\mathcal{S}$ outputs $(ack := \texttt{false}, \bot, \bot, \bot)$. Otherwise, $\mathcal{S}$ invokes $\widetilde{\mathcal{S}}$ on input $(\texttt{Process}, \widetilde{c}_0, \widetilde{\widehat{c}}, \widetilde{\vec{p}})$. $\widetilde{\mathcal{S}}$ then (deterministically) runs the simulated party id on input $(\texttt{Process}, \mathsf{id}, \widetilde{c}_0, \widetilde{\widehat{c}}, \widetilde{\vec{p}})$. If party id returns $\bot$, then $\widetilde{\mathcal{S}}$ outputs $(ack := \texttt{false}, \bot, \bot, \bot)$ to $\mathcal{S}$. $\mathcal{S}$ then outputs $(ack := \texttt{false}, \bot, \bot, \bot)$ to $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$.

Otherwise, if party id returns $(\mathsf{id}_c, upd \| rem \| add)$, $\widetilde{\mathcal{S}}$ decides what to output depending on the following checks (which are the checks identical to [Has+21b]). We also explain what $\mathcal{S}$ does when given the output of $\widetilde{\mathcal{S}}$. Note that `*fill-prop` queries from $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ to $\mathcal{S}$ are answered exactly as in commit queries described above.

Case 1. If $\widetilde{\mathsf{Node}}[\widetilde{c}_0] = \bot$ and if there exists $\widetilde{w}_0$ that includes the same confTag as $\widetilde{c}_0$ such that $\widetilde{\mathsf{Wel}}[\widetilde{w}_0] = \mathsf{root}_{rt'}$ for some $rt' \in \mathbb{N}$, then $\widetilde{\mathcal{S}}$ chooses any such $(\widetilde{w}_0, rt')$ (guaranteed to exist uniquely from [Has+21b, Proposition E.6]) and returns $(ack := \texttt{true}, rt := rt', \bot, \bot)$ to $\mathcal{S}$. By Proposition 5.4.7, Item 3, there exists a unique node-id such that $\mathsf{Node}[\mathsf{node\text{-}id}] = \widetilde{\mathsf{Node}}[\mathsf{root}_{rt}]$. $\mathcal{S}$ then outputs $(ack := \texttt{true}, \mathsf{node\text{-}id}, \bot, \bot)$ to $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$.

Case 2. If $\widetilde{\mathsf{Node}}[\widetilde{c}_0] = \bot$ and no such $\widetilde{w}_0$ exists, then $\widetilde{\mathcal{S}}$ retrieves the associating long-term public key svk_c of id_c (which is guaranteed to exist when process succeeds in the real protocol) and returns $(ack := \texttt{true}, \bot, \mathsf{orig}' := \mathsf{id}_c, \mathsf{svk}' := \mathsf{svk}_c)$ to $\mathcal{S}$. $\mathcal{S}$ then outputs $(ack := \texttt{true}, \bot, \mathsf{orig}', \mathsf{svk}')$ to $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$.

Case 3. Finally, if $\widetilde{\mathsf{Node}}[\widetilde{c}_0] \neq \bot$, then $\widetilde{\mathcal{S}}$ simply returns $(ack := \texttt{true}, \bot, \bot, \bot)$ to $\mathcal{S}$. $\mathcal{S}$ then searches for any $c_0^{\mathsf{ctxt}'}$ such that $\widetilde{c}_0 \leftarrow \texttt{*dec-commit}_0(G.\mathsf{encSecret}, c_0^{\mathsf{ctxt}'})$ and $\mathsf{node\text{-}id} = \mathsf{NodeID}[c_0^{\mathsf{ctxt}'}] \neq \bot$ (see Proposition 5.4.7 for the definition of $\texttt{*dec-commit}_0$). It then choses any such $c_0^{\mathsf{ctxt}'}$ and outputs $(ack := \texttt{true}, \bot, \bot, \bot)$. By Proposition 5.4.7, Item 1, such $c_0^{\mathsf{ctxt}'}$ exists and node-id is unique regardless of the choice of $c_0^{\mathsf{ctxt}'}$.

(5) Join query from id**.** This concerns the case when $\mathcal{Z}$ queries $(\texttt{Join}, \widehat{w}^{\mathsf{ctxt}})$ to $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$. If $\mathsf{Ptr}[\mathsf{id}] = \bot$, then $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ outputs $(\texttt{Join}, \mathsf{id}, \widehat{w}^{\mathsf{ctxt}})$ to $\mathcal{S}$. $\mathcal{S}$ then decrypts the input messages as $(\widetilde{w}_0, \widetilde{\widehat{w}}) := \texttt{*dec-welcome}(\widehat{w}^{\mathsf{ctxt}})$ as in the real protocol. If decryption fails, $\mathcal{S}$ outputs $(ack := \texttt{false}, \bot, \bot, \bot, \bot, \bot)$. Otherwise, $\mathcal{S}$ invokes $\widetilde{\mathcal{S}}$ on input $(\texttt{Join}, \mathsf{id}, \widetilde{w}_0, \widetilde{\widehat{w}})$. $\widetilde{\mathcal{S}}$ then (deterministically) runs the simulated party id on input $(\texttt{Join}, \widetilde{w}_0, \widetilde{\widehat{w}})$. If party id returns $\bot$, then $\widetilde{\mathcal{S}}$ outputs $(ack := \texttt{false}, \bot, \bot, \bot)$ to $\mathcal{S}$. $\mathcal{S}$ then outputs $(ack := \texttt{false}, \bot, \bot, \bot, \bot, \bot)$ to $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$.

Otherwise, if party id returns $(\mathsf{id}_c, \mathsf{mem})$, $\widetilde{\mathcal{S}}$ decides what to output depending on the following checks (which are the checks identical to [Has+21b]). We also explain what $\mathcal{S}$ does when given the output of $\widetilde{\mathcal{S}}$.

Case 1. If $\widetilde{\mathsf{Wel}}[\widetilde{w}_0] \neq \bot$, $\widetilde{\mathcal{S}}$ returns $(ack := \texttt{true}, \bot, \bot, \bot)$ to $\mathcal{S}$. $\mathcal{S}$ then returns $(ack := \texttt{true}, \bot, \bot, \bot, \bot, \bot)$ to $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$.

Case 2. Otherwise, $\widetilde{\mathcal{S}}$ checks if there exists a non-root $\widetilde{c}_0$ such that $\widetilde{\mathsf{Node}}[\widetilde{c}_0] \neq \bot$ and $\widetilde{c}_0$ includes the same confTag as the one included in w_0. If such $\widetilde{c}_0$ exists (guaranteed to be unique by [Has+21b]),

then $\widetilde{\mathcal{S}}$ returns $(ack := \texttt{true}, \widetilde{c}'_0 := \widetilde{c}_0, \bot, \bot)$ to $\mathcal{S}$. $\mathcal{S}$ then retrieves $G.\mathsf{encSecret}$ stored on the node $\widetilde{\mathsf{Node}}[\widetilde{c}_0]$ and searches for any $c_0^{\mathsf{ctxt}\prime}$ such that $\widetilde{c}_0 \leftarrow \texttt{*dec-commit}_0(G.\mathsf{encSecret}, c_0^{\mathsf{ctxt}\prime})$ and $\mathsf{node\text{-}id} = \mathsf{NodeID}[c_0^{\mathsf{ctxt}\prime}] \neq \bot$ (see Proposition 5.4.7 for the definition of $\texttt{*dec-commit}_0$). It then choses any such $c_0^{\mathsf{ctxt}\prime}$ and outputs $(ack := \texttt{true}, \mathsf{node\text{-}id}, \bot, \bot, \bot, \bot)$ to $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$.

Case 3. Otherwise, if no such $\widetilde{c}_0$ exists, then $\widetilde{\mathcal{S}}$ further checks if there exists $\widetilde{w}'_0$ such that $\widetilde{\mathsf{Wel}}[\widetilde{w}'_0] \neq \bot$ that includes the same $\mathsf{confTag}$ as the one included in $\widetilde{w}_0$. If so, $\widetilde{\mathcal{S}}$ chooses any such $\widetilde{w}'_0$ and returns $(ack := \texttt{true}, \widetilde{c}'_0 := \widetilde{\mathsf{Wel}}[\widetilde{w}'_0], \bot, \bot)$. Here, [Has+21b, Proposition E.6] establishes that the value of $\widetilde{\mathsf{Wel}}[\widetilde{w}'_0]$ is guaranteed to be the same root value, i.e., root_{rt} for some $rt \in \mathbb{N}$, for all such $\widetilde{w}'_0$. Then, by Proposition 5.4.7, Item 3, there exists a unique $\mathsf{node\text{-}id}$ such that $\mathsf{Node}[\mathsf{node\text{-}id}] = \widetilde{\mathsf{Node}}[\mathsf{root}_{rt}]$. $\mathcal{S}$ then outputs $(ack := \texttt{true}, \mathsf{node\text{-}id}, \bot, \bot, \bot, \bot)$ to $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$.

Case 4. Finally, if no such $\widetilde{c}_0$ or $\widetilde{w}'_0$ exist, then $\widetilde{\mathcal{S}}$ returns $(ack := \texttt{true}, \bot, \mathsf{orig}' := \mathsf{id}_c, \mathsf{mem}' := \mathsf{mem})$ to $\mathcal{S}$. This corresponds to the case $\widetilde{\mathsf{Wel}}[\widetilde{w}_0]$ is initialized by root_{rt} for a new $rt \in \mathbb{N}$, whose value is known to $\widetilde{\mathcal{S}}$. $\mathcal{S}$ then outputs $(ack := \texttt{true}, \bot, \mathsf{gid}', \mathsf{epoch}', \mathsf{orig}', \mathsf{mem}')$ to $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$, where gid' and epoch' are included in $\widetilde{w}'_0$ in the clear.

(6) Key query from id**.** This concerns the case when $\mathcal{Z}$ queries $\texttt{Key}$ to $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$. If $\mathsf{Ptr}[\mathsf{id}] \neq \bot$ and $\mathsf{Node}[\mathsf{Ptr}[\mathsf{id}]].\mathsf{key} = \bot$, then $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ outputs $(\texttt{Key}, \mathsf{id})$ to $\mathcal{S}$. (Recall that **safe** is always set to $\texttt{false}$ in this hybrid.) $\mathcal{S}$ invokes $\widetilde{\mathcal{S}}$ on input $(\texttt{Key}, \mathsf{id})$ and receives the group secret key k. $\mathcal{S}$ returns k to $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$.

Part 4: Correspondence Between the Two History Graphs. Before proving the supporting Propositions 5.4.6 and 5.4.7, we prove that the history graph created within $\widetilde{\mathcal{S}}$ (and hence the ideal functionality $\mathcal{F}_{\mathrm{CGKA}}$) has a "one-to-one" correspondence between the history graph maintained by $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$. To formalize this, we first define what it means for the two history graphs to be *isomorphic*.

Definition 5.4.4. *Let* HG *and* $\widetilde{\mathsf{HG}}$ *be the history graphs maintained by* $\mathcal{F}_{CGKA}$ *and* $\mathcal{F}_{CGKA}^{\mathsf{ctxt}}$*, respectively. We say the two history graphs are* isomorphic *if the following holds:*

- *Let* $S_p := \{\, \mathsf{prop\text{-}id} \,\}_{\mathsf{prop\text{-}id}}$ *and* $\widetilde{S}_p := \{\, p \,\}_p$ *be the sets of all proposal nodes in* HG *and* $\widetilde{\mathsf{HG}}$*, respectively. Then, there is a bijection* f_p *between these two sets such that* $\mathsf{Prop}[\mathsf{prop\text{-}id}]$ *and* $\widetilde{\mathsf{Prop}}[f_p(\mathsf{prop\text{-}id})]$ *store the same values (modulo the syntactical difference that* $.\mathsf{par}$ *stores* $\mathsf{node\text{-}id}$ *or* c_0/root_{rt}*).*
- *Let* $S_c := \{\, \mathsf{node\text{-}id} \,\}_{\mathsf{node\text{-}id}}$ *and* $\widetilde{S}_c := \{\, c_0 \,\}_{c_0} \cup \{\, \mathsf{root}_{rt} \,\}_{rt}$ *be the sets of all commit nodes in* HG *and* $\widetilde{\mathsf{HG}}$*, respectively. Then, there is a bijection* f_c *between these two sets such that* $\mathsf{Node}[\mathsf{node\text{-}id}]$ *and* $\widetilde{\mathsf{Node}}[f_c(\mathsf{node\text{-}id})]$ *store the same values (modulo the syntactical difference that* $.\mathsf{par}$ *stores* $\mathsf{node\text{-}id}$ *or* c_0/root_{rt} *and* $.\mathsf{prop}$ *stores the encrypted or non-encrypted proposal).*

The following lemma is the key lemma to prove Lemma 5.4.3. Since isomorphic graphs agree on the same syntactical information, this also implies that the nodes for which the predicate **safe** is true are identical. Notably, all the proofs used to move from Hybrids 2 to 6 in Hashimoto et al. [Has+21b] with respect to $\mathcal{F}_{\mathrm{CGKA}}$ carryovers to those of $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$.

Lemma 5.4.5. *The history graph* HG *maintained by* $\mathcal{F}_{CGKA}$ *and* $\widetilde{\mathsf{HG}}$ *maintained by* $\mathcal{F}_{CGKA}^{\mathsf{ctxt}}$ *are isomorphic.*

Proof of Lemma 5.4.5. We prove by induction. When the history graphs are both empty, then they are isomorphic with the two bijections f_p and f_c defined to be the zero function. Now, assume the history

graphs HG and $\widetilde{\mathsf{HG}}$ are currently isomorphic, i.e., there exists bijections f_p and f_c between the propose and commit nodes. We check that the isomorphism is maintained even after a `Propose`, a `Commit`, a `Process`, or a `Join` query is performed. We omit queries to `Create` and key as they trivially maintain the isomorphism.

Consistency of isomorphism after `Propose`**.** Let us consider *Case 1* in **(2)**, i.e., $\widetilde{\mathsf{Prop}}[\widetilde{p}] = \bot$. By Proposition 5.4.6, Item 2, we have $\mathsf{PropID}[p^{\mathsf{ctxt}}] = \bot$. Thus, both $\mathcal{F}_{\mathrm{CGKA}}$ and $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ creates a new proposal node. By assumption on the induction and how `*create-prop` is defined, we have isomorphism even after `Propose`. In particular, letting $\mathsf{propCtr}$ be the current stored by $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$, the new bijection f'_p extends the domain and range of f_p by adding $f'_p(\mathsf{propCtr}) = \widetilde{p}$.

Next, let us consider *Case 2* in **(2)**, i.e., $\widetilde{\mathsf{Prop}}[\widetilde{p}] \neq \bot$. In this case $\mathcal{F}_{\mathrm{CGKA}}$ and $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ performs a consistency check `*consistent-prop`. Since no new proposal node is created, we only need to show that the output of this consistency check is identical for both ideal functionalities. There are two cases: $\mathsf{PropID}[p^{\mathsf{ctxt}}] = \bot$ or $\mathsf{PropID}[p^{\mathsf{ctxt}}] = \mathsf{prop\text{-}id}'$ for some $\mathsf{prop\text{-}id}' \neq \bot$. In the former case, by Proposition 5.4.6, Item 1, $\mathcal{S}$ finds $p^{\mathsf{ctxt}\prime} \neq p^{\mathsf{ctxt}}$ and outputs $\mathsf{prop\text{-}id} = \mathsf{PropID}[p^{\mathsf{ctxt}\prime}]$. $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ performs the consistency check using this $\mathsf{prop\text{-}id}$. In the latter case, $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ uses the existing $\mathsf{prop\text{-}id}' = \mathsf{PropID}[p^{\mathsf{ctxt}}]$ and ignores $\mathsf{prop\text{-}id}$ output by $\mathcal{S}$. By Proposition 5.4.6, Item 1, the values of $\mathsf{prop\text{-}id}$ and $\mathsf{prop\text{-}id}'$ are identical. We thus only need to focus on $\mathsf{prop\text{-}id}'$.

Recall $\mathsf{prop\text{-}id}' = \mathsf{PropID}[p^{\mathsf{ctxt}}]$ is created either during `Propose` or `*fill-prop`. Due to the key-committing property of SKE, it must be created by a node $\widetilde{\mathsf{Ptr}}[\mathsf{id}']$ that has the same $G.\mathsf{encSecret}$ as $\widetilde{\mathsf{Ptr}}[\mathsf{id}]$. Using [Has+21b, Proposition E.8], we must have $\widetilde{\mathsf{Ptr}}[\mathsf{id}'] = \widetilde{\mathsf{Ptr}}[\mathsf{id}]$. In other words, $\mathsf{prop\text{-}id}'$ was created by the node $\mathsf{Ptr}[\mathsf{id}]$. Thus, by the assumption on the induction, the consistency check by $\mathcal{F}_{\mathrm{CGKA}}$ using $\widetilde{p}$ and those by $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ using $\mathsf{prop\text{-}id}'$ are identical.

Consistency of isomorphism after `Commit`**.** Let us consider *Case 1* in **(3)**. This is the case where $\mathcal{S}$ assigns a detached root to c_0^{ctxt}. By Proposition 5.4.7, Item 3, there exists a unique $\mathsf{node\text{-}id}$ such that $\mathsf{Node}[\mathsf{node\text{-}id}] = \widetilde{\mathsf{Node}}[\mathsf{root}_{rt}]$. Thus, both $\mathcal{F}_{\mathrm{CGKA}}$ and $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ perform the same consistency check and the attach procedure. At the end of `Commit`, the new bijection f'_c is identical to f_c except that $f'_c(\mathsf{node\text{-}id}) = \widetilde{c}_0$ rather than $f_c(\mathsf{node\text{-}id}) = \mathsf{root}_{rt}$.

Next, let us consider *Case 2* in **(3)**. This is the case where $\mathcal{S}$ assigns c_0^{ctxt} to a new node. By Proposition 5.4.6, Item 2, we have $\mathsf{PropID}[p^{\mathsf{ctxt}}] = \bot$. Thus, both $\mathcal{F}_{\mathrm{CGKA}}$ and $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ create a new commit node. By assumption on the induction and how `*create-child` is defined, we have isomorphism even after `Commit`. In particular, letting $\mathsf{nodeCtr}$ be the current stored by $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$, the new bijection f'_c extends the domain and range of f_c by adding $f'_c(\mathsf{nodeCtr}) = \widetilde{c}_0$.

Finally, let us consider *Case 3* in **(3)**. In this case $\mathcal{F}_{\mathrm{CGKA}}$ and $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ performs a consistency check `*consistent-com`. Since no new commit node is created, we only need to show that the output of this consistency check is identical for both ideal functionalities. Similarly to the argument made in `Propose`, there are two cases to consider: $\mathsf{NodeID}[c_0^{\mathsf{ctxt}}] = \bot$ or $\mathsf{NodeID}[c_0^{\mathsf{ctxt}}] = \mathsf{node\text{-}id}'$ for some $\mathsf{node\text{-}id}' \neq \bot$. In the former case, by Proposition 5.4.7, Item 1, $\mathcal{S}$ finds $c_0^{\mathsf{ctxt}\prime} \neq c^{\mathsf{ctxt}}$ and outputs $\mathsf{node\text{-}id} = \mathsf{NodeID}[c_0^{\mathsf{ctxt}\prime}]$. $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ performs the consistency check using this $\mathsf{node\text{-}id}$. In the latter case, $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ uses the existing $\mathsf{node\text{-}id}' = \mathsf{NodeID}[c_0^{\mathsf{ctxt}}]$ and ignores $\mathsf{node\text{-}id}$ output by $\mathcal{S}$. By Proposition 5.4.7, Item 1, the values of $\mathsf{node\text{-}id}$ and $\mathsf{node\text{-}id}'$ are identical. We thus only need to focus on $\mathsf{node\text{-}id}'$.

Recall $\mathsf{node\text{-}id}' = \mathsf{NodeID}[c_0^{\mathsf{ctxt}}]$ is created either during `Commit` or `Process`. Following an almost exact argument made for PropID above, we can show $\mathsf{node\text{-}id}'$ was created by the node $\mathsf{Ptr}[\mathsf{id}]$. Thus, by the assumption on the induction, the consistency check by $\mathcal{F}_{\mathrm{CGKA}}$ using $\widetilde{c}$ and those by $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ using $\mathsf{node\text{-}id}'$ are identical.

Consistency of isomorphism after `Process`**.** All three cases considered in `Process` are covered by the cases considered in `Commit`. Namely, $\mathcal{S}$ chooses to either attach a detached root to c_0^{ctxt}, create a new commit node, or attach the commit to an already existing node.

Isomorphism after `Join`**.** First, we show that if $\widetilde{\mathsf{Wel}}[\widetilde{w}_0] = \bot$, then $\mathsf{Wel}[\mathsf{id}, \widehat{w}] = \bot$. Assume $\mathsf{Wel}[\mathsf{id}, \widehat{w}] \neq \bot$. Since $\mathsf{Wel}[\mathsf{id}, \widehat{w}]$ is only created during a `Commit`, this implies that there exists $\widetilde{w}_0' \neq \widetilde{w}_0$ such that $\widetilde{\mathsf{Wel}}[\widetilde{w}_0'] \neq \bot$. This further implies that $\widehat{w}$ decrypts to two different values $\widetilde{w}_0'$ and $\widetilde{w}_0$, which contradicts the committing property of the SKE. Therefore, we have $\mathsf{Wel}[\mathsf{id}, \widehat{w}] = \bot$ when $\widetilde{\mathsf{Wel}}[\widetilde{w}_0] = \bot$.

With this in mind, *Cases 1* and *2* in **(5)** do not alter the propose or commit nodes of the history graphs maintained by both $\mathcal{F}_{\mathrm{CGKA}}$ and $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$. These cases correspond to attaching a welcome message to an existing node on the history graph. Moreover, in *Case 3* in **(5)**, as we already established that $\mathsf{Node}[\mathsf{node\text{-}id}] = \widetilde{\mathsf{Node}}[\mathsf{root}_{rt}] \neq \bot$. Therefor, this also does not alter the history graph.

The final *Case 4* in **(5)** corresponds to the case where a new detached root is created within $\mathcal{F}_{\mathrm{CGKA}}$. By Proposition 5.4.7, Item 2, $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ also creates a new detached root. Since `*create-root` is called on the same input, it is clear that the newly created history graphs remain isomorphic. Namely, the new bijection f_c' extends the domain and range of f_c by adding $f_c'(\mathsf{nodeCtr}) = rt$, where rt and $\mathsf{nodeCtr}$ are the counters used to add a new commit node in $\mathcal{F}_{\mathrm{CGKA}}$ and $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$, respectively. □

Part 5: Proving the Supporting Propositions. To finish the proof of Lemma 5.4.3, we finally prove Propositions 5.4.6 and 5.4.7. Below, we rely on [Has+21b, Proposition E.8] that establishes that any member at the same node $\widetilde{\mathsf{Ptr}}[\mathsf{id}]$ stores the same secret key $G.\mathsf{encSecret}$.

Proposition 5.4.6. *We have the following:*

1. *If* $\widetilde{\mathsf{Prop}}[\widetilde{p}] \neq \bot$*, then there exists some* p^{ctxt} *such that* $\widetilde{p} \leftarrow$ `*dec-prop`$(G.\mathsf{encSecret}, p^{\mathsf{ctxt}})$ *and* $\mathsf{PropID}[p^{\mathsf{ctxt}}] \neq \bot$*, where* $G.\mathsf{encSecret}$ *is the secret key maintained in the node* $\widetilde{\mathsf{Prop}}[\widetilde{p}]$*. Moreover, for all such* p^{ctxt}*, the value of* $\mathsf{PropID}[p^{\mathsf{ctxt}}]$ *is identical.*

2. *If* $\widetilde{\mathsf{Prop}}[\widetilde{p}] = \bot$*, then does not exist any* p^{ctxt} *and* $\mathsf{encSecret}$ *such that* $\widetilde{p} \leftarrow$ `*dec-prop`$(\mathsf{encSecret}, p^{\mathsf{ctxt}})$ *and* $\mathsf{PropID}[p^{\mathsf{ctxt}}] \neq \bot$*.*

Proof. Let us first consider Item 1. $\widetilde{\mathsf{Prop}}[\widetilde{p}]$ is only created during a `Propose` or a `*fill-prop`. If it was created during a `Propose`, then by definition such p^{ctxt} exists. If it was created during a `*fill-prop`, then $\mathcal{S}$ was given p^{ctxt} such that $\widetilde{p} \leftarrow$ `*dec-prop`$(G.\mathsf{encSecret}, p^{\mathsf{ctxt}})$. At the end of `*fill-prop`, $\mathsf{PropID}[p^{\mathsf{ctxt}}]$ is created. Moreover, due to the key-committing property of the SKE, the proposal node assigned to such proposals p^{ctxt} are identical.

Next, let us consider Item 2. Due to the key-committing property of the SKE, if such a p^{ctxt} existed, then we must have $\widetilde{\mathsf{Prop}}[\widetilde{p}] \neq \bot$. Thus, Item 2 follows by taking the contrapositive. □

Proposition 5.4.7. *Let us define the function* `*dec-commit`$_0$ *such that given* c_0^{ctxt} *and* $\mathsf{encSecret}$*, it parses* $(\mathsf{gid}, \mathsf{epoch}, \text{'commit'}, \mathsf{CT}_{\bar{c}_0}) \leftarrow c_0^{\mathsf{ctxt}}$*, runs* $\bar{c}_0 \leftarrow \mathsf{SKE.Dec}(\mathsf{encSecret}, \mathsf{CT}_{\bar{c}_0})$*, and outputs* $\widetilde{c}_0 := (\mathsf{gid}, \mathsf{epoch}, \text{'commit'}, \bar{c}_0)$*.*

Then, we have the following:

1. *If* $\widetilde{\mathsf{Node}}[\widetilde{c}_0] \neq \bot$*, then there exists some* c_0^{ctxt} *such that* $\widetilde{c}_0 \leftarrow$ `*dec-commit`$_0(G.\mathsf{encSecret}, c_0^{\mathsf{ctxt}})$ *and* $\mathsf{NodeID}[c_0^{\mathsf{ctxt}}] \neq \bot$*, where* $G.\mathsf{encSecret}$ *is the secret key maintained in the node* $\widetilde{\mathsf{Node}}[\widetilde{c}_0]$*. Moreover, for all such* c^{ctxt}*, the value of* $\mathsf{NodeID}[c_0^{\mathsf{ctxt}}]$ *is identical.*

2. *If* $\widetilde{\mathsf{Node}}[\widetilde{c}_0] = \bot$*, then there does not exist any* c_0^{ctxt} *and* encSecret *such that* $\widetilde{c}_0 \leftarrow$ $*\texttt{dec-commit}_0(\mathsf{encSecret}, c_0^{\mathsf{ctxt}})$ *and* $\mathsf{NodeID}[c_0^{\mathsf{ctxt}}] \neq \bot$.

3. *If* $\widetilde{\mathsf{Node}}[\widetilde{c}_0] = \bot$ *and there exists* $\widetilde{w}_0$ *that includes the same* confTag *as* $\widetilde{c}_0$ *such that* $\widetilde{\mathsf{Wel}}[\widetilde{w}_0] = \mathsf{root}_{rt}$ *for some* $rt \in \mathbb{N}$*, then there exists a unique* node-id *such that* $\mathsf{Node}[\mathsf{node\text{-}id}] = \widetilde{\mathsf{Node}}[\mathsf{root}_{rt}]$*. That is, the syntactical information stored on the node, e.g.,* gid, epoch, prop, orig, *and so on, are identical.*

Proof. Let us first consider Item 1. $\widetilde{\mathsf{Node}}[\widetilde{c}]$ is only created during a `Commit` or a `Process`. More concretely, $\widetilde{\mathsf{Node}}[\widetilde{c}_0]$ is only created if $\widetilde{\mathcal{S}}$ assigns a detached root to $\widetilde{c}_0$ or a new commit node during a `Commit` or a `Process`. In either cases, a corresponding c_0^{ctxt} is output by $\mathcal{S}$ and $\mathsf{NodeID}[c_0^{\mathsf{ctxt}}]$ is generated within $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$. Moreover, due to the key-committing property of the SKE, the commit node assigned to such commits c_0^{ctxt} are identical.

Next, let us consider Item 2. Due to the key-committing property of the SKE, if such a c^{ctxt} existed, then we must have $\widetilde{\mathsf{Node}}[\widetilde{c}_0] \neq \bot$. Thus, Item 2 follows by taking the contrapositive.

Finally, let us consider Item 3. Such $\widetilde{w}_0$ is generated when $\widetilde{\mathcal{S}}$ creates a new root detached root during a `Join` (corresponding to *Case 4* in **(5)**). Since $\mathcal{S}$ creates a new commit node with the same syntactical information that $\widetilde{\mathcal{S}}$ used, Item 3 follows. □

This concludes the proof of Lemma 5.4.3. □

From Hybrid 6 to 7: Lemma 5.4.8. In this section, we prove the indistinguishability of remaining hybrid 6 and hybrid 7. As explained in the overview, we assume the adversary *cannot* inject bad randomness throughout these hybrids, i.e., Rand is always 'good'.

Also, we slightly modify the description of $\mathcal{S}_6$. We assume $\mathcal{S}_6$ executes the code of $\widetilde{\mathcal{S}}_6$ by itself instead of invoking $\widetilde{\mathcal{S}}_6$.

To prove Lemma 5.4.8, we first formally define the behavior of simulator $\mathcal{S}_7$ in Hybrid 7; then we analyze the simulation provided by $\mathcal{S}_7$ provides an indistinguishable view to $\mathcal{Z}$ as in Hybrid 6.

Part 1: Description of the Simulator $\mathcal{S}_7$. We first explain the description of $\mathcal{S}_7$. For simulation, $\mathcal{S}_7$ stores group encryption key encSecret for epoch node-id in the list $L_{\mathsf{encSecret}}$.

$\mathcal{S}_7$ simulates group states G (which includes e.g., membership list and group secret keys) of node-id depending on the status of the parent epoch of node-id denoted by $\mathsf{node\text{-}id}_p$.

Case (1). If **safe**$(\mathsf{node\text{-}id}_p) =$ `false` when node-id is created, $\mathcal{S}_7$ knows the group state of $\mathsf{node\text{-}id}_p$. Thus, $\mathcal{S}_7$ generates the group state of node-id from it following the protocol as in $\mathcal{S}_6$.

Case (2). If **safe**$(\mathsf{node\text{-}id}_p) =$ `true` when node-id is created, $\mathcal{S}_7$ defers the generation of group state of node-id to the following timing. Note that due to the restricted environment, these are the only subcase of Case (2), and only one of the following cases occurs for each node-id.

Case (2-1) safe(node-id) $=$ `true` **when proposal/commit messages are generated/processed at** node-id**:** $\mathcal{S}_7$ sets G.gid $\leftarrow \mathsf{Node}[\mathsf{node\text{-}id}].\mathsf{gid}$ and G.epoch $\leftarrow \mathsf{Node}[\mathsf{node\text{-}id}].\mathsf{epoch} + 1$. Also, it chooses a random encSecret and stores $L_{\mathsf{encSecret}}[\mathsf{node\text{-}id}] \leftarrow \mathsf{encSecret}$. In this case, it does not generate the other group state.

Case (2-2) Party id **at** node-id **is corrupted (i.e., safe**(node-id) **becomes** `false`**) before proposal/commit messages are generated/processed at** node-id**:** When $\mathcal{A}$ corrupts id at node-id, $\mathcal{S}_7$ receives the following information from the functionality.

- id's current CmPKE secret key dk (stored in CurrDK array)
- id's current signature signing key ssk (stored in SSK array)
- The epoch pointer node-id $=$ Ptr[id] (i.e., id's current node in the history graph)
- The information stored in Node[node-id]

Using this information, $\mathcal{S}_7$ simulates the group states. It first initializes the group state G as follows:

- G.gid $\leftarrow$ Node[node-id].gid
- G.epoch $\leftarrow$ Node[node-id].epoch
- G.member $\leftarrow$ Node[node-id].mem
- G.joinerSecret $\leftarrow\!\!\$\ \{0,1\}^\kappa$
- G.confTransHash-w.o-'id$_c$' $\leftarrow\!\!\$\ \{0,1\}^\kappa$
- G.confTransHash $\leftarrow$ H(G.confTransHash-w.o-'id$_c$', id$_c$), where id$_c$:= Node[node-id].orig
- G.certSvks$[*]$, G.pendUpd$[*]$, G.pendCom$[*]$ $\leftarrow \emptyset$
- G.id $\leftarrow$ id
- G.ssk $\leftarrow$ ssk
- G.member[id].dk $\leftarrow$ dk

Then, $\mathcal{S}_7$ executes the following functions in order to generate group secrets and hash values.

1. G.memberHash $\leftarrow$ `*derive-member-hash`(G)
2. $(G, \mathsf{confKey}) \leftarrow$ `*derive-epoch-keys`$(G, G.\mathsf{joinerSecret})$
3. confTag $\leftarrow$ `*gen-conf-tag`$(G, \mathsf{confKey})$
4. $G \leftarrow$ `*set-interim-trans-hash`$(G, \mathsf{confTag})$

Since the previous epoch is secure and due to the restricted environment, $\mathcal{A}$ cannot corrupt the past secure epochs (cf. lines 9-10 in Figure 5.11). Thus, $\mathcal{A}$ cannot distinguish the simulated group state from a real one since $\mathcal{A}$ cannot check whether the simulated group states are consistent for the previous epoch. Finally, $\mathcal{S}_7$ stores $L_{\mathsf{encSecret}}$[node-id] $\leftarrow$ G.encSecret.

$\mathcal{S}_7$ answers each query made by the ideal functionality $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA},7}$ as in $\mathcal{S}_6$ except for the differences shown below. We only describe $\mathcal{S}_7$ when **safe** is true for the epoch $\mathcal{Z}$ queries. This is because if **safe** is false, $\mathcal{S}_7$ knows the corresponding group state (see above) and obtains the same information $\mathcal{S}_6$ receives from $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA},7}$; thus $\mathcal{S}_7$ can simulate the protocol messages as $\mathcal{S}_6$ does.

Simulation on input (`Create`)**.** $\mathcal{S}_7$ chooses gid at random and returns it to the functionality. Note that $\mathcal{S}_7$ does not generate other group information in this timing (see above for when and how group state is generated). h$(1-\delta)$-correct **Simulation on input** (`Propose`, node-id, |id|, |act|)**.** Let encSecret $:=$ $L_{\mathsf{encSecret}}$[node-id]. $\mathcal{S}_7$ computes $\mathsf{CT}_{\overline{p}} \leftarrow \mathsf{SKE.Enc}(\mathsf{encSecret}, 0^{\ell_{\overline{p}}})$ and sets $p^{\mathsf{ctxt}} := (\mathsf{gid}, \mathsf{epoch}, \text{'proposal'}, \mathsf{CT}_{\overline{p}})$, where gid and epoch is the group identity and the current epoch number of node-id.[26] It returns $(ack := \mathtt{true}, \mathsf{prop\text{-}id} := \bot, p^{\mathsf{ctxt}})$ to the functionality Note that $\ell_{\overline{p}}$, the length of $\overline{p}$, can be computed from the received information and public parameters (which determine the length of the hash value, CmPKE ciphertext, signature, etc.).

Simulation on input (`Commit`, node-id, |id|, $\vec{p}^{\mathsf{ctxt}}$, |svk|, |mem|)**.** For each $p^{\mathsf{ctxt}} \in \vec{p}^{\mathsf{ctxt}}$, $\mathcal{S}_7$ checks whether it satisfies the following conditions.

[26] The simulator always knows gid and epoch for each node-id.

- $\mathsf{PropID}[p^{\mathsf{ctxt}}] \neq \mathsf{node\text{-}id}$, i.e., the received proposal was issued at a different epoch. We call the event $\mathsf{E}_{\mathsf{rej\text{-}1}}$.
- $\mathsf{PropID}[p^{\mathsf{ctxt}}] = \perp$, i.e., the received proposal was generated by the adversary. We call the event $\mathsf{E}_{\mathsf{rej\text{-}2}}$.

If one of the above condition holds for some $p^{\mathsf{ctxt}} \in \vec{p}^{\mathsf{ctxt}}$, $\mathcal{S}_7$ outputs $ack := \texttt{false}$.

Otherwise[27], $\mathcal{S}_7$ generates $\mathsf{CT}_{\bar{c}_0} \leftarrow \mathsf{SKE.Enc}(\mathsf{encSecret}, 0^{\ell_{\bar{c}_0}})$ and $\widehat{c}_i^{\mathsf{ctxt}} \leftarrow \mathsf{SKE.Enc}(\mathsf{encSecret}, 0^{\ell_{\widehat{c}}})$ for $\mathsf{encSecret} := L_{\mathsf{encSecret}}[\mathsf{node\text{-}id}]$ and each $i \in [|\mathsf{mem}|]$. Then, it sets $c_0^{\mathsf{ctxt}} := (\mathsf{gid}, \mathsf{epoch}, \text{'commit'}, \mathsf{CT}_{\bar{c}_0})$ and $\vec{c}^{\mathsf{ctxt}} := \{\widehat{c}_i^{\mathsf{ctxt}}\}_{i \in [|\mathsf{mem}|]}$, and returns $(ack := \texttt{true}, \mathsf{node\text{-}id} := \perp, c_0^{\mathsf{ctxt}}, \vec{c}^{\mathsf{ctxt}})$ to the functionality, where gid and epoch is the group information of the epoch node-id. $\mathcal{S}_7$ stores $L_{\widehat{c}^{\mathsf{ctxt}}}[\mathsf{node\text{-}id}] \leftarrow \vec{c}^{\mathsf{ctxt}}$. Note that both the length of $\bar{c}_0$ and $\widehat{c}$ ($\ell_{\bar{c}_0}$ and $\ell_{\widehat{c}}$) can be computed from the received information and public parameters.

Simulation on input $(\texttt{Welcome}, \mathsf{kp}_t, |\mathsf{gid}|, |\mathsf{epoch}|, |\mathsf{id}|, |\mathsf{mem}|)$**.** $\mathcal{S}_7$ receives this message if **safe** is true for the next epoch, where new members will join. Let ek_t be the CmPKE encryption key in kp_t. $\mathcal{S}_7$ computes the following:

- $\mathsf{ct} \leftarrow \mathsf{CmEnc}(\mathsf{ek}_t, 0)$
- $\mathsf{welcomeSecret}_{\mathsf{id}_t} \leftarrow\!\!\$\ \{0,1\}^\kappa$
- $\mathsf{CT} \leftarrow \mathsf{SKE.Enc}(\mathsf{welcomeSecret}_{\mathsf{id}_t}, 0^{\ell_{(\mathsf{groupInfo,sig})}})$.

Note that $\ell_{(\mathsf{groupInfo,sig})}$ can be computed from the received leakage information and public system parameters. $\mathcal{S}_7$ sets $\widehat{w}^{\mathsf{ctxt}} := (\mathsf{id}_t, (\mathsf{H}(\mathsf{kp}_t), \mathsf{ct}), \mathsf{CT})$ and returns $(ack := \texttt{true}, \widehat{w}^{\mathsf{ctxt}})$ to the functionality. $\mathcal{S}_7$ also stores $L_{\widehat{w}^{\mathsf{ctxt}}} \mathrel{+}\!\!\leftarrow (\mathsf{id}_t, \mathsf{H}(\mathsf{kp}_t), \mathsf{ct}, \mathsf{CT})$.

Simulation on input $(\texttt{Process}, \mathsf{node\text{-}id}, \mathsf{index}, c_0^{\mathsf{ctxt}}, \widehat{c}^{\mathsf{ctxt}}, \vec{p}^{\mathsf{ctxt}})$**.** $\mathcal{S}_7$ first checks the following conditions and outputs $ack := \texttt{false}$ if one of the conditions holds.

- $\mathsf{PropID}[p^{\mathsf{ctxt}}] = \mathsf{node\text{-}id}'$, $\mathsf{NodeID}[c_0^{\mathsf{ctxt}}] = \mathsf{node\text{-}id}'$ or $\widehat{c}^{\mathsf{ctxt}} \in \mathsf{Node}[\mathsf{node\text{-}id}'].\mathsf{vcom}$ such that $\mathsf{node\text{-}id}' \neq \mathsf{node\text{-}id}$, i.e., the received proposal/commit message was generated at a different epoch. We call the event $\mathsf{E}_{\mathsf{rej\text{-}1}}$.
- $\mathsf{PropID}[p^{\mathsf{ctxt}}] = \perp$, $\mathsf{NodeID}[c_0^{\mathsf{ctxt}}] = \perp$ or $\widehat{c}^{\mathsf{ctxt}} \notin \mathsf{Node}[\mathsf{node\text{-}id}'].\mathsf{vcom}$ for all $\mathsf{node\text{-}id}'$, i.e., the received proposal/commit was generated by the adversary. We call the event $\mathsf{E}_{\mathsf{rej\text{-}2}}$.

Otherwise[28], $\mathcal{S}_7$ returns $(ack := \texttt{true}, \perp, \perp, \perp)$.

Simulation on input $(\texttt{Join}, \mathsf{id}, \widehat{w}^{\mathsf{ctxt}})$**.** Let $(\mathsf{id}_t, \mathsf{kphash}_t, \mathsf{ct}_t, \mathsf{CT}_t) := \widehat{w}^{\mathsf{ctxt}}$. (In the following, we assume $\mathsf{id}_t = \mathsf{id}$ and id succeeds to fetch kp_t such that $\mathsf{kphash}_t = \mathsf{h}(\mathsf{kp}_t)$; otherwise both $\mathcal{S}_6$ and $\mathcal{S}_7$ return $ack := \texttt{false}$.) If $\widehat{w}^{\mathsf{ctxt}}$ is generated by an honest committer (i.e., `*succeed-wel` returns `true`), $\mathcal{S}_7$ returns $(ack := \texttt{true}, \perp, \perp, \perp, \perp, \perp)$. Else, $\mathcal{S}_7$ processes the welcome message as in $\mathcal{S}_6$ (i.e., following protocol description).

Part 2: Indistinguishability of Two Hybrids. We then prove the following lemma.

Lemma 5.4.8. *Hybrid 6 and Hybrid 7 are indistinguishable assuming the INDCCA security of* CmPKE *and the* IND-CCA *security of* SKE.

Proof. To show Lemma 5.4.8, we consider the following sub-hybrids between Hybrid 6 and Hybrid 7.

[27] This implies $\vec{p}^{\mathsf{ctxt}}$ only contains proposals such that $\mathsf{PropID}[p^{\mathsf{ctxt}}] = \mathsf{node\text{-}id}$ and thus `*succeed-com` returns true.

[28] This implies $(c_0^{\mathsf{ctxt}}, \widehat{c}^{\mathsf{ctxt}}, \vec{p}^{\mathsf{ctxt}})$ is honestly issued at node-id and thus `*succeed-proc` returns `true`

Hybrid 6-0 := Hybrid 6. This is identical to Hybrid 6. We use the functionality $\mathcal{F}^{\text{ctxt}}_{\text{CGKA},6}$, and the simulator $\mathcal{S}_{6\text{-}0} := \mathcal{S}_6$.

Hybrid 6-1 In this hybrid, the simulator $\mathcal{S}_{6\text{-}1}$ is defined exactly as $\mathcal{S}_{6\text{-}0}$ except that it always outputs $ack :=$ `false` to the functionality when the event $\mathsf{E}_{\text{rej-1}}$ occurs.

Hybrid 6-2 In this hybrid, the simulator $\mathcal{S}_{6\text{-}2}$ is defined exactly as $\mathcal{S}_{6\text{-}1}$ except that it replaces encSecret and $\mathsf{welcomeSecret}_{\mathsf{id}_t}$ with a random value if **safe** is true.

Hybrid 6-3. In this hybrid, the simulator $\mathcal{S}_{6\text{-}3}$ is defined exactly as $\mathcal{S}_{6\text{-}2}$ except that it always outputs $ack :=$ `false` when the event $\mathsf{E}_{\text{rej-2}}$ occurs.

Hybrid 6-4. In this hybrid, the simulator $\mathcal{S}_{6\text{-}4}$ is defined exactly as $\mathcal{S}_{6\text{-}3}$ except that it replaces a CmPKE ciphertext in welcome messages with a ciphertext of 0 if **safe** is true for the next epoch where new members join with the welcome messages.

Hybrid 6-5. In this hybrid, the simulator $\mathcal{S}_{6\text{-}5}$ is defined exactly as $\mathcal{S}_{6\text{-}4}$ except that it replaces SKE ciphertexts in welcome messages with encryption of zero-string if **safe** is true for the epoch where new members join with the welcome messages.

Hybrid 6-6. In this hybrid, the simulator $\mathcal{S}_{6\text{-}6}$ is defined exactly as $\mathcal{S}_{6\text{-}5}$ except that it replaces SKE ciphertexts in proposal and commit messages with encryption of zero-string if **safe** is true for the epoch where these messages are issued. Note that $\mathcal{S}_{6\text{-}6}$ is identical to $\mathcal{S}_7$.

Hybrid 6-7 = Hybrid 7. We replace the functionality $\mathcal{F}^{\text{ctxt}}_{\text{CGKA},6}$ with the functionality $\mathcal{F}^{\text{ctxt}}_{\text{CGKA},7}$. The simulator $\mathcal{S}_7$ is defined by the above description. Since $\mathcal{S}_7$ has simulated the protocol with the information given from $\mathcal{F}^{\text{ctxt}}_{\text{CGKA},7}$, Hybrid 6-6 and Hybrid 6-7 are identical.

From Lemmata 5.4.9 to 5.4.12, 5.4.14 and 5.4.16 provided below, Hybrid 6-0 and Hybrid 6-6 are indistinguishable. Therefore, we conclude that Hybrid 6 and Hybrid 7 are indistinguishable. □

From Hybrid 6-0 to 6-1: Lemma 5.4.9.

Lemma 5.4.9. *Hybrid 6-0 and Hybrid 6-1 are indistinguishable assuming the key-committing property of* SKE.

Proof. The difference between $\mathcal{S}_{6\text{-}0}$ and $\mathcal{S}_{6\text{-}1}$ is $\mathcal{S}_{6\text{-}1}$ always outputs $ack :=$ `false` when the event $\mathsf{E}_{\text{rej-1}}$ occurs. In other words, if $\mathcal{S}_{6\text{-}0}$ outputs $ack :=$ `true` when $\mathsf{E}_{\text{rej-1}}$ occurs, $\mathcal{Z}$ can distinguish the two hybrids. We show that, if $\mathcal{Z}$ can distinguish the two hybrids, then there exists an adversary $\mathcal{B}$ that breaks the key-committing property of SKE.

$\mathcal{B}$ simulates for $\mathcal{Z}$ the interaction with $\mathcal{F}^{\text{IW}}_{\text{AS}}$ and $\mathcal{F}^{\text{IW}}_{\text{KS}}$, and the protocol execution of each party as in $\mathcal{S}_{6\text{-}0}$. Assume $\mathsf{E}_{\text{rej-1}}$ occurs for some SKE ciphertext CT and $\mathcal{Z}$ distinguish the two hybrids when CT is processed. By the condition of $\mathsf{E}_{\text{rej-1}}$, there exists the encryption key k (encSecret or $\mathsf{welcomeSecret}_{\mathsf{id}_t}$) used to generate CT and the other encryption key $k' \neq k$ that correctly decrypts CT. This means CT can be correctly decrypted with both k and k', which implies $\mathcal{B}$ can break the key-committing property of SKE. This contradicts the assumption that SKE has key-committing property. Therefore, both $\mathcal{S}_{6\text{-}0}$ and $\mathcal{S}_{6\text{-}1}$ always outputs $ack :=$ `false` when $\mathsf{E}_{\text{rej-1}}$ occurs. □

From Hybrid 6-1 to 6-2: Lemma 5.4.10.

Lemma 5.4.10. *Hybrid 6-1 and Hybrid 6-2 are indistinguishable assuming* CmPKE *is* Chained CmPKE *conforming GSD secure.*

Proof. The proof is identical to the proof in [Has+21c, Lemma E.29]. To prove the indistinguishability of the two hybrids, we gradually replace each encSecret and $\mathsf{welcomeSecret}_{\mathsf{id}_t}$ with a random value if **safe** is true when a propose/commit/welcome message is created. Similar to the proof of randomness of appSecret provided in [Has+21c, Lemma E.29], we can show that, if $\mathcal{Z}$ can distinguish whether encSecret or $\mathsf{welcomeSecret}_{\mathsf{id}}$ are real or random, it can be used to break the Chained CmPKE conforming GSD security of CmPKE. Note that the value of **safe** is fixed when a propose/commit/welcome message is created at the epoch (cf. `*mark-content-hiden-epoch`), and the adversary is restricted from colluding to change this value (cf. line 11 in Figure 5.11). Thus, we can construct a reduction $\mathcal{B}$ as in [Has+21c, Lemma E.29]. Therefore, if CmPKE is Chained CmPKE conforming GSD secure, Hybrid 6-1 and Hybrid 6-2 are indistinguishable. □

From Hybrid 6-2 to 6-3: Lemma 5.4.11.

Lemma 5.4.11. *Hybrid 6-2 and Hybrid 6-3 are indistinguishable assuming* SKE *is the* IND-CCA *secure.*

Proof. The difference between $\mathcal{S}_{\text{6-2}}$ and $\mathcal{S}_{\text{6-3}}$ is $\mathcal{S}_{\text{6-3}}$ always outputs *ack* := `false` when the event $\mathsf{E}_{\text{rej-2}}$ occurs[29]. In other words, $\mathcal{Z}$ distinguishes the two hybrids if the adversary produces malicious ciphertexts that can be decrypted correctly. Such a malicious ciphertext may contain (1) a malicious plain message (i.e., the message generated by the adversary) or (2) an honest plain message (i.e., the message generated by an honest party at the same epoch). For the former case, we already proved that the adversary cannot produce acceptable messages without knowing MAC key (cf. Hybrid 5). Thus, the previous simulator also outputs *ack* := `false` in the former case. Therefore, we care about the latter case. We show that, if $\mathcal{Z}$ can distinguish the two hybrids when the adversary produces a malicious ciphertext that contains an honestly generated plain message, then we can construct an adversary $\mathcal{B}$ that breaks the IND-CCA security of SKE.

$\mathcal{B}$ simulates for $\mathcal{Z}$ the interaction with $\mathcal{F}_{\text{AS}}^{\text{IW}}$ and $\mathcal{F}_{\text{KS}}^{\text{IW}}$, and the protocol execution of each party as in $\mathcal{S}_{\text{6-2}}$ except for encrypting messages. At the beginning of the game, $\mathcal{B}$ chooses $i \in [Q]$ at random (where Q is the total number of honestly generated epochs) and hopes that $\mathcal{Z}$ distinguishes the hybrids when the event occurs in the i-th honest epoch. That is, we assume that, when a party processes some ciphertext CT with the i-th honest encryption key, $\mathcal{S}_{\text{6-2}}$ outputs *ack* := `true`, but $\mathcal{S}_{\text{6-3}}$ outputs *ack* := `false`. ($\mathcal{B}$ succeeds to guess such an epoch with probability $1/Q$.) $\mathcal{B}$ embed the challenge SKE key of the IND-CCA game to the i-th encSecret. Note that if **safe** is true for the corresponding epoch, the encryption key is chosen at random due to the modification we made in Hybrid 6-2; thus the challenge key can be embedded. In addition, the adversary cannot corrupt the challenge key after the key is used for encryption since the adversary's corruption is restricted (cf. `*mark-content-hiden-epoch` function). When $\mathcal{B}$ encrypts messages with the i-th encSecret (i.e., the challenge key), it uses the encryption oracle $\mathcal{LR}$ by setting m_0 as the actual message and m_1 as a random message with the same length. $\mathcal{B}$ uses its decryption oracle when it wants to decrypt a ciphertext[30]. If the decryption result is identical to the message m_b, $\mathcal{B}$ outputs the bit b to the challenger of the IND-CCA game. Else, the decryption result is different from the messages sent to the encryption oracle, it outputs *ack* := `false`. Note that for other keys, $\mathcal{B}$ simulates protocol as in the previous hybrid.

We can verify that $\mathcal{B}$ wins the IND-CCA game if the adversary can produce a ciphertext that contains an honestly generated message. Note that the messages m_{1-b} are information-theoretically hidden from the

[29] The event $\mathsf{E}_{\text{rej-2}}$ only occurs in epochs where **safe** is true.

[30] By definition of the simulator, $\mathcal{B}$ only sends ciphertexts that are not produced by the encryption oracle.

adversary (and $\mathcal{Z}$). Hence, if $\mathcal{Z}$ can distinguish the two hybrids, $\mathcal{B}$ can break the IND-CCA security of SKE. This contradicts the assumption that SKE is the IND-CCA secure. Therefore, $\mathcal{Z}$ cannot distinguish the two hybrids. In other words, both $\mathcal{S}_{6\text{-}2}$ and $\mathcal{S}_{6\text{-}3}$ outputs $ack = \texttt{false}$ when $\mathsf{E}_{\mathsf{rej}\text{-}2}$ occurs. □

From Hybrid 6-3 to 6-4: Lemma 5.4.12.

Lemma 5.4.12. *Hybrid 6-3 and Hybrid 6-4 are indistinguishable assuming* CmPKE *is* IND-CCA *secure.*

Proof. We assume $\mathcal{Z}$ creates at most W welcome messages by `Commit` query. To show Lemma 5.4.12, we consider the following sub-hybrids between Hybrid 6-3 and Hybrid 6-4.

Hybrid 6-3-0 := Hybrid 6-3. This is identical to Hybrid 6-3. The simulator $\mathcal{S}_{6\text{-}3\text{-}0} = \mathcal{S}_{6\text{-}3}$ generates protocol messages following the protocol procedures.

Hybrid 6-3-*i*. i runs through $[W]$. The simulator $\mathcal{S}_{6\text{-}3\text{-}i}$ is defined exactly as $\mathcal{S}_{6\text{-}3\text{-}(i-1)}$ except that, when it simulates the i-th welcome message, if the **safe** predicates is true for the next epoch the welcome message is associated with, it encrypts zero-strings instead of the joiner secret with the new member's CmPKE encryption key. Note that we count welcome messages in the order the simulator creates. We show in Lemma 5.4.13 that Hybrid 6-3-$(i-1)$ and Hybrid 6-3-i are indistinguishable.

Hybrid 6-4. This is identical to Hybrid 6-3-W. In this hybrid, CmPKE ciphertexts in welcome messages issued for secure epochs (i.e., **safe** is true) are replaced with the encryption of zero-string.

The indistinguishability between Hybrid 6-3-0 and Hybrid 6-3-Q is derived by applying Lemma 5.4.13 for all $i \in [W]$. Therefore, we conclude that Hybrid 6-3 and Hybrid 6-4 are indistinguishable. □

The indistinguishability of Hybrid 6-3 and Hybrid 6-4 follows from Lemma 5.4.13 below.

Lemma 5.4.13. *Hybrid 6-3-$(i-1)$ and Hybrid 6-3-i are indistinguishable assuming* CmPKE *is* IND-CCA *secure*[31].

Proof. The difference between $\mathcal{S}_{6\text{-}3\text{-}(i-1)}$ and $\mathcal{S}_{6\text{-}3\text{-}i}$ is, when generating the i-th welcome message, if the next epoch is secure (i.e., **safe** is true for the next epoch), $\mathcal{S}_{6\text{-}3\text{-}i}$ replaces the CmPKE ciphertext in the i-th welcome message with a ciphertext of 0 encrypted with the new member's CmPKE encryption key (the simulator receives the new member's key package from the functionality). We show that, if $\mathcal{Z}$ can distinguish the two hybrids, then there exists an adversary $\mathcal{B}$ that breaks IND-CCA security of CmPKE. We first explain the description of $\mathcal{B}$ and then evaluate $\mathcal{B}$'s advantage.

$\mathcal{B}$ simulates for $\mathcal{Z}$ the interaction with $\mathcal{F}_{\mathrm{AS}}^{\mathrm{IW}}$ and $\mathcal{F}_{\mathrm{KS}}^{\mathrm{IW}}$, and the protocol execution as in $\mathcal{S}_{6\text{-}3\text{-}(i-1)}$ except for generating the i-th welcome message. Let ek^* be the challenge CmPKE encryption key provided by the IND-CCA game. Observer that honest key packages are generated on `register-kp` queries to $\mathcal{F}_{\mathrm{KS}}^{\mathrm{IW}}$. We assume $\mathcal{Z}$ issues at most Q `register-kp` queries. At the beginning of the game, $\mathcal{B}$ chooses an index $J \in [Q]$ at random, embeds the challenge key in the J-th `register-kp` query, and hopes that the J-th key package will be used to add the i-th welcome message. ($\mathcal{B}$ succeeds to guess with probability $1/Q$.) For other `register-kp` queries, $\mathcal{B}$ generates key packages following the description of $\mathcal{F}_{\mathrm{KS}}^{\mathrm{IW}}$. If $\mathcal{B}$ decrypts a ciphertext with the challenge decryption key, $\mathcal{B}$ uses the decryption oracle provided by the IND-CCA game.

Assume the i-th welcome message is created with the J-th key package and **safe** is true for the new epoch where the i-th welcome message is created. Let joinerSecret be the corresponding joiner secret. $\mathcal{B}$ outputs $\mathsf{m}_0 :=$ joinerSecret and $\mathsf{m}_1 := 0$ to IND-CCA game challenger and receives the challenge ciphertext

[31] We use the IND-CCA game with $N = 1$.

$(\mathsf{ct}_0^*, \widehat{\mathsf{ct}}^*)$. $\mathcal{B}$ uses it as the CmPKE ciphertext in the i-th welcome message. Note that, since **safe** is true, the adversary has not corrupted the challenge key, and it is restricted from colluding to compute the challenge key (cf. `*mark-content-hiden-epoch` function). Thus, $\mathcal{B}$ never corrupts the challenge key.

We finally analyze $\mathcal{B}$'s advantage. If the challenger returns the ciphertext of the joiner secret, $\mathcal{Z}$'s view is identical to Hybrid 6-3-$(i-1)$; else, i.e., the challenger returns the ciphertext of 0, $\mathcal{Z}$'s view is identical to Hybrid 6-3-i. Hence, if $\mathcal{Z}$ distinguishes Hybrid 6-3-$(i-1)$ and Hybrid Hybrid 6-3-i with non-negligible probability, $\mathcal{B}$ wins the IND-CCA game with non-negligible probability by using $\mathcal{Z}$'s output. This contradicts the assumption that CmPKE is IND-CCA secure. Therefore, Hybrid 6-3-$(i-1)$ and Hybrid Hybrid 6-3-i are indistinguishable. □

From Hybrid 6-4 to 6-5: Lemma 5.4.14.

Lemma 5.4.14. *Hybrid 6-4 and Hybrid 6-5 are indistinguishable assuming* SKE *is the* IND-CCA *secure.*

Proof. We assume $\mathcal{Z}$ creates at most W welcome messages by `Commit` query. To show Lemma 5.4.14, we consider the following sub-hybrids between Hybrid 6-4 and Hybrid 6-5.

Hybrid 6-4-0 := Hybrid 6-4. This is identical to Hybrid 6-4. The simulator $\mathcal{S}_{\text{6-4-0}} = \mathcal{S}_{\text{6-4}}$ generates protocol messages following the protocol procedures.

Hybrid 6-4-i. i runs through $[W]$. The simulator $\mathcal{S}_{\text{6-4-}i}$ is defined exactly as $\mathcal{S}_{\text{6-4-}(i-1)}$ except that, when it simulates the i-th welcome message, if **safe** is true for the next epoch the welcome message is associated with, it generates a ciphertext as $\mathsf{CT} \leftarrow \mathsf{SKE.Enc}(\mathsf{welcomeSecret}_{\mathsf{id}_t}, 0^{\ell(\mathsf{groupInfo,sig})})$. Note that we count welcome messages in the order the simulator creates. We show in Lemma 5.4.15 that Hybrid 6-4-$(i-1)$ and Hybrid 6-4-i are indistinguishable.

Hybrid 6-5. This is identical to Hybrid 6-4-W. In this hybrid, SKE ciphertext in welcome messages issued for secure epochs (i.e., **safe** is true) is replaced with a random string.

The indistinguishability between Hybrid 6-4-0 and Hybrid 6-4-Q is derived by applying Lemma 5.4.15 for all $i \in [W]$. Therefore, we conclude that Hybrid 6-4 and Hybrid 6-5 are indistinguishable. □

The indistinguishability of Hybrid 6-4 and Hybrid 6-5 follows from Lemma 5.4.15 below.

Lemma 5.4.15. *Hybrid 6-4-$(i-1)$ and Hybrid 6-4-i are indistinguishable assuming* SKE *is the* IND-CCA *secure.*

Proof. The difference between $\mathcal{S}_{\text{6-4-}(i-1)}$ and $\mathcal{S}_{\text{6-4-}i}$ is, when generating the i-th welcome message, if the next epoch is secure (i.e., **safe** is true for the next epoch), $\mathcal{S}_{\text{6-4-}i}$ replaces the SKE ciphertext in the i-th welcome message with an encryption of zero-string. We show that, if $\mathcal{Z}$ can distinguish the two hybrids, then there exists an adversary $\mathcal{B}$ that breaks the IND-CCA security of SKE. We first explain the description of $\mathcal{B}$ and then evaluate $\mathcal{B}$'s advantage.

$\mathcal{B}$ simulates for $\mathcal{Z}$ the interaction with $\mathcal{F}_{\text{AS}}^{\text{IW}}$ and $\mathcal{F}_{\text{KS}}^{\text{IW}}$, and the protocol execution as in $\mathcal{S}_{\text{6-4-}(i-1)}$ except for generation of SKE ciphertext in the i-th welcome message if **safe** is true for the new epoch corresponding to the i-th welcome message. $\mathcal{B}$ embed the challenge SKE key of the IND-CCA game to $\mathsf{welcomeSecret}_{\mathsf{id}_t}$, which is used to encrypt the i-th welcome message. If **safe** is true for the corresponding epoch, welcomeSecret is chosen at random due to the modification we made in Hybrid 6-1; thus the challenge key can be embedded. In addition, the adversary cannot corrupt the challenge key after messages are encrypted with the key since the adversary's corruption is restricted (cf. `*mark-content-hiden-epoch` function).

When $\mathcal{B}$ encrypts the contents m in the i-th welcome message, it queries $\mathsf{CT} := \mathcal{LR}(\mathsf{m}_0 := (\mathsf{groupInfo}, \mathsf{sig}), \mathsf{m}_1 := 0^{\ell_{(\mathsf{groupInfo},\mathsf{sig})}})$ and uses the challenge ciphertext CT as the ciphertext in the i-th welcome message. When $\mathcal{B}$ decrypts ciphertexts different from challenge ciphertexts with the challenge key, it uses its decryption oracle.

We finally analyze $\mathcal{B}$'s advantage. If the oracle $\mathcal{LR}$ returns a ciphertext of the real contents m_0, $\mathcal{Z}$'s view is identical to Hybrid 6-4-$(i-1)$; else, i.e., the oracle $\mathcal{LR}$ returns a ciphertext of zero-string, $\mathcal{Z}$'s view is identical to Hybrid 6-4-i. Hence, if $\mathcal{Z}$ distinguishes Hybrid 6-4-$(i-1)$ and Hybrid Hybrid 6-4-i with non-negligible probability, $\mathcal{B}$ breaks the IND-CCA security of SKE with non-negligible probability by using $\mathcal{Z}$'s output. This contradicts the assumption that SKE is IND-CCA secure. Therefore, Hybrid 6-4-$(i-1)$ and Hybrid Hybrid 6-4-i are indistinguishable. □

From Hybrid 6-5 to 6-6: Lemma 5.4.16.

Lemma 5.4.16. *Hybrid 6-5 and Hybrid 6-6 are indistinguishable assuming* SKE *is* IND-CCA *secure.*

Proof. We assume $\mathcal{Z}$ creates at most Q epochs (i.e., commit nodes). To show Lemma 5.4.16, we consider the following sub-hybrids between Hybrid 6-5 and Hybrid 6-6.

Hybrid 6-5-0 := Hybrid 6-5. This is identical to Hybrid 6-5. We use the functionality $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA},6}$, and the simulator $\mathcal{S}_{6\text{-}5} = \mathcal{S}_{6\text{-}5\text{-}0}$.

Hybrid 6-5-i. i runs through $[Q]$. The simulator $\mathcal{S}_{6\text{-}5\text{-}i}$ is defined exactly as $\mathcal{S}_{6\text{-}5\text{-}(i-1)}$ except that, when a party issues proposal or commit messages at the i-th epoch, if **safe** is true for the epoch, it encrypts a zero-string instead of the real contents in the non-encrypted proposal/commit message. Note that we count epochs in the order in which `Propose` or `Commit` is called. We show in Lemma 5.4.17 that Hybrid 6-5-$(i-1)$ and Hybrid 6-5-i are indistinguishable.

Hybrid 6-6. This is identical to Hybrid 6-5-Q. In this hybrid, ciphertexts in proposal and commit messages issued at secure epochs (epochs such that **safe** is true) are replaced with random bit-strings.

The indistinguishability between Hybrid 6-5-0 and Hybrid 6-5-Q is derived by applying Lemma 5.4.17 for all $i \in [Q]$. Therefore, we conclude that Hybrid 6-5 and Hybrid 6-6 are indistinguishable. □

The indistinguishability of Hybrid 6-5 and Hybrid 6-6 follows from Lemma 5.4.17 below.

Lemma 5.4.17. *Hybrid 6-5-$(i-1)$ and Hybrid 6-5-i are indistinguishable assuming* SKE *is* IND-CCA *secure.*

Proof. The difference between $\mathcal{S}_{6\text{-}5\text{-}(i-1)}$ and $\mathcal{S}_{6\text{-}5\text{-}i}$ is, when generating proposal or commit messages at the i-th epoch such that **safe** is true, $\mathcal{S}_{6\text{-}5\text{-}i}$ encrypts a zero-string instead of the real contents. We show that, if $\mathcal{Z}$ can distinguish the two hybrids, then there exists an adversary $\mathcal{B}$ that breaks the IND-CCA security of SKE. We first explain the description of $\mathcal{B}$. Then we evaluate $\mathcal{B}$'s advantage.

$\mathcal{B}$ simulates for $\mathcal{Z}$ the interaction with $\mathcal{F}^{\mathrm{IW}}_{\mathrm{AS}}$ and $\mathcal{F}^{\mathrm{IW}}_{\mathrm{KS}}$, and the protocol execution of each party as in $\mathcal{S}_{6\text{-}5\text{-}(i-1)}$ except for encryption of proposal and commit messages generated at the i-th epoch if **safe** is true. $\mathcal{B}$ embed the challenge SKE key of the IND-CCA game to $\mathsf{encSecret}$ of the i-th epoch (if **safe** is true for the i-th epoch, $\mathsf{encSecret}$ is chosen at random due to the modification we made in Hybrid 6-2; thus the challenge key can be embedded). In addition, the adversary cannot corrupt the challenge key after messages are encrypted with the key since the adversary's corruption is restricted (cf. `*mark-content-hiden-epoch` function). When $\mathcal{B}$ encrypts contents m in a proposal or commit message at the i-th epoch, it queries

$\mathsf{CT} := \mathcal{LR}(\mathsf{m}_0 := \mathsf{m}, \mathsf{m}_1 := 0^{|\mathsf{m}|})$. Note that due to the modification we made in Hybrid 6-3, $\mathcal{B}$ can reject any messages which do not generate with the i-th encryption key or generated by the adversary. Thus it does not need to use the decryption oracle.

We finally analyze $\mathcal{B}$'s advantage. If the encryption oracle $\mathcal{LR}$ returns ciphertexts of m, $\mathcal{Z}$'s view is identical to Hybrid 6-5-$(i-1)$; else, i.e., the $\mathcal{LR}$ oracle returns ciphertexts of zero-string, $\mathcal{Z}$'s view is identical to Hybrid 6-5-i. In addition, the i-th encryption secret is hidden from the adversary. Hence, if $\mathcal{Z}$ distinguishes Hybrid 6-5-$(i-1)$ and Hybrid Hybrid 6-5-i with non-negligible probability, $\mathcal{B}$ wins the IND-CCA game with non-negligible probability by using $\mathcal{Z}$'s output. This contradicts the assumption that SKE is IND-CCA secure. Therefore, Hybrid 6-5-$(i-1)$ and Hybrid Hybrid 6-5-i are indistinguishable. □

5.5 Metadata-Hiding CGKA: Construct Wrapper Protocol W^{mh}

We now construct a simple and modular CGKA construction that enables us to secure the 3rd layer (i.e., *dynamic* metadata). This results in the first *metadata-hiding* CGKA. Technically, we construct a wrapper protocol W^{mh} specifically taking care of the 3rd layer security in the $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$-hybrid model. Specifically, W^{mh} can be wrapped around *any* CGKA Π_{ctxt} that UC-realizes $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ and bootstraps Π_{ctxt} into a metadata-hiding CGKA.

Below, we provide an overview of the proposed protocol. We then provide the full detail on the construction of W^{mh} in Section 5.5.3. The formal security proof of our metadata-hiding CGKA is deferred to Section 5.6, where we propose a new UC security model capturing the 3rd layer.

5.5.1 Goal of the Wrapper Protocol W^{mh}

To claim that CGKA secures the 3rd layer, our goal is to informally construct a wrapper protocol W^{mh} with the following properties:

(1) **Anonymous upload.** A group member id can anonymously upload a proposal p or a commit $(c_0, \vec{c})$ to the server.

(2) **Anonymous download.** A group member id can anonymously download a commit $(c_0, \vec{c})$ from the server.

(3) **Unlinkability.** A member performing multiple uploads and downloads remains unlinkable from the server.

(4) **Group authentication.** Only members of the group — excluding the *honest* server — can upload to and download from the server.

Here, unlinkability (Item (3)) is a strictly stronger notion compared to anonymity (Item (1)). Even if the member remains anonymous, some protocol may allow the server to link whether two uploads came from the same member. We also note that Item (4) is only relevant when the server is *honest* — a malicious server can always allow a non-member to perform an upload or a download on behalf of the group.

A wrapper protocol W^{mh} satisfying the above conditions is sufficient to bootstrap any CGKA that UC-realizes $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ into a metadata-hiding CGKA. However, if the underlying CGKA supports *selective downloading*, such as Chained CmPKE, Item (2) fails to provide the same efficiency offered by the underlying selective downloading CGKA. We thus strengthen Item (2) as follows.

(2$^+$) **Anonymous selective download.** A group member id can anonymously and *selectively* download a partial commit $(c_0, \widehat{c}_{\mathsf{id}})$ from the server.

Specifically, the wrapper protocol W^{mh} allows the download cost to remain independent of the group member size in case a selective downloading CGKA is internally used.

Finally, even if the group secret key becomes compromised, we want all the above properties to hold again once the group state is healed. Namely, we seek a protocol with the following property.

(5) **Compromise Resilience.** W^{mh} inherits the FS+PCS guarantees offered by the underlying CGKA UC-realizing $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$.

Insufficiency of client-anonymized authenticated channel. One natural idea is to use a client-anonymized authenticated channel such as a VPN or an anonymized proxy like Tor [DMS04; Gua]. While such a channel solves Item (1) (and Item (2)), this alone cannot solve Items (2$^+$) to (5). For instance, when selective downloading is performed, member id needs to specify its index in the group to retrieve the partial commit $(c_0, \widehat{c}_{\mathsf{id}})$. Even if index may not directly leak id, the second time id performs a download on the same index, it will break linkability (Item (3)). Moreover, since the client-side is unauthenticated, it does not prevent external adversaries to perform a denial of service (DoS) attack on the group by uploading garbage contents, thus contradicting Item (4). Recall here that the server can no longer explicitly check if the uploaded contents come from genuine group members since the static metadata including the identity of the uploading member is hidden.

In summary, a client-anonymized authenticated channel alone is not enough to hide dynamic metadata.

5.5.2 High Level Description of the Wrapper Protocol W^{mh}

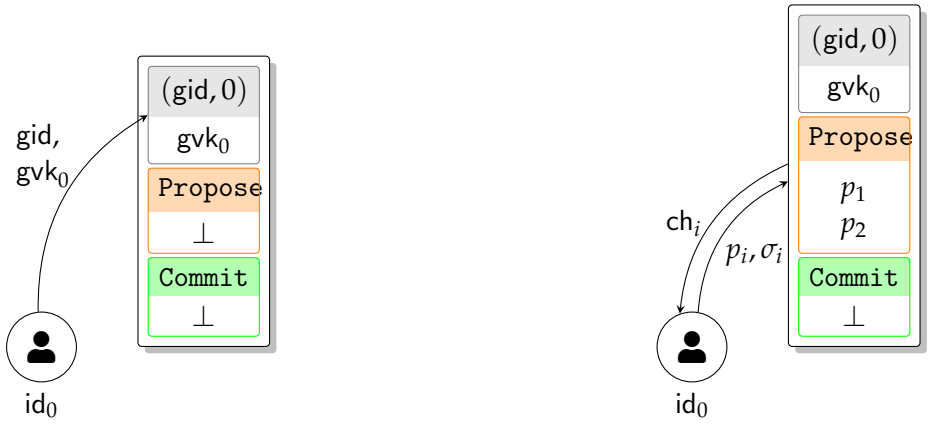
We provide an overview of our wrapper protocol W^{mh}. The main idea is to use the unique group secret key k exchanged among the group to perform an efficient proof of membership to the server. To make the presentation simple, we deliberately provide an informal description of the proposed protocol.

Below, we assume all parties have access to the ideal functionality $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$. Moreover, we assume the party communicates with the server via a client-anonymized authenticated channel, except when a new member is retrieving a welcome message from the server.

Figure 5.33a: Group registration. Assume party id_0 wishes to create a group of three members $(\mathsf{id}_0, \mathsf{id}_1, \mathsf{id}_2)$. id_0 first registers a new *empty* group to the server. Informally, id_0 invokes $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ on input $(\texttt{Create})$ and initializes a new group identifier gid and a group secret key k_0 for $\mathsf{epoch} = 0$. It then *deterministically* creates a group specific signature key $(\mathsf{gvk}_0, \mathsf{gsk}_0) \leftarrow \mathsf{KeyGen}(1^\kappa; \mathsf{PRF}(\mathsf{k}_0, \text{`auth'}))$ from the group secret key k_0. We call this verification key gvk_0 as the *group statement* for $\mathsf{epoch} = 0$. Party id_0 then uploads the pair $(\mathsf{gid}, \mathsf{gvk}_0)$ to the server.[32] Finally, the server creates a new database for the group gid.

Figure 5.33b: Initial proposal to add members. With the database for gid set up on the server, id_0 next adds id_1 and id_2 to the group. Specifically, id_0 invokes $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ on input $(\texttt{Propose}, \text{`add'-}\mathsf{id}_i)$ and generates an add proposal p_i for $i \in [2]$. To upload p_i on the server, id_0 proves that it is a member of the group gid by essentially executing an identification protocol with the server. The server sends a random challenge $\mathsf{ch}_i \leftarrow \{0,1\}^\kappa$ and id_0 creates a signature $\sigma_i \leftarrow \mathsf{Sign}(\mathsf{gsk}_0, \mathsf{ch}_i)$. The server verifies that σ_i is a valid signature with respect to the group statement gvk_0 at $\mathsf{epoch} = 0$. If so, it adds p_i to the database. Due to the unforgeability of the signature scheme, no party without the group signing key gsk_0 can impersonate a group member.

[32]The proposed protocol W^{mh} does not prevent a malicious id_0 from registering multiple groups. As explained in Section 5.1.2, one possible way to thwart such a DoS attack would be to use anonymous credentials [Cha82].



(A) Group registration

(B) Initial proposals from id_0

FIGURE 5.33: Creation of a group by a party id_0. The three-part box represents the group state at a given epoch, as stored on the server. The top box stores in the clear the group identifier gid, epoch and group statement gvk_0. The middle box stores the (encrypted) proposals created during the epoch. The bottom box stores the (encrypted) commit message which concluded this epoch, if it exists.
To prove membership, the server sends a challenge ch_i and id_0 responds with a signature σ_i (Figure 5.33b). The contents are exchanged over a client-anonymized authenticated channel.

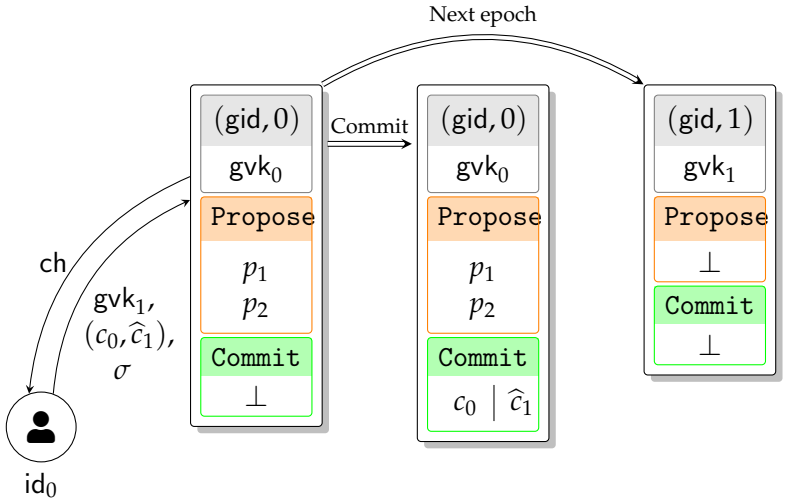


FIGURE 5.34: An initial commit made by the group creator id_0. The server freezes the state of the current epoch $= 0$, and initializes a new epoch in the database.

Figure 5.34: Initial commit to execute initial proposals. To create a new group with members id_1 and id_2, id_0 must commit the proposals $\vec{p} = (p_i)_{i\in[2]}$. It first invokes $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ on input $(\texttt{Commit}, \vec{p})$ and generates $(c_0, \widehat{c}_0, \vec{w} = (\widehat{w}_i)_{i\in[2]})$[33] and (roughly) updates the group secret key k_1 for the next epoch $= 1$. Similar to the group registration phase, id_0 creates a group statement gvk_1 for epoch $= 1$. To upload the commit $(c_0, \widehat{c}_0)$, id_0 performs the same identification protocol as in Figure 5.33b to prove that he is indeed a member of the

[33]Note that id_0 must process the commit $(c_0, \widehat{c}_1)$ to move to the next epoch $= 1$.

group gid. If the identification protocol succeeds, the server stores the commit on the database and further creates a new column for the next epoch $= 1$.

Finally, id_0 uploads the welcome messages $\vec{w}$ to the server. The welcome messages are stored on a party-dependent database and work identically to $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$. In particular, id_1 and id_2 can retrieve $\widehat{w}_1$ and $\widehat{w}_2$, respectively, from the server and execute $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ on input $(\texttt{Join}, \widehat{w}_i)$ to have the same group state as id_0. Effectively, they become a member of the group gid at epoch $= 1$.

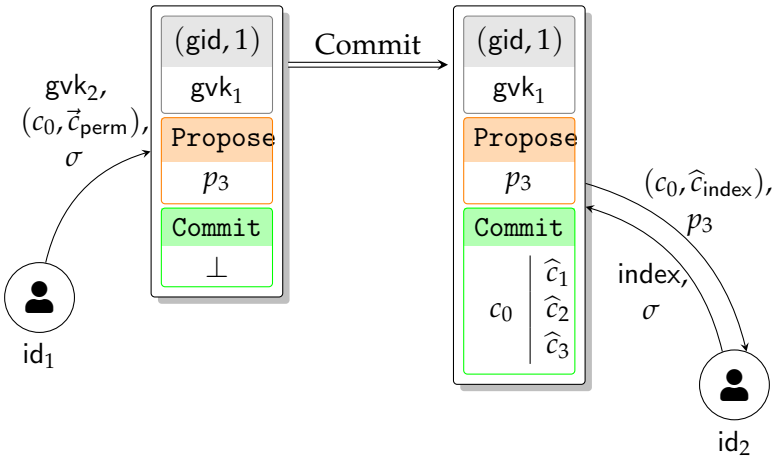


FIGURE 5.35: Left: a commit sent by id_1. Right: a subsequent process made by id_2. Member-dependent commits $\vec{c}_{\mathsf{perm}} := (\widehat{c}_1, \widehat{c}_2, \widehat{c}_3)$ are randomly permuted and id_2 specifies an index to fetch the commit from the server. Challenges sent by the server are omitted for readability.

Commits without selective downloading. Now that id_1 and id_2 joined the group gid at epoch $= 1$ (i.e., share the same group secret key k_1), they can upload proposals and commits to the database defined with respect to the group statement gvk_1.

We now explain the structure of a commit message when the group has more than one member. Assume some member made an update proposal p_3, and id_1 wishes to commit this proposal (left half of Figure 5.35). Then, following the same procedure as the initial commit, id_1 invokes $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ on input $(\texttt{Commit}, p_3)$ and generates $(c_0, \vec{c} = (\widehat{c}_i)_{i\in[3]})$, along with an updated group statement gvk_2 for the next epoch $= 2$. If selective downloading is not performed, then id_1 can simply upload $(c_0, \vec{c})$ to the server. The server then initiates a new column for epoch $= 2$ and the other members can anonymously download the entire commit from the server (by performing the aforementioned identification protocol).

Issues with naive selective downloading. Unfortunately, if selective downloading is naively applied, the above method leaks the access pattern of group members.

Recall that when selective downloading is performed in $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$, a member sends an index and receives the corresponding member-dependent commit $\widehat{c}_{\mathsf{index}}$ from the server. When a member selectively downloads commit messages relative to two distinct epochs, they send the same index for both epochs. The server can infer that both requests were made by the same party, contradicting Item (3).

Even worse, suppose that the group secret key, and thus, the group member list is compromised. Although the group secret key may heal after PCS of $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ kicks in, the access pattern will never heal; the server who learned the member-index correspondence can permanently break *anonymity* when selective downloading is performed by simply looking at the same index used at every epoch, contradicting Items (2^+) and (5).

Commits with *oblivious* selective downloading. While the problem exposed above is known to be solvable using relatively complex tools such as private information retrieval (PIR) [Cho+95], we provide a much simpler solution using a pseudorandom permutation PRP by taking advantage of the fact that selective downloading is performed by each group member *once per epoch*. Continuing with our above example, when id_1 generates a commit $(c_0, \vec{c} = (\widehat{c}_i)_{i\in[3]})$, it further *deterministically* generates a PRP key $\mathsf{permKey} \leftarrow \mathsf{PRF}(\mathsf{k}_1, \text{'perm'})$ which defines a permutation over the group size, which is $[3] = \{1, 2, 3\}$ for our example. It then creates a permuted member dependent commit $\vec{c}_{\mathsf{perm}}$ so that $\widehat{c}_i$ is placed at entry $\mathsf{PRP}(\mathsf{permKey}, i) \in [3]$. Finally, id_1 uploads $(\mathsf{gvk}_2, (c_0, \vec{c}_{\mathsf{perm}}))$ to the server by performing the identification protocol using gsk_1.

When id_2 performs a selective download to retrieve the proposal and (partial) commit from the server, it computes its permuted $\mathsf{index} = \mathsf{PRP}(\mathsf{permKey}, 2)$, where id_2 generates an identical $\mathsf{permKey}$ as id_1. It performs an identification protocol using gsk_1, sends index, and retrieves $(c_0, \widehat{c}_{\mathsf{index}})$ and the proposal p_3 from the server. This is illustrated in the right half of Figure 5.35. id_1 can then invoke $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ on input $(\texttt{Process}, (c_0, \widehat{c}_{\mathsf{index}}), p_3)$ and move to the next $\mathsf{epoch} = 2$.

Observe that a member never performs a selective download more than once per epoch. This is the main reason why a PRP suffices — the access would have been linkable if selective downloading was performed more than twice per epoch using the same PRP key. Moreover, since the group secret key is updated at each epoch, the PRP key is also updated, thus satisfying FS and PCS (Item (5)).

Remark 5.5.1 (Non-Interactive Membership Identification). We provided a challenge-response type *interactive* identification protocol to prove group membership. By allowing the server to perform additional checks on the database and further reasonably weakening the security guarantee (i.e., Item (4) is guaranteed only for uploads), we are able to make the protocol completely *non-interactive*. At a high level, the party simply needs to sign the proposal or commit (rather than a challenge message) to upload, and perform no membership identification to download. The full detail is provided in Section 5.5.4.

5.5.3 Full Description of the Wrapper Protocol W^{mh}

In this section, we propose a metadata CGKA W^{mh} and prove that it UC-realizes $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ in the $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$-hybrid model. We call W^{mh} as a *wrapper* protocol since it works as a wrapper around any static metadata-hiding CGKA that UC-realizes $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$, and turns it into full metadata-hiding CGKA. In particular, the sole functionality of W^{mh} is to take care of how the proposals, commits, and welcome messages are uploaded and downloaded from the server in a dynamic metadata-hiding manner. The construction is very simple and can be implemented only from a standard signature scheme.

$G.\mathsf{gid}$	The identifier of the group.
$G.\mathsf{epoch}$	The current epoch number.
$G.\mathsf{mem}$	A list of $(\mathsf{id}, \mathsf{svk})$-pair. The list is sorted in lexicographic order by ids.
$(G.\mathsf{gsk}, G.\mathsf{gvk})$	The group signature key used to authenticate group membership to the server.
$G.\mathsf{permKey}$	The PRP key used to permute member index in membership list.
$G.\mathsf{indexOf}(\mathsf{id})$	Returns the index of id in the list $G.\mathsf{mem}$.

TABLE 5.6: The party's protocol state and helper method of W^{mh}.

$\mathsf{PropDB}[*,*]$	It stores a list of proposal message p issued at $(\mathsf{gid}, \mathsf{epoch})$.
$\mathsf{ComDB}[*,*]$	It stores a group signature key gvk used to authenticate membership on $(\mathsf{gid}, \mathsf{epoch})$ and possibly a commit message $(c_0, \vec{c})$ to move to $(\mathsf{gid}, \mathsf{epoch})$.
$\mathsf{WelDB}[*]$	It stores a welcome message $\widehat{w}$ for id.

TABLE 5.7: The server's protocol state.

Protocol States. Each party holds a group state G. It consists of the variables listed in Table 5.6. The $G.\mathsf{mem}$ list stores the group member's identity and its signature key. The list is sorted in lexicographic order by the party's identity. Parties can fetch the index of their identities in $G.\mathsf{mem}$ via the method $G.\mathsf{indexOf}(*)$.

The server keeps three databases: $\mathsf{PropDB}[*,*]$, $\mathsf{ComDB}[*,*]$, and $\mathsf{WelDB}[*]$. $\mathsf{PropDB}[*,*]$ and $\mathsf{ComDB}[*,*]$ both use $(\mathsf{gid}, \mathsf{epoch})$ as indices to store proposals and commits, respectively. We refer the readers to Section 5.5.2 for a pictorial example, where we merge the PropDB and ComDB into one database in the figures.

- $\mathsf{PropDB}[\mathsf{gid}, \mathsf{epoch}] = \vec{p}$: The proposal database stores the list of proposals $\vec{p}$ created at $(\mathsf{gid}, \mathsf{epoch})$. These proposals, once committed, are used to move any party at epoch to the next $\mathsf{epoch}' = \mathsf{epoch} + 1$, where the group state is updated accordingly to the proposals.
- $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] = (\mathsf{gvk}, c_0, \vec{c})$ or $(\mathsf{gvk}, \bot, \bot)$: The commit database stores a group signing key (also called a group statement) gvk. This will be used by a group member at epoch to anonymously prove that they are indeed a group member. $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}]$ is initialized with $(\mathsf{gvk}, \bot, \bot)$ and is later updated to $(\mathsf{gvk}, c_0, \vec{c})$ when some party creates a commit $(c_0, \vec{c})$ with the proposals stored in $\mathsf{PropDB}[\mathsf{gid}, \mathsf{epoch}]$. Once $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}]$ has a commit stored, then no other commits can be made at this epoch.
- $\mathsf{WelDB}[\mathsf{id}] = \widehat{w}$: It stores a welcome message for id. We assume $\mathsf{WelDB}[\mathsf{id}]$ stores only one welcome message for id. This is due to the fact that previous CGKA UC-security models assume a party can join at most one group (i.e., $\mathsf{Ptr}[\mathsf{id}]$ identifies the unique group that id is a member of). In the proposed protocol, when the server receives a new welcome message $(\mathsf{id}, \widehat{w})$, it overwrites it as $\mathsf{WelDB}[\mathsf{id}] := \widehat{w}$. We emphasize that this restriction is required only to prove UC-security, and functionality-wise, $\mathsf{WelDB}[\mathsf{id}]$ can store as many welcome messages as it needs.

Protocol Algorithms. The main interface and the associated helper functions are depicted in Figure 5.36. The protocol W^{mh} is defined in the $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$-hybrid model and internally calls $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ to create a group, generate and process protocol messages. To register groups and publish or fetch protocol messages, it uses the subroutines depicted in Figures 5.37 to 5.40.

Group Creation. A group can be created by the group creator $\mathsf{id}_{\mathsf{creator}}$ by invoking $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{mh}}$ on input $(\texttt{Create}, \mathsf{svk})$. The group creator first invokes $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ with the same input. Then, it runs the helper function `*init-states` to initialize the group state of the wrapper protocol. Note that `*init-states` internally queries $(\mathtt{Key}_{\mathtt{mh}})$ to $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ to use the group secret k_{mh}. Finally, it executes the group registration protocol `RegisterGroup` shown in Figure 5.37. The group creator sends the new group's group identity gid, epoch counter epoch, and group signature key gvk via a client-anonymous authenticated channel. Upon receiving the group creation message, the server checks that a group with the same identity does not exist and the epoch counter is equal to 0. If the check passes, the server stores $(\mathsf{gvk}, \bot, \bot)$ in $\mathsf{ComDB}[\mathsf{gid}, 0]$. Finally, the server notifies the party of the success or failure of the group creation. The group creator checks the protocol result. If group creation fails, it unwinds all state changes and outputs $\bot$.

Proposal. A party located at $(\mathsf{gid}, \mathsf{epoch})$ can generate a proposal message p by invoking $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{mh}}$ on input $(\texttt{Propose}, \mathsf{act})$. Then, it executes the publish proposal protocol `PublishProposal` shown in Figure 5.38, where party accesses the server via a client-anonymous authenticated channel. The party and the server perform a challenge-response type *membership authentication protocol*, that allows a party to anonymously prove that it is a valid group member for $(\mathsf{gid}, \mathsf{epoch})$. In more detail, upon receiving the `PublishProposal` proposal message, the server chooses a random challenge message $\mathsf{ch} \leftarrow \{0,1\}^{\kappa}$ and sends it to the party. The party then signs the challenge with its group signing key $G.\mathsf{ssk}$ and sends the signature σ along with the destination $(\mathsf{gid}, \mathsf{epoch})$ and the proposal p. The server checks that $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] = (\mathsf{gvk}, \bot, \bot)$ and the signature σ is valid with respect to gvk. If the check passes, the server stores the proposal p in $\mathsf{PropDB}[\mathsf{gid}, \mathsf{epoch}]$. Note that the server rejects the proposal if $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}]$ contains a commit since this indicates that a new epoch has been created. We later show in Section 5.5.4 that the above membership authentication protocol can be made non-interactive if we allow the server to perform an additional simple check on the database.

Commit. A party located at $(\mathsf{gid}, \mathsf{epoch})$ can perform a commit by invoking $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{mh}}$ on input $(\texttt{Commit}, \mathsf{svk})$. The party first executes the subroutine `FetchProposals` shown in Figure 5.38 to download the list of proposals created in $(\mathsf{gid}, \mathsf{epoch})$. `FetchProposals` consists of a challenge-response type membership authentication protocol almost identical to `PublishProposal` explained above. Once the server accepts, it is convinced that the calling anonymous party is indeed a valid group member at epoch so it sends the list of proposals $\vec{p}$ stored in the database $\mathsf{PropDB}[\mathsf{gid}, \mathsf{epoch}]$.

Once the party succeeds in fetching the proposals $\vec{p}$, it generates a commit and welcome messages $(c_0, \vec{c}, \vec{w})$ by invoking $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ on input $(\texttt{Commit}, \mathsf{svk}, \vec{p})$. The list $\vec{c}$ is then randomly permuted to $\vec{c}_{\mathsf{perm}}$ by `*permute-commit`. This procedure allows for a shuffle of the order of the member for each new epoch and makes the selective downloading performed during `FetchCommit` unlinkable between different epochs. The party then publishes $(c_0, \vec{c}_{\mathsf{perm}}, \vec{w})$ by performing two different uploads. It first executes `PublishCommit` in Figure 5.39 to publish the commit $(c_0, \vec{c}_{\mathsf{perm}})$. The party accesses the server via a client-anonymous authenticated channel and performs the authentication protocol with the server similar to `PublishProposal`. After generating the response signature σ, the party further generates a group signature key $(\mathsf{gvk}', \mathsf{gsk}')$ for the next epoch. This is generated from the group secret $\mathsf{k_{mh}}'$ at the next epoch, which can be obtained by querying $(\texttt{NextKey}_{\mathtt{mh}}, c_0)$ to $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$. The party finally sends $(\sigma, \mathsf{gid}, \mathsf{epoch}, c_0, \vec{c}_{\mathsf{perm}}, \mathsf{gvk}')$ to the server. If the signature σ is a valid signature with respect to the group signing key stored in $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] = (\mathsf{gvk}, \bot, \bot)$, then it updates the database by $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] \leftarrow (\mathsf{gvk}, c_0, \vec{c}_{\mathsf{perm}})$. In case a commit was already stored, the server rejects the commit. Moreover, the server initializes a new entry in the database as $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}+1] \leftarrow (\mathsf{gvk}', \bot, \bot)$. This creates a new epoch to which the parties can upload new proposals. The server returns whether it succeeded or not to the party.

If the party succeeds to publish the commit and welcome messages exit (i.e., $\vec{w} \neq \emptyset$), it then further executes `PublishWelcome` shown in Figure 5.40 for each $\widehat{w} \in \vec{w}$. The party accesses the server via a client-anonymized authenticated channel and uploads $(\mathsf{id}_t, \widehat{w})$, where id_t is extracted from $\widehat{w}$. The server stores $\widehat{w}$ in $\mathsf{WelDB}[\mathsf{id}_t]$. Here, there is no membership authentication protocol.

Process. A party located at $(\mathsf{gid}, \mathsf{epoch})$ can try to process the current commit and proposals by invoking $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{mh}}$ on input $(\texttt{Process})$. The party first executes the `FetchCommit` subroutine shown in Figure 5.39 to download a commit and their associating proposals. The `FetchCommit` protocol is similar to the `FetchProposals` protocol. The party and server engage in the membership authentication protocol explained above. Notably, the party sends an index that specifies the index of the party-dependent commit the party wants to download. Since the party sends an epoch-dependent permuted index using the function `*permuted-commit-index`, the index from different epochs remain unlinkable. If the server accepts the

party, it returns the list of proposals $\vec{p}$ stored on the database $\mathsf{PropDB}[\mathsf{gid}, \mathsf{epoch}]$, the party independent commit c_0, and the party dependent commit $\widehat{c} := \vec{c}_{\mathsf{perm}}[\mathsf{index}]$ stored on the database $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}]$.

Upon fetching the content $(c_0, \widehat{c}, \vec{p})$, the party processes them by invoking $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ on input $(c_0, \widehat{c}, \vec{p})$. Finally, it also updates the internal state by calling `*update-states` function.

Join. A party can join a group by invoking $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{mh}}$ on input $(\texttt{Join})$. The party first executes the `FetchWelcome` subroutine shown in Figure 5.40 to download a welcome message. The party accesses the server via a standard authenticated channel (i.e., the party discloses its identity id). This is necessary for the server to identify which welcome message to provide. The server either returns the welcome message $\widehat{w}$ designated to id or notifies that there is no welcome message. Once the party receives the welcome message $\widehat{w} \neq \bot$, it processes $\widehat{w}$ by invoking $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$ on input $(\texttt{Join}, \widehat{w})$. Finally, it setups the initial state by calling `*init-states` function.

5.5.4 Making Membership Authentication Protocol Non-Interactive

We discuss two simple ways to make the membership authentication protocol ran to publish and fetch the contents from the server non-interactive.

Recall that when the group member tries to access the server, they always engage in a membership authentication protocol to prove anonymously to the server that they are indeed a valid group member. Our W^{mh} protocol realized this membership authentication protocol in an *interactive* challenge-response type manner using a digital signature scheme. The server sends a challenge and asks the group member to sign the challenge. Assuming that the same challenge is never reused, this allowed the server to be convinced that the communicating party indeed has a group signing key.

We can remove this interaction between the server in some scenarios by asking the server to perform an extra check on the database. Namely, during the membership authentication protocol run during the protocols `PublishProposal` and `PublishCommit`, we allow the parties to sign on to the content they wish to upload to the server, rather than being provided a challenge from the server. For instance, in the protocol `PublishProposal`, the party id signs the message $(\mathsf{gid}, \mathsf{epoch}, p)$. The server checks if the signature is valid with respect to the group signing key at $(\mathsf{gid}, \mathsf{epoch})$, and additionally checks if such p is included in the proposal database $\mathsf{PropDB}[\mathsf{gid}, \mathsf{epoch}]$. If all check passes, it updates the database with p. Notice that the server did not need to check if p was in the database in the previous interactive protocol. This non-interactive variant securely realizes the desired functionality since due to the EUF-CMA security of the signature scheme, a non-group member cannot upload any proposals that haven't been signed. The only possible attack would be to resend the observed signature-proposal pair to the server. However, since the server is modified to never accept the same proposal, this attack fails. The same idea can be used to make the protocol `PublishCommit` non-interactive.

Unfortunately, it is not clear how to make the protocols `FetchProposals` and `FetchCommit` non-interactive using the above idea. This is because the party does not have any contents to sign when it is performing a fetch/download. A potential idea is to weaken our UC-security model to allow non-group members to fetch/download the contents from the server, while still disallowing them to publish/upload any contents on the server. In particular, the protocols `FetchProposals` and `FetchCommit` will simply consist of a party querying the server $(\mathsf{gid}, \mathsf{epoch})$, and the server responding with the proposals and commits on the database without checking membership. This seems like a reasonable compromise considering that the contents uploaded on the server can be provided to an adversary without compromising the security of the group — this follows since the contents are commits and proposals generated from $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$, which by definition secures the group secret key and static metadata. One possible issue is that without any authentication on the fetch/download, it will allow an adversary to learn if a group with the

group identifier gid exists. Our interactive protocol does not leak such information since it will output $\texttt{accept} = \texttt{false}$ when authentication fails.

In summary, by reasonably weakening the ideal functionality $\mathcal{F}_{\text{CGKA}}^{\mathsf{mh}}$ in a meaningful way, we can make the wrapper protocol W^{mh} fully non-interactive. It remains an interesting future work to investigate whether this weakening has any practical impact on the protocol.

Input $(\texttt{Create}, \mathsf{svk})$

1 : **req** $G = \bot$
2 : **try** query $(\texttt{Create}, \mathsf{svk})$ to $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$
3 : $G \leftarrow \texttt{*init-states}(\mathsf{gid}, 0, \{ (\mathsf{id}_{\text{creator}}, \mathsf{svk}) \})$
4 : **req** $\texttt{RegisterGroup}(\mathsf{gid}, 0)$

Input $(\texttt{Propose}, \mathsf{act}^{\dagger})$

1 : **req** $G \neq \bot$
2 : **try** $p \leftarrow$ query $(\texttt{Propose}, \mathsf{act})$ to $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$
3 : **req** $\texttt{PublishProposal}(G.\mathsf{gid}, G.\mathsf{epoch}, p)$
4 : **return** p

Input $(\texttt{Commit}, \mathsf{svk})$

1 : **req** $G \neq \bot$
2 : $(\texttt{accept}, \vec{p}) \leftarrow \texttt{FetchProposals}(G.\mathsf{gid}, G.\mathsf{epoch})$
3 : **req** $\texttt{accept}$
4 : **try** $(c_0, \vec{c}, \vec{w}) \leftarrow$ query $(\texttt{Commit}, \mathsf{svk}, \vec{p})$ to $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$
5 : $\vec{c}_{\mathsf{perm}} \leftarrow \texttt{*permute-commit}(G, \vec{c})$
6 : **try** $\texttt{PublishCommit}(G.\mathsf{gid}, G.\mathsf{epoch}, c_0, \vec{c}_{\mathsf{perm}})$
7 : **foreach** $\widehat{w} \in \vec{w}$ **do**
8 : **parse** $(\mathsf{id}_t, *) \leftarrow \widehat{w}$
9 : **try** $\texttt{PublishWelcome}(\mathsf{id}_t, \widehat{w})$
10 : **return** $(c_0, \vec{c}_{\mathsf{perm}}, \vec{w})$

Input $\texttt{Process}$

1 : **req** $G \neq \bot$
2 : $\mathsf{index}_{\widehat{c}} \leftarrow \texttt{*permuted-commit-index}(G, \mathsf{id})$
3 : **try** $(c_0, \widehat{c}, \vec{p}) \leftarrow \texttt{FetchCommit}(G.\mathsf{gid}, G.\mathsf{epoch}, \mathsf{index}_{\widehat{c}})$
4 : **try** $(\mathsf{id}_c, \mathsf{propSem}, \mathsf{mem})$
$\leftarrow$ query $(\texttt{Process}, c_0, \widehat{c}, \vec{p})$ to $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$
5 : $G' \leftarrow \texttt{*update-states}(G, \mathsf{mem})$
6 : **return** $(\mathsf{id}_c, \mathsf{propSem}, \mathsf{mem})$

Input $(\texttt{Key})$

1 : **req** $G \neq \bot$
2 : **try** $\mathsf{k} \leftarrow$ query $(\texttt{Key})$ to $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$
3 : **return** k

Input $\texttt{Join}$

1 : **req** $G = \bot$
2 : **try** $\widehat{w} \leftarrow \texttt{FetchWelcome}(\mathsf{id})$
3 : **try** $(\mathsf{id}_c, \mathsf{gid}, \mathsf{epoch}, \mathsf{mem})$
$\leftarrow$ query $(\texttt{Join}, \widehat{w})$ to $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$
4 : $G \leftarrow \texttt{*init-states}(\mathsf{gid}, \mathsf{epoch}, \mathsf{mem})$
5 : **return** $(\mathsf{id}_c, \mathsf{gid}, \mathsf{epoch}, \mathsf{mem})$

$\texttt{*init-states}(\mathsf{gid}, \mathsf{epoch}, \mathsf{mem})$

1 : $G.\mathsf{gid} \leftarrow \mathsf{gid}; G.\mathsf{epoch} \leftarrow \mathsf{epoch}; G.\mathsf{mem} \leftarrow \mathsf{mem}$
2 : $\mathsf{k}_{\mathsf{mh}} \leftarrow$ query $\texttt{Key}_{\texttt{mh}}$ to $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$
3 : $G.\mathsf{permKey} \leftarrow \mathsf{PRF}(\mathsf{k}_{\mathsf{mh}}, \text{'perm'})$
4 : $\mathsf{authKey} \leftarrow \mathsf{PRF}(\mathsf{k}_{\mathsf{mh}}, \text{'auth'})$
5 : $(G.\mathsf{gvk}, G.\mathsf{gsk}) \leftarrow \mathsf{SIG}'.\mathsf{KeyGen}(\mathsf{pp}_{\mathsf{SIG}'}; \mathsf{authKey})$
6 : **return** G

$\texttt{*update-states}(G, \mathsf{mem})$

1 : $G'.\mathsf{gid} \leftarrow G.\mathsf{gid}; G'.\mathsf{epoch} \leftarrow G.\mathsf{epoch} + 1$
2 : $G'.\mathsf{mem} \leftarrow \mathsf{mem}$
3 : $\mathsf{k}_{\mathsf{mh}} \leftarrow$ query $\texttt{Key}_{\texttt{mh}}$ to $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$
4 : $G'.\mathsf{permKey} \leftarrow \mathsf{PRF}(\mathsf{k}_{\mathsf{mh}}, \text{'perm'})$
5 : $\mathsf{authKey} \leftarrow \mathsf{PRF}(\mathsf{k}_{\mathsf{mh}}, \text{'auth'})$
6 : $(G'.\mathsf{gvk}, G'.\mathsf{gsk}) \leftarrow \mathsf{SIG}'.\mathsf{KeyGen}(\mathsf{pp}_{\mathsf{SIG}'}; \mathsf{authKey})$
7 : **return** G'

$\texttt{*permute-commit}(G, \vec{c})$

1 : $\vec{c}_{\mathsf{perm}} \leftarrow ()$
2 : **for** $\mathsf{index} = 1, \ldots, |\vec{c}|$ **do**
3 : $\vec{c}_{\mathsf{perm}} \mathbin{+\!\!\leftarrow} \vec{c}[\mathsf{PRP}(G.\mathsf{permKey}, \mathsf{index})]$
4 : **return** $\vec{c}_{\mathsf{perm}}$

$\texttt{*permuted-commit-index}(G, \mathsf{id})$

1 : $\mathsf{index} \leftarrow G.\mathsf{indexOf}(\mathsf{id})$
2 : **return** $\mathsf{PRP}(G.\mathsf{permKey}, \mathsf{index})$

FIGURE 5.36: Metadata-hiding CGKA protocol W^{mh} in the $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$-hybrid model: `Create`, `Propose`, `Commit`, `Process`, `Join`, `Key`, and some helper functions. †: $\mathsf{act} \in \{ \text{'upd'-svk}, \text{'add'-id}_t\text{-kp}_t, \text{'rem'-id}_t \}$.

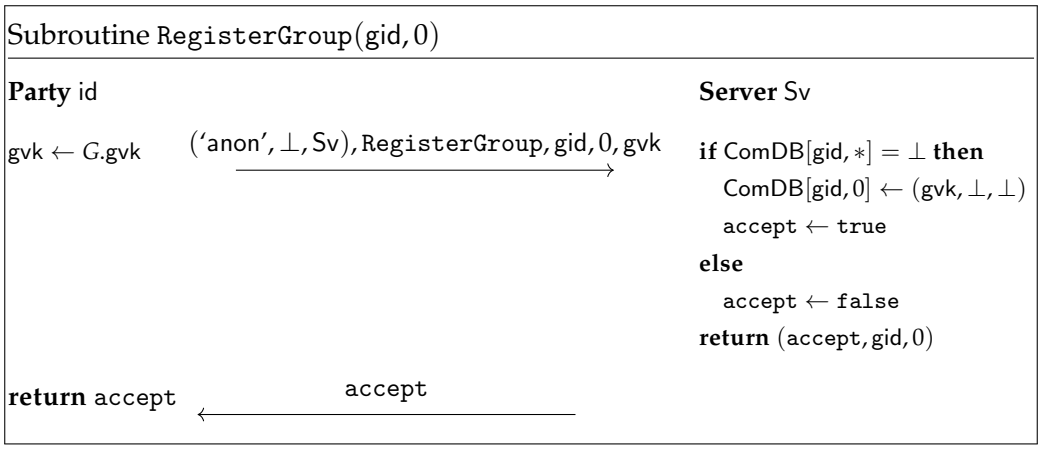


FIGURE 5.37: Subroutines for metadata-hiding CGKA protocol W^{mh} in the $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$-hybrid model: Register a new group and initialize group states for the current epoch. Party id and the server Sv are connected via a client-anonymized authenticated channel.

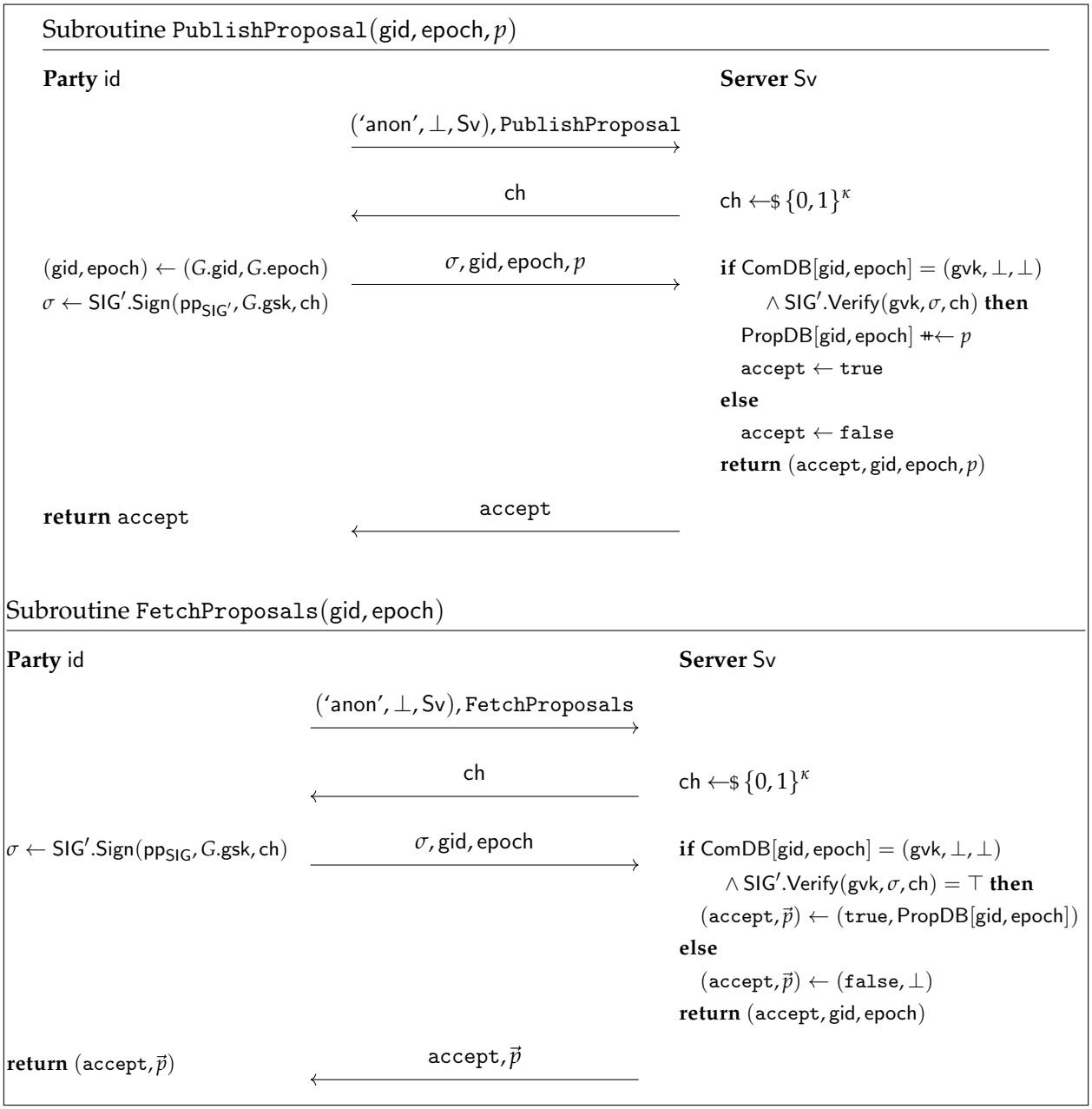


FIGURE 5.38: Subroutines for metadata-hiding CGKA protocol W^{mh} in the $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$-hybrid model: Publish and fetch proposal messages. Party id and the server Sv are connected via a client-anonymized authenticated channel.

Subroutine $\texttt{PublishCommit}(\mathsf{gid}, \mathsf{epoch}, c_0, \vec{c})$

Party id		**Server** Sv
	$(\text{'anon'}, \bot, \mathsf{Sv}), \texttt{PublishCommit}$ →	
	← ch	$\mathsf{ch} \leftarrow\$ \{0,1\}^\kappa$
$(\mathsf{gid}, \mathsf{epoch}) \leftarrow (G.\mathsf{gid}, G.\mathsf{epoch})$	$\sigma, \mathsf{gid}, \mathsf{epoch}, \mathsf{gvk}', c_0, \vec{c}_{\mathsf{perm}}$ →	**if** $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] = (\mathsf{gvk}, \bot, \bot)$
$\sigma \leftarrow \mathsf{SIG}'.\mathsf{Sign}(\mathsf{pp}_{\mathsf{SIG}'}, G.\mathsf{gsk}, \mathsf{ch})$		$\wedge\ \mathsf{SIG}'.\mathsf{Verify}(\mathsf{gvk}, \sigma, \mathsf{ch}) = \top$ **then**
// Generate the next epoch's group statement.		$\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] \leftarrow (\mathsf{gvk}, c_0, \vec{c}_{\mathsf{perm}})$
$\mathsf{k_{mh}}' \leftarrow$ query $(\texttt{NextKey}_{\texttt{mh}}, c_0)$ to $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$		$\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}+1] \leftarrow (\mathsf{gvk}', \bot, \bot)$
$\mathsf{authKey}' \leftarrow \mathsf{PRF}(\mathsf{k_{mh}}', \text{'auth'})$		$\texttt{accept} \leftarrow \texttt{true}$
$(\mathsf{gvk}', \mathsf{gsk}') \leftarrow \mathsf{SIG}'.\mathsf{KeyGen}(\mathsf{pp}_{\mathsf{SIG}'}; \mathsf{authKey}')$		**else**
		$\texttt{accept} \leftarrow \texttt{false}$
		return $(\texttt{accept}, \mathsf{gid}, \mathsf{epoch}, c_0, \vec{c}_{\mathsf{perm}})$
return accept	← accept	

Subroutine $\texttt{FetchCommit}(\mathsf{gid}, \mathsf{epoch}, \mathsf{index})$

Party id		**Server** Sv
	$(\text{'anon'}, \bot, \mathsf{Sv}), \texttt{FetchCommit}$ →	
	← ch	$\mathsf{ch} \leftarrow\$ \{0,1\}^\kappa$
$\sigma \leftarrow \mathsf{SIG}'.\mathsf{Sign}(\mathsf{pp}_{\mathsf{SIG}'}, G.\mathsf{gsk}, \mathsf{ch})$	$\sigma, \mathsf{gid}, \mathsf{epoch}, \mathsf{index}$ →	**if** $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] = (\mathsf{gvk}, c_0, \vec{c}_{\mathsf{perm}})$
		$\wedge\ \mathsf{SIG}'.\mathsf{Verify}(\mathsf{gvk}, \sigma, \mathsf{ch}) = \top$ **then**
		$(\texttt{accept}, \vec{p}) \leftarrow (\texttt{true}, \mathsf{PropDB}[\mathsf{gid}, \mathsf{epoch}])$
		$\widehat{c} \leftarrow \vec{c}_{\mathsf{perm}}[\mathsf{index}]$
		else
		$(\texttt{accept}, c_0, \widehat{c}, \vec{p}) \leftarrow (\texttt{false}, \bot, \bot, \bot)$
		return $(\texttt{accept}, \mathsf{gid}, \mathsf{epoch}, \mathsf{index})$
return $(\texttt{accept}, c_0, \widehat{c}, \vec{p})$	← $\texttt{accept}, c_0, \widehat{c}, \vec{p}$	

FIGURE 5.39: Subroutines for metadata-hiding CGKA protocol W^{mh} in the $\mathcal{F}_{\mathrm{CGKA}}^{\mathsf{ctxt}}$-hybrid model: Publish and fetch commit messages. Party id and the server Sv are connected via a client-anonymized authenticated channel.

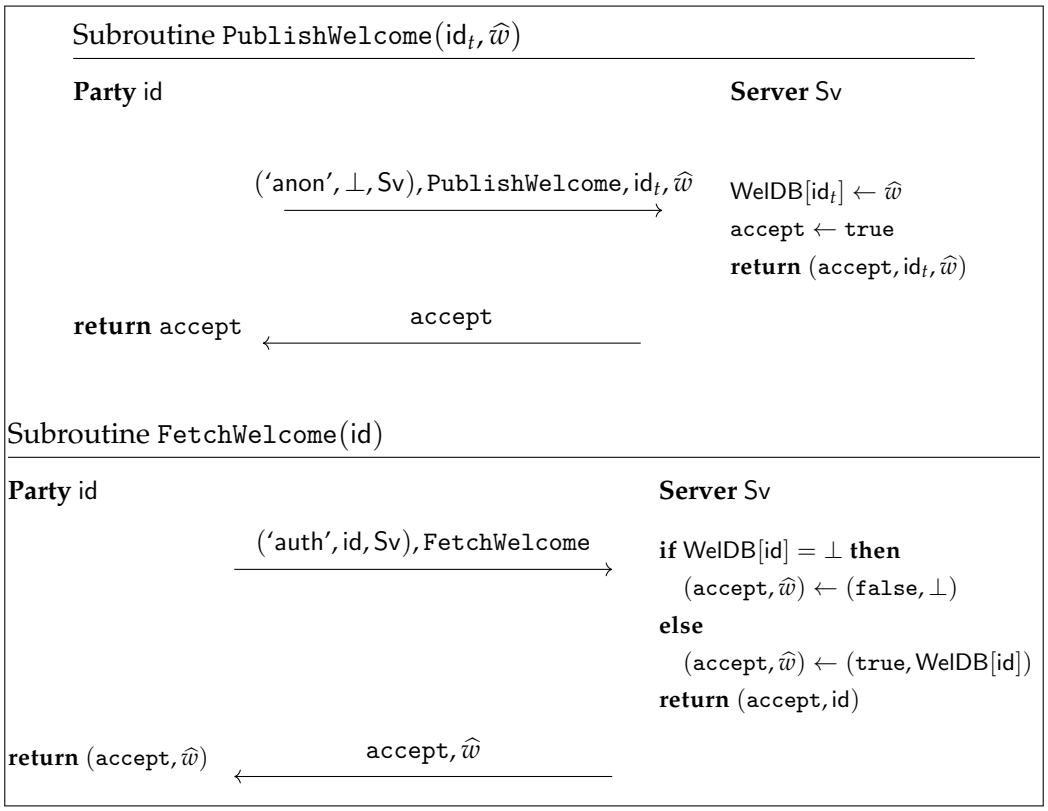


FIGURE 5.40: Subroutines for metadata-hiding CGKA protocol W^{mh} in the $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$-hybrid model: Publish and fetch welcome messages. Party id and the server Sv are connected via a client-anonymized authenticated channel.

Random index and group authentication.
random-index$(c) \iff$ **safe**(c)
adv-access-allowed$(c) \iff \neg$**safe**(c)

FIGURE 5.41: Additional safety predicates unique to the wrapper protocol W^{mh}. The other safety predicates used to implicitly define the underlying ideal functionality $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ are identical to those provided in Section 5.4.3.

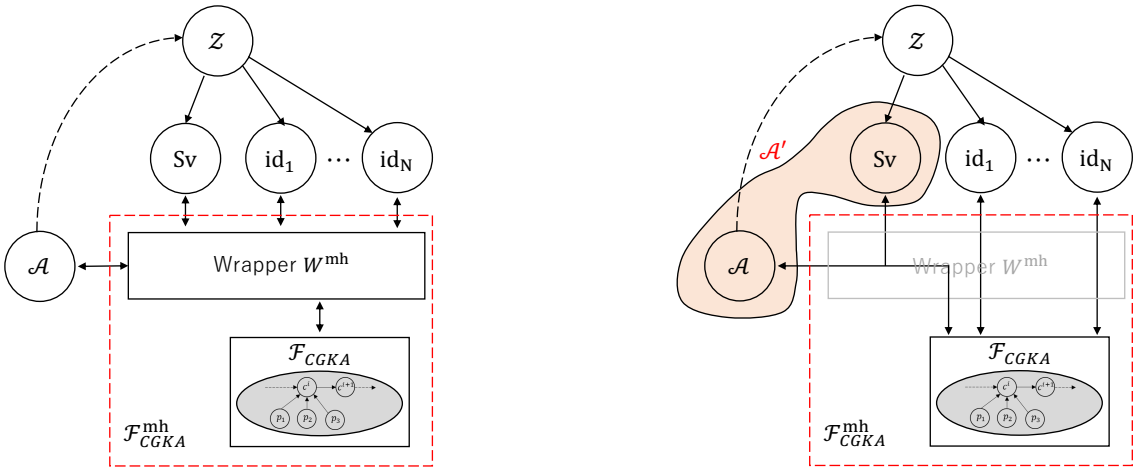


FIGURE 5.42: (Left) metadata-hiding CGKA $\Pi^{\mathsf{mh}}_{\mathrm{CGKA}}$ when the server Sv is honest and (Right) when Sv is malicious. The red dotted box denotes the entire protocol $\Pi^{\mathsf{mh}}_{\mathrm{CGKA}}$, where it is further decomposed as a combination of the wrapper protocol W^{mh} and ideal functionality $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$. The red shaded region denotes that Sv is corrupted and that $(\mathcal{A}, \mathsf{Sv})$ are viewed as a single adversary $\mathcal{A}'$. In this case, W^{mh} is ignored and $\Pi^{\mathsf{mh}}_{\mathrm{CGKA}}$ degenerates to a UC-realization of $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$.

5.6 Metadata-Hiding CGKA: Define UC Security Model

We define a UC security model capturing the security of the entire 1st, 2nd & 3rd layers (i.e., group secret keys, static and *dynamic* metadata) by defining a new ideal functionality $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$. Any CGKA UC-realizing $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ is provably a *metadata-hiding* CGKA.

Reusing most of the description of $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ handling the 1st and 2nd layers, the description of $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ can focus mainly on the 3rd layer. Our model succinctly captures all the properties explained in Section 5.5.1, Items (1) to (5). We then show that the wrapper protocol W^{mh} presented in the previous section UC-realizes $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ in the $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$-hybrid model. The full details of this section is provided in Section 5.6.2. Below, we provide an overview of our idea.

5.6.1 Overview of the Proposed Ideal Functionality

Modeling an Honest but Curious Sever.

In previous constructions of CGKA, the server was assumed to be *always* malicious. This is for instance captured in the ideal functionalities $\mathcal{F}_{\mathrm{CGKA}}$ and $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ by observing that the `Commit` and `Process` take as input arbitrary proposals and commits — not just those created by the honest group members.

While assuming the server to be always malicious allows to capture a strong level of security against group secret keys and static metadata, this is far too inflexible for our use case. Recall Section 5.5.1, Item (4). To properly model that any non-member cannot upload and download from the server on behalf of the group, we must model an *honest but curious* server — a server that honestly follows the protocol but tries to learn as much metadata as possible.

To this end, we explicitly incorporate a server into our model as depicted in Figure 5.42. We allow the server to be in two states: honest[34] or corrupt. When the server is honest, we are able to properly model

[34]We drop "but curious" for simplicity.

Section 5.5.1, Item (4). Otherwise, since a malicious server can arbitrarily choose to accept or reject the identification protocol executed by the wrapper protocol W^{mh}, W^{mh} does not provide any meaningful functionality. In particular, our metadata-hiding CGKA $\Pi^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ degenerates to offer the same functionality as $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$.

UC Security Model for Dynamic Metadata. As already mentioned, the new ideal functionality $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ inherits all functionalities offered by $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$. $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ comes with seven additional functionalities:

- `RegisterGroup`,
- `PublishProposal`, `FetchProposals`,
- `PublishCommit`, `FetchCommit`,
- `PublishWelcome`, `FetchWelcome`.

As the name indicates, `Publish-`$*$ (resp. `Fetch-`$*$) is invoked to upload (resp. download) a proposal, commit, or welcome message from the server. These functions are mainly invoked during an execution of `Create`, `Propose`, `Commit`, `Process`, and `Join`. For instance, when `Commit` is invoked, the member first runs `FetchProposals` to retrieve the proposals $\vec{p}$ from the server and invokes $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ on input $(\texttt{Commit}, \vec{p})$. It then uploads the commit and welcome message output by $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ using `PublishCommit`.

These functions are defined differently depending on (i) whether the calling party is an honest group member or an adversary,[35] and (ii) whether the server is honest or malicious. As explained above, if the server is malicious, then we let the adversary $\mathcal{A}$ (i.e., server) decide whether `Publish-`$*$ or `Fetch-`$*$ succeeds. In this case, $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ becomes functionally identical to $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$, modulo some syntactical difference due to the inclusion of the server into the model.

Otherwise, if the server is honest, then $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ captures the correctness and security guarantees. For correctness, if the calling party is a group member and the group statement for the next epoch was honestly generated, then the functionality demands the server to accept the upload or download. On the other hand, for security, if the calling party is the adversary, then we require the server to reject the upload or download as long as the predicate **safe** at that epoch is true. That is, if the group secret key is not compromised, then the adversary should not be able to upload and download on behalf of the group.

Additionally, the database stored by the server (see Figures 5.33 to 5.35) is modeled in $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ by two lists PropDB and ComDB, each maintaining the proposals and commit for group gid at epoch. $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ also models the permutation-based selective downloading explained in Section 5.5.2 by an ideal (helper) function `*permute-commit`. Finally, due to the already complex nature of the metadata-hiding CGKA, we did not model adversarially controlled randomness in $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$. We leave this as an important future work.

Proof of Dynamic Metadata-Hiding CGKA. We prove that the wrapper protocol W^{mh} UC-realizes the metadata-hiding ideal functionality $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ in the $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$-hybrid model. The full proof is provided in Section 5.6.3.

The proof is relatively simple and modular since we defined W^{mh} in the $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$-hybrid model. We can in essence ignore all the description of $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ that relates to the 1st and 2nd layers' correctness and security since the same checks can be handled by the simulator $\mathcal{S}$ internally simulating $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$. Specifically, our proof only needs to focus on the 3rd layer of correctness and security. The proof is standard and consists of invoking the security of the pseudorandom permutation and signature scheme.

[35] In our security definition (in Section 5.6.2), we use `Publish-`$*$`-Adv` and `Fetch-`$*$`-Adv` to indicate that the calling party is the adversary.

5.6.2 UC Security Model $\mathcal{F}^{\mathsf{mh}}_{\mathbf{CGKA}}$

In this section, we propose a UC security model capturing the security of the 1st, 2nd, and 3rd layers (i.e., group secret keys, static and *dynamic* metadata) by defining a new ideal functionality $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$.

Overview of $\mathcal{F}^{\mathsf{mh}}_{\mathbf{CGKA}}$. The ideal functionality $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ is formally defined in Figures 5.43 to 5.47 and 5.49 to 5.51, along with the helper functions in Figure 5.48 to aid the readability. As it can be checked, $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ shares a large portion of its code with $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$. This is because $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ by definition also models an ideal functionality of a CGKA securing the 1st & 2nd layers. For better readability, we outsourced the description of $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ that is non-essential to the dynamic metadata to $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$. This should not be misunderstood as $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ making oracle calls to $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ — the former simply reuses the codes of $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$.

While $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ allows the adversary to corrupt the parties (and the server) as in $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$, we do not model adversarial controlled randomness (i.e., Rand is always set to 'good'). As the first work to formally capture the dynamic metadata layer, we opted for simplicity and better readability. We leave it as future work to allow such types of attacks. We now explain the ideal functionality $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ in more detail.

States. Similarly to $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$, $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ maintains a history graph. As a custom, we assume one honest group is created by the designated party $\mathsf{id}_{\mathsf{creator}}$. Such group is assigned to the main root $\mathsf{node\text{-}id} = 0$. Since $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ does not allow the adversary to manipulate the party's randomness, $\mathsf{Rand}[\mathsf{id}]$ cannot be switched to 'bad'.

Other than the history graph, $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ also maintains three databases: $\mathsf{PropDB}[*,*]$, $\mathsf{ComDB}[*,*]$, and $\mathsf{WelDB}[*]$. The proposal database $\mathsf{PropDB}[\mathsf{gid},\mathsf{epoch}]$ stores proposals issued at $(\mathsf{gid},\mathsf{epoch})$. The welcome database $\mathsf{WelDB}[\mathsf{id}]$ stores a welcome message sent to id. Since $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ processes a join query only when $\mathsf{Ptr}[\mathsf{id}] = \perp$ (i.e., a party id is not assigned to any group), $\mathsf{WelDB}[\mathsf{id}]$ stores only one welcome message for each id, and overwrites the old one if a new welcome message is published to id. Finally, the commit database $\mathsf{ComDB}[\mathsf{gid},\mathsf{epoch}]$ stores the status of the group gid at epoch. Depending on the status of the group, it takes one of the following five values:

- $\mathsf{ComDB}[\mathsf{gid},\mathsf{epoch}] = \perp$: It indicates that the epoch has not been initialized yet.
- $\mathsf{ComDB}[\mathsf{gid},\mathsf{epoch}] = (\top, \mathsf{node\text{-}id})$: It indicates that the epochhas been initialized by an honest party located at the commit node $\mathsf{node\text{-}id}$ (of the history graph), and no commit has been issued at epoch.
- $\mathsf{ComDB}[\mathsf{gid},\mathsf{epoch}] = ((c_0,\vec{c}), \mathsf{node\text{-}id})$: It indicates that the epoch has been initialized by an honest party located at the commit node $\mathsf{node\text{-}id}$, and a commit $(c_0,\vec{c})$ has been issued at epoch.
- $\mathsf{ComDB}[\mathsf{gid},\mathsf{epoch}] = (\top, \text{'adv'})$: It indicates that the epoch has been initialized by an adversary, and no commit has been issued at epoch.
- $\mathsf{ComDB}[\mathsf{gid},\mathsf{epoch}] = ((c_0,\vec{c}), \text{'adv'})$:It indicates that the epoch has been initialized by an adversary, and a commit $(c_0,\vec{c})$ has been issued at epoch.

Looking ahead, (assuming the server Sv is honest) $\mathsf{ComDB}[\mathsf{gid},\mathsf{epoch}]$ is initialized with $\perp$. If some party at $\mathsf{epoch}-1$ creates a commit, $\mathsf{ComDB}[\mathsf{gid},\mathsf{epoch}]$ is initialized to $(\top,*)$, where $*$ depends on whether the party was honest or corrupt. Once the proposals in at epoch, i.e., $\mathsf{PropDB}[\mathsf{gid},\mathsf{epoch}]$, are committed, then $\mathsf{ComDB}[\mathsf{gid},\mathsf{epoch}]$ stores that commit and this freezes the epoch, and initializes a new $\mathsf{ComDB}[\mathsf{gid},\mathsf{epoch}+1] = \perp$. This models the fact that a commit is generated only once per epoch when the server is honest. We refer the readers to see Section 5.5.2 for a pictorial example.

Interfaces. $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ offers similar interfaces to the parties as $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$. A party can create a group, create proposal, commit, or welcome messages, process these messages, and obtain group secret keys. In addition

to these functionalities, it also offers the functionalities of publishing and fetching messages, which models the message delivery between a (possibly malicious) party and the server. To formally capture this, $\mathcal{F}_{\text{CGKA}}^{\text{mh}}$ makes the existence of the server explicit — recall that the server was implicit in $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ and abstracted away as a malicious network that injects arbitrary message. Specifically $\mathcal{F}_{\text{CGKA}}^{\text{mh}}$ models an *honest-but-curious* server. When the server is corrupted by the adversary, we end up with the same functionality as $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$.

In the following, we explain each of the functionalities in mode detail.

Functions Used by Legitimate Parties. *Create and register group (See Figure 5.44).* The designated party $\mathsf{id}_{\mathsf{creator}}$ first creates the group as in $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ (cf. *Group creation* in Section 5.3.3). $\mathsf{id}_{\mathsf{creator}}$ then registers the group to the server via the subroutine `RegisterGroup`. $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ first checks that the party $\mathsf{id}_{\mathsf{creator}}$ is located at $\mathsf{epoch} = 0$ (i.e., $\mathsf{Ptr}[\mathsf{id}_{\mathsf{creator}}] = 0$). Then, $\mathcal{F}_{\text{CGKA}}^{\text{mh}}$ informs the adversary that a new group with $(\mathsf{gid}, 0)$ is being registered.

- If the server is honest (i.e., $\texttt{ServerStat} = \text{'good'}$), the ideal server checks that the group with gid has not been registered yet. If so, the server initializes the group as $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] \leftarrow (\top, 0)$[36] and returns $\texttt{accept} = \texttt{true}$ to the party. Else, the registration is rejected and the party receives $\texttt{accept} = \texttt{false}$. The ideal server outputs $(\texttt{accept}, \mathsf{gid}, \mathsf{epoch} = 0)$ (to the environment $\mathcal{Z}$). Since parties are assumed to access the server via a client-anonymous authenticated channel, the server never outputs the accessing party's identity.
- If the server is malicious, the adversary specifies the protocol result $\texttt{accept}'$, and it is outputted to id. This means the malicious server can decide the protocol result arbitrarily.

Create and publish proposals (See Figure 5.44). When a party id create a proposal, $\mathcal{Z}$ invokes $\mathcal{F}_{\text{CGKA}}^{\text{mh}}$ on input $(\texttt{Propose}, \mathsf{act})$. $\mathcal{F}_{\text{CGKA}}^{\text{mh}}$ first creates a proposal message p as in $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ (cf. *Creating proposals* in Section 5.3.3). Then, id publishes p to the server via the subroutine `PublishProposal`.

- If the server is honest (i.e., $\texttt{ServerStat} = \text{'good'}$), $\mathcal{F}_{\text{CGKA}}^{\text{mh}}$ gives $(\mathsf{gid}, \mathsf{epoch}, p)$ to the adversary, where $(\mathsf{gid}, \mathsf{epoch})$ is the destination group identity and epoch. The party accesses the server via the client-anonymous authenticated channel, and therefore, the adversary does not receive the party's identity. The adversary reports whether the protocol succeeds or not by setting $\texttt{accept}'$ to true or false. $\mathcal{F}_{\text{CGKA}}^{\text{mh}}$ then determines the party and server's output as follows.
  - If $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] = (\top, \mathsf{node\text{-}id})$ and $\mathsf{node\text{-}id} = \mathsf{Ptr}[\mathsf{id}]$, the ideal party always outputs $\texttt{accept} := \texttt{true}$. This models correctness: if a party holds the same group state used to initialize $\mathsf{node\text{-}id}$, the server must accept the published proposal.
  - If $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] = (\top, \text{'adv'})$ and $\texttt{accept}' = \texttt{true}$, the ideal party outputs $\texttt{accept} := \texttt{true}$. This models the fact that when the destination $(\mathsf{gid}, \mathsf{epoch})$ is initialized by the adversary, we let the adversary decide if the ideal server should accept the published proposal or not. Looking at the example from Section 5.5.2, if $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}]$ was initialized with a fake group statement gvk, then the adversary can deliberately make the publish proposal fail.
  - Otherwise, the ideal party outputs $\texttt{accept} := \texttt{false}$. This case occurs if $\texttt{accept}' = \texttt{false}$ (i.e., the adversary decides to reject), $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] \neq (\top, *)$ (i.e., the destination has not been initialized or a commit has been issued), or $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] = (\top, \mathsf{node\text{-}id})$ but $\mathsf{node\text{-}id} \neq \mathsf{Ptr}[\mathsf{id}]$ (i.e., the party is located in a different commit node).

[36] The `RegisterGroup` subroutine only occurs when the main group (which is assigned to $\mathsf{node\text{-}id} = 0$) is registered.

The ideal server outputs the destination $(\mathsf{gid}, \mathsf{epoch})$ and the published proposal p to $\mathcal{Z}$. Since parties access the server via a client-anonymous authenticated channel, the server never outputs the accessing party's identity.

- If the server is malicious, $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ gives $\mathsf{Ptr}[\mathsf{id}]$ to the adversary, where $\mathsf{Ptr}[\mathsf{id}] = \mathsf{node\text{-}id}$ is the current commit node on the history graph, which id is located at. The reason why we don't give $(\mathsf{gid}, \mathsf{epoch})$ as above is that when the server is malicious, it can arbitrary fork the group. In such a case, $(\mathsf{gid}, \mathsf{epoch})$ is not enough to identify the location of the party. Note that in the real protocol, the malicious server gets to learn which fork a party is in since the party will try to access the server using the group state. Hence, $\mathsf{Ptr}[\mathsf{id}]$ correctly models what the malicious server learns in the real world. Finally, the adversary specifies the protocol result $\mathtt{accept}'$, and it is outputted to id. This means the malicious server can decide the protocol result arbitrarily.

Create and publish commits (See Figure 5.44). When a party id wants to issue a commit message, $\mathcal{Z}$ invokes $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ on input $(\mathtt{Commit}, \mathsf{svk})$. The description composes of two parts.

Fetch proposals id must first fetch the list of proposals from the server via the subroutine `FetchProposals`.

- If the server is honest (i.e., $\mathtt{ServerStat} = \text{'good'}$), $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ gives $(\mathsf{gid}, \mathsf{epoch})$ to the adversary, where $(\mathsf{gid}, \mathsf{epoch})$ is the destination. Since the party accesses to the server via a client-anonymous authenticated channel, and therefore the server does not receive the party's identity. The adversary reports whether the protocol succeeds or not by setting $\mathtt{accept}'$ to true or false, and specifies the proposals $\vec{p}'$. $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ then determines the party and server's output as follows.
  - If $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] = (\top, \mathsf{node\text{-}id})$ and $\mathsf{node\text{-}id} = \mathsf{Ptr}[\mathsf{id}]$, the ideal party always outputs $\mathtt{accept} := \mathtt{true}$ and $\vec{p} := \mathsf{PropDB}[\mathsf{gid}, \mathsf{epoch}]$. This models correctnes: if a party holds the same group state used to initialize node-id, the server must accept and returns the proposals stored on the database.
  - If $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] = (\top, \text{'adv'})$ and $\mathtt{accept}' = \mathtt{true}$, the ideal party outputs $\mathtt{accept} := \mathtt{true}$ and $\vec{p} := \mathsf{PropDB}[\mathsf{gid}, \mathsf{epoch}]$. This models the fact that when the destination $(\mathsf{gid}, \mathsf{epoch})$ is initialized by the adversary, we let the adversary decide if the ideal server should return the stored proposals or not.
  - Otherwise, the ideal party outputs $\mathtt{accept} := \mathtt{false}$. This case occurs if $\mathtt{accept}' = \mathtt{false}$ (i.e., the adversary decides to reject) or $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] = \bot$ (i.e., the destination has not been initialized).

  In all three cases, the ideal server outputs the destination $(\mathsf{gid}, \mathsf{epoch})$ to $\mathcal{Z}$. Since parties access the server via a client-anonymous authenticated channel, the server never outputs the accessing party's identity.
- If the server is malicious, $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ gives $\mathsf{Ptr}[\mathsf{id}]$ to the adversary. The adversary specifies the protocol result $\mathtt{accept}'$ and the proposals $\vec{p}'$, and they ate outputted to id. This means the malicious server can return an arbitrary message to the parties. This is consistent with $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ which allows an adversary to inject malicious proposals.

If id successfully fetches the proposals (i.e., id receives $\mathtt{accept} = \mathtt{true}$), then id creates a corresponding commit $(c_0, \vec{c})$ and welcome messages $\vec{w} = \{\, \hat{w} \,\}$ as in $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ (cf. *Committing to proposals* in Section 5.3.3).

Publish commit and welcome messages id finally publishes the commit and welcome messages to the server. We first describe how it publishes a commit. Before publishing the commit message, the party permutes $\vec{c}$ to $\vec{c}_{\mathsf{perm}}$ using the function `*permute-commit`. This effectively makes the party's identity and index of $\vec{c}$ unlikable. Note that the `*permute-commit` function uses a random permutation (if **safe** is true for the id's current epoch) or a permutation the adversary chooses (if **safe** is false). This means if **safe** is true the permuted index will look random from the adversary.

- If the server is honest (i.e., `ServerStat` = 'good'), $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ gives $(\mathsf{gid}, \mathsf{epoch}, c_0, \vec{c}_{\mathsf{perm}})$ to the adversary. The party accesses the server via a client-anonymous authenticated channel, and therefore, the adversary does not receive the party's identity. The adversary reports whether the protocol succeeds or not by setting `accept`′ to true or false. $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ then determines the party and server's output as follows.
  - If $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] = (\top, \mathsf{node\text{-}id})$ and $\mathsf{node\text{-}id} = \mathsf{Ptr}[\mathsf{id}]$, the ideal party always outputs `accept` := `true`. This models correctness: if a party holds the same group state used to initialize node-id, the server must accept the published commit message.
  - If $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] = (\top, \text{'adv'})$ and `accept`′ = `true`, the ideal party outputs `accept` := `true`. This models the fact that if the destination $(\mathsf{gid}, \mathsf{epoch})$ is initialized by the adversary, we let the adversary decide if the ideal server accepts the published commit.
  - Otherwise, the ideal party outputs `accept` := `false`. This case occurs if `accept`′ = `false` (i.e., the adversary decides to reject), $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] \neq (\top, *)$ (i.e., the destination has not been initialized or a commit message has been issued), or $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] = (\top, \mathsf{node\text{-}id})$ but $\mathsf{node\text{-}id} \neq \mathsf{Ptr}[\mathsf{id}]$ (i.e., the party is located in a different commit node).

  In all three cases, the ideal server outputs the destination $(\mathsf{gid}, \mathsf{epoch})$ and the published commit $(c_0, \vec{c}_{\mathsf{perm}})$ to $\mathcal{Z}$. Since parties access the server via a client-anonymous authenticated channel, the server never outputs the accessing party's identity.
- If the server is malicious, $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ gives $(\mathsf{Ptr}[\mathsf{id}], c_0, \vec{c}_{\mathsf{perm}})$ to the adversary. The adversary specifies the protocol result `accept`′, and it is outputted to id. This means the malicious server can decide the protocol result arbitrarily.

Finally, id publishes the welcome messages $\widehat{w} \in \vec{w}$ via the subroutine `PublishWelcome`. To hide the relationship among welcome messages, id publish $\widehat{w}$ separately. $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ informs the adversary of publishing a welcome message by giving $(\mathsf{id}_t, \widehat{w})$, where id_t is the intended recipient of $\widehat{w}$. Note that the party accesses the server via a client-anonymous authenticated channel, and therefore the adversary does not receive the party's identity. The adversary reports whether the protocol succeeds or not by setting `accept`′ to true or false. $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ then determines the party and server's output as follows.

- If the server is honest (i.e., `ServerStat` = 'good'), $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ stores $\mathsf{WelDB}[\mathsf{id}_t] \leftarrow \widehat{w}$ and outputs `accept` := `true` to id. The ideal server outputs $(\mathsf{id}_t, \widehat{w})$. This models correctness: the honest server always accepts the welcome message and does not know who sent it.
- If the server is malicious, $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ outputs `accept`′ specified by the adversary. This means the malicious server can decide the protocol result arbitrarily.

Fetch and process commits (See Figure 5.44) To process, party id needs to fetch the commit and proposals from the server via the subroutine `FetchCommit`. Before fetching a commit and proposals, the party permutes

its index to $\mathsf{index}_{\widehat{c}}$ by the function `*permuted-commit-index` to make the party's identity and index of $\vec{c}$ unlikable. Here, the function uses a random permutation (if **safe** is true for the id's current epoch) or a permutation the adversary chooses (if **safe** is false). This models the fact that if **safe** is true, then the permuted index seems random to the adversary.

- If the server is honest (i.e., $\texttt{ServerStat} = \text{'good'}$), $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ gives $(\mathsf{gid}, \mathsf{epoch}, \mathsf{index}_{\widehat{c}})$ to the adversary, where $\mathsf{index}_{\widehat{c}}$ is the permuted id's index. Note that the party accesses to the server via a client-anonymous authenticated channel, and therefore, the adversary does not receive the party's identity. The adversary reports whether the protocol succeeds or not by setting $\texttt{accept}'$ to true or false, and specifies the commit message $(c'_0, \widehat{c}', \vec{p}')$. $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ then determines the party and server's output as follows.
  - If $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] = ((c'_0, \vec{c}'), \mathsf{node\text{-}id})$ and $\mathsf{node\text{-}id} = \mathsf{Ptr}[\mathsf{id}]$, the ideal party always outputs $\texttt{accept} := \texttt{true}$ and the commit message $(c_0, \widehat{c}, \vec{p}) := (c'_0, \vec{c}'[\mathsf{index}_{\widehat{c}}], \mathsf{PropDB}[\mathsf{gid}, \mathsf{epoch}])$. This models correctness: if a party holds the same group state used to initialize node-id, the server must accept and returns the stored commit (which was received during protocol `PublishCommit`)
  - If $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] = ((c'_0, \vec{c}'), \text{'adv'})$ and $\texttt{accept}' = \texttt{true}$, the ideal party outputs $\texttt{accept} := \texttt{true}$ and the commit $(c_0, \widehat{c}, \vec{p}) := (c'_0, \vec{c}'[\mathsf{index}_{\widehat{c}}], \mathsf{PropDB}[\mathsf{gid}, \mathsf{epoch}])$. This models the fact that if the destination $(\mathsf{gid}, \mathsf{epoch})$ was initialized by the adversary, the adversary gets to decide if the ideal server accepts.
  - Otherwise, the ideal party outputs $\texttt{accept} := \texttt{false}$. This case occurs if $\texttt{accept}' = \texttt{false}$ (i.e., the adversary decides to reject), $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] \neq ((c'_0, \vec{c}'), *)$ (i.e., the destination has not been initialized or a commit message has not been issued), or $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] = (\top, \mathsf{node\text{-}id})$ but $\mathsf{node\text{-}id} \neq \mathsf{Ptr}[\mathsf{id}]$ (i.e., the party is located in a different commit node).

  In all three cases, the ideal server outputs $(\mathsf{gid}, \mathsf{epoch}, \mathsf{index}_{\widehat{c}})$ to $\mathcal{Z}$.
- If the server is malicious, $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ gives $(\mathsf{Ptr}[\mathsf{id}], \mathsf{index}_{\widehat{c}})$ to the adversary. The adversary specifies the protocol result $\texttt{accept}'$ and the commit $(c'_0, \widehat{c}', \vec{p}')$, and it is outputted to id. This means the malicious server can return an arbitrary message to the parties. This is consistent with $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ which allows an adversary to inject malicious commits and proposals.

Finally, if id successfully fetches a commit and a list of proposals (i.e., id receives $\texttt{accept} = \texttt{true}$), id then processes them as in $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ (cf. *Processing commits* in Section 5.3.3).

Join a group (See Figure 5.44). To join a group, party id fetches a welcome message from the server via the subroutine `FetchWelcome`. $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ first informs the adversary that id fetches a welcome message. In this case, the server and the adversary lean who is accessing. The adversary reports whether the protocol succeeds or not by setting $\texttt{accept}'$ to true or false, and specifies the welcome message $\widehat{w}'$. $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ then determines the party and server's output as follows.

- If the server is honest (i.e., $\texttt{ServerStat} = \text{'good'}$), $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ returns to id the welcome message $\widehat{w} := \mathsf{WelDB}[\mathsf{id}]$ stored in its database or $\texttt{accept} = \texttt{false}$ if $\mathsf{WelDB}[\mathsf{id}] = \bot$. This models correctness: the honest server must return the message received during the subroutine `PublishWelcome`. The ideal server outputs the accessing party's identity id, which models the fact that any concrete protocol allows the server to know who was fetching the welcome message.
- If the server is malicious, the specified welcome message $\widehat{w}'$ is outputted to id. This models that the malicious server can send an arbitrary message to parties.

Finally, if id successfully fetches a welcome message (i.e., id receives `accept = true`), $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ then processes the welcome message as in $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ (cf. *Joining a group* in Section 5.3.3).

Group keys (See Figure 5.44). Parties can fetch the current group secret via the `Key` query. The returned group secret k is random if the protocol guarantees its confidentiality (identified by the **safe** predicate). Otherwise, k is set by the adversary. Unlike $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$, $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ only provides an interface to retrieve one group secret k. That is, it does not have the interface to retrieve the group metadata secret k_{mh} and the next key function. This is because, those were special keys only used to secure the dynamic metadata (see Section 5.4.1), and in particular, $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ only requires one group secret k that will be used to exchange the actual message.

Corruption. Similar to $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$, the adversary can obtain party id's internal states via the $(\texttt{exposed}, \mathsf{id})$ query. To prevent the so-called commitment problem, $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ fixes the value of the safety predicate **safe** when a new group or epoch is initialized. This is controlled by the function `*mark-next-db-initialized-epoch` in `Create` and `Commit`. This is because the group secret generated at a specific commit node is explicitly used in the real protocol and we must restrict the adversary to not corrupting these nodes in order not to trivially win the security game. In addition, the adversary can corrupt the server via the `CorruptServer` query, and take over the role of the server. For simplicity, we assume once the server becomes malicious, it remains malicious and will never become honest. Finally, as mentioned in the overview, we do not consider adversary-controlled randomness in the current model.

Functions Used by the Adversary. $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ offers the publish and fetch message interfaces to the adversary so that it can impersonate an honest party to a server. Similar to the case an honest party accesses the sever, $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ defines the ideal function when the adversary (malicious party) accesses the server. Since the party is malicious, $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ never requires correctness. In contrast, $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ defines the security requirements: It defines the conditions under which the adversary can access the server. In case the conditions do not hold, the functionality models the fact that the adversary cannot access the server. Note that when both party and server are malicious, $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ does nothing since the adversary can perform all the protocols by itself.

Publish or fetch proposal or commit messages by the adversary. To publish a proposal or commit, the adversary sends `PublishProposalAdv` or `PublishCommitAdv` to $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$. $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ determines the server's output as follows.

- If $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] = (\top, \mathsf{node\text{-}id})$ and $\texttt{accept}' = \texttt{true}$, $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ checks whether authenticity is guaranteed for the honestly generated epoch node-id by the safety predicate **adv-access-allowed**$(\mathsf{node\text{-}id})$. If the predicate returns `false`, then $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ halts. This models the fact that the adversary can publish messages for an honestly initialized epoch only when adversarial access is allowed. If access is allowed, the server stores the published message from the adversary in its database.
- If $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] = (\top, \text{'adv'})$ and $\texttt{accept}' = \texttt{true}$, the ideal server outputs $\texttt{accept} := \texttt{true}$. This models the fact that if the destination $(\mathsf{gid}, \mathsf{epoch})$ was initialized by the adversary, the adversary gets to decide whether the ideal server accepts.
- Otherwise, the ideal server outputs $\texttt{accept} := \texttt{false}$. This case occurs if $\texttt{accept}' = \texttt{false}$ (i.e., the adversary decides to reject) or $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] \neq (\top, *)$ (i.e., the destination has not been initialized or a commit message has been issued).

To fetch proposals, the adversary sends `FetchProposalsAdv` to $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$. $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ determines the server's output as follows.

- If $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] = (*, \mathsf{node\text{-}id})$ and $\texttt{accept}' = \texttt{true}$, $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ checks whether authenticity is guaranteed for the honestly generated epoch node-id by the safety predicate **adv-access-allowed**$(\mathsf{node\text{-}id})$.If

the predicate returns `false`, then $\mathcal{F}_{\text{CGKA}}^{\text{mh}}$ halts. This models the fact that the adversary can fetch proposals from an honestly initialized epoch only when adversarial access is allowed. If the access is allowed, the adversary obtains $\vec{p} := \mathsf{PropDB}[\mathsf{gid}, \mathsf{epoch}]$.

- If $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] = (*, \text{'adv'})$ and $\texttt{accept}' = \texttt{true}$, the ideal server outputs $\texttt{accept} := \texttt{true}$. This models the fact that if the destination $(\mathsf{gid}, \mathsf{epoch})$ was initialized by the adversary, the adversary can decide to accept.
- Otherwise, the ideal party outputs $\texttt{accept} := \texttt{false}$. This case occurs if $\texttt{accept}' = \texttt{false}$ (i.e., the adversary decides to reject) or $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] = \bot$ (i.e., the destination has not been initialized).

Finally, to publish a proposal or a commit the adversary sends $(\texttt{FetchCommitAdv}, \mathsf{gid}, \mathsf{epoch}, \mathsf{index})$ to $\mathcal{F}_{\text{CGKA}}^{\text{mh}}$. $\mathcal{F}_{\text{CGKA}}^{\text{mh}}$ determines the server's output as follows.

- If $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] = ((c_0', \vec{c}'), \mathsf{node\text{-}id})$ and $\texttt{accept}' = \texttt{true}$, $\mathcal{F}_{\text{CGKA}}^{\text{mh}}$ checks whether authenticity is guaranteed for the honestly generated epoch node-id by the safety predicate **adv-access-allowed**$(\mathsf{node\text{-}id})$. If If the predicate returns `false`, then $\mathcal{F}_{\text{CGKA}}^{\text{mh}}$ halts. This models the fact that the adversary can fetch messages for an honestly initialized epoch only when adversarial access is allowed. If access is allowed, the adversary obtains the stored $(c_0, \widehat{c}, \vec{p}) := (c_0', \vec{c}'[\mathsf{index}_{\widehat{c}}], \mathsf{PropDB}[\mathsf{gid}, \mathsf{epoch}])$.
- If $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] = ((c_0', \vec{c}'), \text{'adv'})$ and $\texttt{accept}' = \texttt{true}$, the adversary obtains the stored $(c_0, \widehat{c}, \vec{p}) := (c_0', \vec{c}'[\mathsf{index}_{\widehat{c}}], \mathsf{PropDB}[\mathsf{gid}, \mathsf{epoch}])$. This models the fact that if the destination $(\mathsf{gid}, \mathsf{epoch})$ was initialized by the adversary, the adversary can decide to accept.
- Otherwise, the ideal party outputs $\texttt{accept} := \texttt{false}$. This case occurs if $\texttt{accept}' = \texttt{false}$ (i.e., the adversary decides to reject) or $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] \neq ((c_0', \vec{c}'), *)$ (i.e., the destination has not been initialized or a commit message has not been issued).

Publish or fetch welcome messages by the adversary. To publish or fetch messages, the adversary sends `PublishWelcome` or `FetchWelcome` to $\mathcal{F}_{\text{CGKA}}^{\text{mh}}$. In this case, $\mathcal{F}_{\text{CGKA}}^{\text{mh}}$ determines the server's output as in the case an honest party accesses the server. The protocol succession means the adversary succeeds to publish or fetch a welcome message.

Initialization

1 : // Line 1-7 below are identical to selective downloading $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$
2 : $\mathsf{Ptr}[*], \mathsf{Prop}[*], \mathsf{Node}[*], \mathsf{Wel}[*] \leftarrow \perp$
3 : $\mathsf{propCtr}, \mathsf{nodeCtr} \leftarrow 1$
4 : $\mathtt{flag}_{\mathsf{selDL}} = \mathtt{true}, \mathsf{DesignatedCom}[*] \leftarrow \perp$
5 : $\mathtt{flag}_{\mathsf{contHide}} = \mathtt{true}$
6 : $\mathsf{PropID}[*], \mathsf{NodeID}[*] \leftarrow \perp$
7 : $\mathsf{Rand}[*] \leftarrow$ 'good'
8 : // Initialize below for dynamic metadata-hiding
9 : $\mathtt{flag}_{\mathsf{dbinit}} = \mathtt{true}$
10 : $\mathsf{PropDB}[*,*], \mathsf{ComDB}[*,*], \mathsf{WelDB} \leftarrow \perp$
11 : $\mathtt{ServerStat} \leftarrow$ 'good'

Input from the party $\mathsf{id}_{\mathsf{creator}}$
Input $(\mathtt{Create}, \mathsf{svk})$

1 : **req** $\mathsf{Ptr}[\mathsf{id}_{\mathsf{creator}}] = \perp$
2 : // Run identical code as Input $(\mathtt{Create}, \mathsf{svk})$ of $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$,
3 : // excluding the final **return** $(\mathsf{gid}, 0, \mathsf{mem})$ line
4 : $\langle\!\langle$Insert $(\mathtt{Create}, \mathsf{svk})$ of $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}\rangle\!\rangle$
5 : **req** $\mathtt{RegisterGroup}(\mathsf{gid}, 0)$
6 : $\mathtt{*mark\text{-}next\text{-}db\text{-}initialized\text{-}epoch}(\mathsf{Ptr}[\mathsf{id}_{\mathsf{creator}}])$

Subroutine $\mathtt{RegisterGroup}(\mathsf{gid}, \mathsf{epoch})$

1 : $\mathsf{channelType} \leftarrow (\text{'anon'}, \perp, \mathsf{Sv})$
2 : **req** $\mathsf{Ptr}[\mathsf{id}] = 0$
3 : Send $(\mathsf{channelType}, \mathtt{RegisterGroup}, \mathsf{gid}, 0)$ to $\mathcal{S}$ and
receive $\mathtt{accept}'$
4 : // If Sv is good, then accept if $\mathsf{ComDB}[\mathsf{gid}, *]$ is empty
5 : **if** $\mathtt{ServerStat} =$ 'good' **then**
6 : **if** $\mathsf{ComDB}[\mathsf{gid}, *] = \perp$ **then**
7 : $\mathtt{accept} \leftarrow \mathtt{true}$
8 : // Main group is assigned to node-id $= 0$.
9 : $\mathsf{ComDB}[\mathsf{gid}, 0] \leftarrow (\top, 0)$
10 : **else**
11 : $\mathtt{accept} \leftarrow \mathtt{false}$
12 : Send $(\mathsf{channelType}, \mathtt{RegisterGroup}, (\mathtt{accept}, \mathsf{gid}, 0))$ to Sv
13 : // If Sv is corrupt, then let $\mathcal{S}$ decide if Sv accepts
14 : **else**
15 : $\mathtt{accept} \leftarrow \mathtt{accept}'$
16 : **return** accept

FIGURE 5.43: The ideal MH-CGKA functionality $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$: `Create` function for honest parties. For better readability, we outsource the lines identical to $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ to Figure 5.5. Note that the `Create` function is invoked only once by the previously designated party $\mathsf{id}_{\mathsf{creator}}$. (Only one $\mathsf{id}_{\mathsf{creator}}$ exists.)

Inputs from a party id

Input $(\texttt{Propose}, \mathsf{act}^{\dagger})$

1 : **req** $\mathsf{Ptr}[\mathsf{id}] \neq \bot$
2 : $\mathsf{gid} \leftarrow \mathsf{Node}[\mathsf{Ptr}[\mathsf{id}]].\mathsf{gid}$
3 : $\mathsf{epoch} \leftarrow \mathsf{Node}[\mathsf{Ptr}[\mathsf{id}]].\mathsf{epoch}$
4 : // Run identical code as Input $(\texttt{Propose}, \mathsf{act})$ of $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$,
5 : // excluding the final **return** p line
6 : $\langle\!\langle$Insert $(\texttt{Propose}, \mathsf{act})$ of $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}\rangle\!\rangle$
7 : **req** $\texttt{PublishProposal}(\mathsf{gid}, \mathsf{epoch}, p)$
8 : **return** p

Input $(\texttt{Commit}, \mathsf{svk})$

1 : **req** $\mathsf{Ptr}[\mathsf{id}] \neq \bot$
2 : $\mathsf{gid} \leftarrow \mathsf{Node}[\mathsf{Ptr}[\mathsf{id}]].\mathsf{gid}$
3 : $\mathsf{epoch} \leftarrow \mathsf{Node}[\mathsf{Ptr}[\mathsf{id}]].\mathsf{epoch}$
4 : $(\texttt{accept}, \vec{p}) \leftarrow \texttt{FetchProposals}(\mathsf{gid}, \mathsf{epoch})$
5 : **req** accept
6 : // Run identical code as Input $(\texttt{Commit}, \vec{p}, \mathsf{svk})$ of $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$,
7 : // excluding the final **return** $(c_0, \vec{c}, \vec{w} = \{\widehat{w}\})$ line
8 : $\langle\!\langle$Insert $(\texttt{Commit}, \vec{p}, \mathsf{svk})$ of $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}\rangle\!\rangle$
9 : // Permute the order of party sensitive commitment $\vec{c}$
10 : $\vec{c}_{\mathsf{perm}} \leftarrow \texttt{*permute-commit}(\mathsf{Ptr}[\mathsf{id}], \vec{c})$
11 : **try** $\texttt{PublishCommit}(\mathsf{gid}, \mathsf{epoch}, c_0, \vec{c}_{\mathsf{perm}})$
12 : **foreach** $\widehat{w} \in \vec{w}$ **do**
13 : **parse** $(\mathsf{id}_t, *) \leftarrow \widehat{w}$
14 : **try** $\texttt{PublishWelcome}(\mathsf{id}_t, \widehat{w})$
15 : $\texttt{*mark-next-db-initialized-epoch}(\mathsf{NodeID}[c_0])$
16 : **return** $(c_0, \vec{c}_{\mathsf{perm}}, \vec{w})$

Input Process

1 : **req** $\mathsf{Ptr}[\mathsf{id}] \neq \bot$
2 : $\mathsf{gid} \leftarrow \mathsf{Node}[\mathsf{Ptr}[\mathsf{id}]].\mathsf{gid}$
3 : $\mathsf{epoch} \leftarrow \mathsf{Node}[\mathsf{Ptr}[\mathsf{id}]].\mathsf{epoch}$
4 : // Permute index of party id for selective downloading.
5 : $\mathsf{index}_{\widehat{c}} \leftarrow \texttt{*permuted-commit-index}(\mathsf{Ptr}[\mathsf{id}], \mathsf{id})$
6 : $(\texttt{accept}, c_0, \widehat{c}, \vec{p}) \leftarrow \texttt{FetchCommit}(\mathsf{gid}, \mathsf{epoch}, \mathsf{index}_{\widehat{c}})$
7 : **req** accept
8 : // Run identical code as Input $(\texttt{Process}, c_0, \widehat{c}, \vec{p})$ of $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$,
9 : // including the final **return** $\texttt{*output-proc}(\text{node-id}')$ line
10 : $\langle\!\langle$Insert $(\texttt{Process}, c_0, \widehat{c}, \vec{p})$ of $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}\rangle\!\rangle$

Input Join

1 : **req** $\mathsf{Ptr}[\mathsf{id}] = \bot$
2 : $(\texttt{accept}, \widehat{w}) \leftarrow \texttt{FetchWelcome}(\mathsf{id})$
3 : **req** accept
4 : // Run identical code as Input $(\texttt{Join}, \widehat{w})$ of $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$,
5 : // including the final **return** $\texttt{*output-join}(\text{node-id})$ line
6 : $\langle\!\langle$Insert $(\texttt{Join}, \widehat{w})$ of $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}\rangle\!\rangle$

Input Key

1 : // Run identical code as Input $(\texttt{Key})$ of $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$,
2 : // including the final **return** $\mathsf{Node}[\mathsf{Ptr}[\mathsf{id}]].\mathsf{key}$ line
3 : $\langle\!\langle$Inset $(\texttt{Key})$ of $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}\rangle\!\rangle$

FIGURE 5.44: The ideal MH-CGKA functionality $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$: `Propose`, `Commit`, `Process`, and `Join` functions for honest parties. For better readability, we outsource the lines identical to $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ to Figures 5.6, 5.7 and 5.9. †: $\mathsf{act} \in \{$ 'upd'-svk, 'add'-id_t, 'rem'-id_t $\}$

Subroutine $\texttt{PublishProposal}(\mathsf{gid}, \mathsf{epoch}, p)$

1 : $\mathsf{channelType} \leftarrow (\text{'anon'}, \bot, \mathsf{Sv})$ // Connect to the server via anonymous channel.
2 : **if** $\texttt{ServerStat} = \text{'good'}$ **then**
3 : Send $(\mathsf{channelType}, \texttt{PublishProposal}, \mathsf{gid}, \mathsf{epoch}, p)$ to $\mathcal{S}$ and receive $\texttt{accept}'$
4 : // If the party assigned to gid-epoch accesses, the server must accept.
5 : **if** $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] = (\top, \mathsf{node\text{-}id}) \wedge \mathsf{node\text{-}id} = \mathsf{Ptr}[\mathsf{id}]$ **then**
6 : $\mathsf{PropDB}[\mathsf{gid}, \mathsf{epoch}] \mathbin{+}\!\!\leftarrow p$
7 : $\texttt{accept} \leftarrow \texttt{true}$
8 : // If gid-epoch was initialized by the adversary, $\mathcal{S}$ decide to accept or reject.
9 : **elseif** $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] = (\top, \text{'adv'}) \wedge \texttt{accept}'$ **then**
10 : $\mathsf{PropDB}[\mathsf{gid}, \mathsf{epoch}] \mathbin{+}\!\!\leftarrow p$
11 : $\texttt{accept} \leftarrow \texttt{true}$
12 : // Otherwise, the server must reject.
13 : **else** $\texttt{accept} \leftarrow \texttt{false}$
14 : Send $(\mathsf{channelType}, \texttt{PublishProposal}, (\texttt{accept}, \mathsf{gid}, \mathsf{epoch}, p))$ to Sv
15 : // If Sv is corrupt, then $\mathcal{S}$ decides if Sv accepts.
16 : **else**
17 : Send $(\mathsf{channelType}, \texttt{PublishProposal}, \mathsf{Ptr}[\mathsf{id}], p)$ to $\mathcal{S}$ and receive $\texttt{accept}'$
18 : $\texttt{accept} \leftarrow \texttt{accept}'$
19 : **return** accept

Subroutine $\texttt{FetchProposals}(\mathsf{gid}, \mathsf{epoch})$

1 : $\mathsf{channelType} \leftarrow (\text{'anon'}, \bot, \mathsf{Sv})$ // Connect to the server via anonymous channel.
2 : **if** $\texttt{ServerStat} = \text{'good'}$ **then**
3 : Send $(\mathsf{channelType}, \texttt{FetchProposals}, \mathsf{gid}, \mathsf{epoch})$ to $\mathcal{S}$ and receive $\texttt{accept}'$
4 : // If the party assigned to gid-epoch accesses, the server must accept.
5 : **if** $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] = (*, \mathsf{node\text{-}id}) \wedge \mathsf{node\text{-}id} = \mathsf{Ptr}[\mathsf{id}]$ **then**
6 : $(\texttt{accept}, \vec{p}) \leftarrow (\texttt{true}, \mathsf{PropDB}[\mathsf{gid}, \mathsf{epoch}])$
7 : // If gid-epoch was initialized by the adversary, $\mathcal{S}$ decide to accept or reject.
8 : **elseif** $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] = (*, \text{'adv'}) \wedge \texttt{accept}'$ **then**
9 : $(\texttt{accept}, \vec{p}) \leftarrow (\texttt{true}, \mathsf{PropDB}[\mathsf{gid}, \mathsf{epoch}])$
10 : // Otherwise, the server must reject.
11 : **else** $(\texttt{accept}, \vec{p}) \leftarrow (\texttt{false}, \bot)$
12 : Send $(\mathsf{channelType}, \texttt{FetchProposals}, (\texttt{accept}, \mathsf{gid}, \mathsf{epoch}))$ to Sv
13 : // If Sv is corrupt, then $\mathcal{S}$ decides what Sv returns to id.
14 : **else**
15 : Send $(\mathsf{channelType}, \texttt{FetchProposals}, \mathsf{Ptr}[\mathsf{id}])$ to $\mathcal{S}$ and receive $(\texttt{accept}', \vec{p}')$
16 : $(\texttt{accept}, \vec{p}) \leftarrow (\texttt{accept}', \vec{p}')$
17 : **return** $(\texttt{accept}, \vec{p})$

FIGURE 5.45: The ideal metadata-hiding CGKA functionality $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$: subroutines for publish and fetch proposal messages. They are used in `Propose` and `Commit` interface shown in Figure 5.44.

Subroutine $\texttt{PublishCommit}(\mathsf{gid}, \mathsf{epoch}, c_0, \vec{c}_{\mathsf{perm}})$

1 : $\mathsf{channelType} \leftarrow (\text{'anon'}, \bot, \mathsf{Sv})$ // Connect to the server via anonymous channel.
2 : **if** $\texttt{ServerStat} = \text{'good'}$ **then**
3 : Send $(\mathsf{channelType}, \texttt{PublishCommit}, \mathsf{gid}, \mathsf{epoch}, c_0, \vec{c}_{\mathsf{perm}})$ to $\mathcal{S}$ and receive $\texttt{accept}'$
4 : // If the party uses the state assigned to gid-epoch, the server must accept.
5 : **if** $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] = (\top, \mathsf{node\text{-}id}) \wedge \mathsf{node\text{-}id} = \mathsf{Ptr}[\mathsf{id}]$ **then**
6 : $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] \leftarrow ((c_0, \vec{c}_{\mathsf{perm}}), \mathsf{node\text{-}id})$
7 : // If an honest party initialize the next epoch, the corresponding node is marked.
8 : $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}+1] \leftarrow (\top, \mathsf{NodeID}[c_0])$
9 : $\texttt{accept} \leftarrow \texttt{true}$
10 : // If gid-epoch was initialized by the adversary, $\mathcal{S}$ decide to accept or reject.
11 : **elseif** $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] = (\top, \text{'adv'}) \wedge \texttt{accept}'$ **then**
12 : $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] \leftarrow ((c_0, \vec{c}_{\mathsf{perm}}), \text{'adv'})$
13 : // If an honest party initialize the next epoch, the corresponding node is marked.
14 : $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}+1] \leftarrow (\top, \mathsf{NodeID}[c_0])$
15 : $\texttt{accept} \leftarrow \texttt{true}$
16 : // Otherwise, the server must reject.
17 : **else** $\texttt{accept} \leftarrow \texttt{false}$
18 : Send $(\mathsf{channelType}, \texttt{PublishCommit}, (\texttt{accept}, \mathsf{gid}, \mathsf{epoch}, c_0, \vec{c}_{\mathsf{perm}}))$ to Sv
19 : **else** // If Sv is corrupt, then $\mathcal{S}$ decides if Sv accepts.
20 : Send $(\mathsf{channelType}, \texttt{PublishCommit}, \mathsf{gid}, \mathsf{epoch}, c_0, \vec{c}_{\mathsf{perm}})$ to $\mathcal{S}$ and receive $\texttt{accept}'$
21 : $\texttt{accept} \leftarrow \texttt{accept}'$
22 : **return** $\texttt{accept}$

Subroutine $\texttt{FetchCommit}(\mathsf{gid}, \mathsf{epoch}, \mathsf{index}_{\widehat{c}})$

1 : $\mathsf{channelType} \leftarrow (\text{'anon'}, \bot, \mathsf{Sv})$ // Connect to the server via anonymous channel.
2 : **if** $\texttt{ServerStat} = \text{'good'}$ **then**
3 : Send $(\mathsf{channelType}, \texttt{FetchCommit}, \mathsf{gid}, \mathsf{epoch}, \mathsf{index}_{\widehat{c}})$ to $\mathcal{S}$ and receive $\texttt{accept}'$
4 : // If the party assigned to gid-epoch accesses, the server must accept.
5 : **if** $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] = ((c_0', \vec{c}'), \mathsf{node\text{-}id}) \wedge \mathsf{node\text{-}id} = \mathsf{Ptr}[\mathsf{id}]$ **then**
6 : $(\texttt{accept}, c_0, \widehat{c}, \vec{p}) \leftarrow (\texttt{true}, c_0', \vec{c}'[\mathsf{index}_{\widehat{c}}], \mathsf{PropDB}[\mathsf{gid}, \mathsf{epoch}])$
7 : // If gid-epoch was initialized by the adversary, $\mathcal{S}$ decide to accept or reject.
8 : **elseif** $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] = ((c_0', \vec{c}'), \text{'adv'}) \wedge \texttt{accept}'$ **then**
9 : $(\texttt{accept}, c_0, \widehat{c}, \vec{p}) \leftarrow (\texttt{true}, c_0', \vec{c}'[\mathsf{index}_{\widehat{c}}], \mathsf{PropDB}[\mathsf{gid}, \mathsf{epoch}])$
10 : // Otherwise, the server must reject.
11 : **else** $(\texttt{accept}, c_0, \widehat{c}, \vec{p}) \leftarrow (\texttt{false}, \bot, \bot, \bot)$
12 : Send $(\mathsf{channelType}, \texttt{FetchCommit}, (\texttt{accept}, \mathsf{gid}, \mathsf{epoch}, \mathsf{index}_{\widehat{c}}))$ to Sv
13 : **else**
14 : Send $(\mathsf{channelType}, \texttt{FetchCommit}, \mathsf{Ptr}[\mathsf{id}], \mathsf{index}_{\widehat{c}})$ to $\mathcal{S}$ and receive $(\texttt{accept}', c_0', \widehat{c}', \vec{p}')$
15 : $(\texttt{accept}, c_0, \widehat{c}, \vec{p}) \leftarrow (\texttt{accept}', c_0', \widehat{c}', \vec{p}')$
16 : **return** $(\texttt{accept}, c_0, \widehat{c}, \vec{p})$

FIGURE 5.46: The ideal metadata-hiding CGKA functionality $\mathcal{F}_{\text{CGKA}}^{\mathsf{mh}}$: subroutines for publish and fetch commit messages. They are used in `Commit` and `Process` interface shown in Figure 5.44. If $\vec{c} = \bot$, we define $\vec{c}[\mathsf{index}] = \bot$ for any index index.

Input from a party id **or adversary** $\mathcal{S}$
Input `PublishWelcome`$(\mathsf{id}_t, \widehat{w})$

1 : // Connect to the server via anonymous channel.
2 : channelType $\leftarrow$ ('anon', $\perp$, Sv)
3 : Send (channelType, `PublishWelcome`, $\mathsf{id}_t, \widehat{w}$) to $\mathcal{S}$ and receive `accept`$'$
4 : // If Sv is honest, then store $\widehat{w}$ in WelDB
5 : **if** `ServerStat` $=$ 'good' **then**
6 : `accept` $\leftarrow$ `true`
7 : WelDB$[\mathsf{id}_t] \leftarrow \widehat{w}$
8 : Send (channelType, `PublishWelcome`, (`accept`, $\mathsf{id}_t, \widehat{w}$)) to Sv
9 : // If Sv is corrupt, then $\mathcal{S}$ decides if Sv accepts.
10 : **else**
11 : `accept` $\leftarrow$ `accept`$'$
12 : // If invoked by $\mathcal{S}$, then no output is required
13 : **if** `ServerStat` $=$ 'good' $\vee$ invoked by party id **then**
14 : **return** `accept`

Input `FetchWelcome`(id)

1 : // Connect to the server via authenticated channel.
2 : channelType $\leftarrow$ ('auth', id, Sv)
3 : Send (channelType, `FetchWelcome`) to $\mathcal{S}$ and receive (`accept`$', \widehat{w}'$)
4 : **if** `ServerStat` $=$ 'good' **then**
5 : **if** WelDB[id] $\neq \perp$ **then**
6 : (`accept`, $\widehat{w}$) $\leftarrow$ (`true`, WelDB[id])
7 : **else**
8 : (`accept`, $\widehat{w}$) $\leftarrow$ (`false`, $\perp$)
9 : Send (channelType, `FetchWelcome`, (`accept`, id)) to Sv
10 : // If Sv is corrupt, then let $\mathcal{S}$ decide Sv's output
11 : **else**
12 : (`accept`, $\widehat{w}$) $\leftarrow$ (`accept`$', \widehat{w}'$)
13 : // If invoked by $\mathcal{S}$, then no output is required
14 : **if** `ServerStat` $=$ 'good' $\vee$ invoked by party id
15 : **return** (`accept`, $\widehat{w}$)

FIGURE 5.47: The ideal metadata-hiding CGKA functionality $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$: Subroutines for publish and fetch welcome messages. They are used in `Commit` and `Join` interface shown in Figure 5.44.

*permuted-commit-index(node-id, id)

1 : **assert** Node[node-id] $\neq \perp$
2 : // If $\texttt{flag}_{\mathsf{selDL}}$ is false, there is no need to permute index.
3 : **if** $\neg\texttt{flag}_{\mathsf{selDL}}$ **then**
4 : **return** $\perp$
5 : mem $\leftarrow$ Node[node-id].mem
6 : **assert** id $\in$ mem
7 : index $\leftarrow$ Node[node-id].index(id)
8 : **if** Node[node-id].perm $= \perp$ **then**
9 : *set-index-permutation(node-id)
10 : $\phi \leftarrow$ Node[node-id].perm
11 : **return** ϕ(index)

*set-index-permutation(node-id)

1 : mem $\leftarrow$ Node[node-id].mem
2 : // If the input node is not corrupted,
// sample a random permutation over $S_{|\mathsf{mem}|}$.
3 : **if random-index**(node-id) **then**
4 : $\phi \leftarrow\!\!\$\ S_{|\mathsf{mem}|}$
5 : Node[node-id].perm $\leftarrow \phi$
6 : **else**
7 : Send (Permutation, node-id) to $\mathcal{S}$ and receive ϕ
8 : **assert** $\phi \in S_{|\mathsf{mem}|}$
9 : Node[node-id].perm $\leftarrow \phi$

*permute-commit(node-id, $\vec{c}$)

1 : **assert** Node[node-id] $\neq \perp$
2 : // If $\texttt{flag}_{\mathsf{selDL}}$ is false, no party-depend commitment can exist.
3 : **if** $\neg\texttt{flag}_{\mathsf{selDL}}$ **then**
4 : **assert** $\vec{c} = \perp$
5 : **return** $\perp$
6 : **if** Node[node-id].perm $= \perp$ **then**
7 : *set-index-permutation(node-id)
8 : $\phi \leftarrow$ Node[node-id].perm
9 : $\vec{c}_{\mathsf{perm}} \leftarrow ()$
10 : **for** index $= 1, \ldots, |\vec{c}|$ **do**
11 : $\vec{c}_{\mathsf{perm}} \mathbin{+\!\!+\!\!\leftarrow} \vec{c}[\phi(\mathsf{index})]$
12 : **return** $\vec{c}_{\mathsf{perm}}$

FIGURE 5.48: Helper functions for $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$: Permute indices. $S_{|\mathsf{mem}|}$ is the set of permutations on the range $[|\mathsf{mem}|]$.

Input from the adversary $\mathcal{S}$
Input $(\texttt{PublishProposalAdv}, \mathsf{gid}, \mathsf{epoch}, p)$

```
1 : channelType ← ('anon', ⊥, Sv)
2 : Send (channelType, PublishProposalAdv, gid, epoch, p) to S and receive accept'
3 : // The honest server accepts if S decides to accept and ComDB[gid, epoch] = (⊤, *).
4 : if ServerStat = 'good' then
5 :   if ComDB[gid, epoch] = (⊤, node-id) ∧ accept' then
6 :     // If gid-epoch was initialized by an honest party, F^mh_CGKA checks authenticity is guaranteed.
7 :     assert adv-access-allowed(node-id)
8 :     PropDB[gid, epoch] +← p
9 :     accept ← true
10 :  elseif ComDB[gid, epoch] = (⊤, 'adv') ∧ accept' then
11 :    PropDB[gid, epoch] +← p
12 :    accept ← true
13 :  else
14 :    accept ← false
15 :  Send (channelType, PublishProposal, (accept, gid, epoch, p)) to Sv
16 :  return accept
```

Input $(\texttt{FetchProposalsAdv}, \mathsf{gid}, \mathsf{epoch})$

```
1 : channelType ← ('anon', ⊥, Sv)
2 : Send (channelType, FetchProposalsAdv, gid, epoch) to S and
      receive accept'
3 : // The honest server accepts if S decides to accept and ComDB[gid, epoch] ≠ ⊥.
4 : if ServerStat = 'good' then
5 :   elseif ComDB[gid, epoch] = (*, node-id) ∧ accept' then
6 :     // If gid-epoch was initialized by an honest party, F^mh_CGKA checks authenticity is guaranteed.
7 :     assert adv-access-allowed(node-id)
8 :     (accept, p⃗) ← (true, PropDB[gid, epoch])
9 :   elseif ComDB[gid, epoch] = (*, 'adv') ∧ accept' then
10 :    (accept, p⃗) ← (true, PropDB[gid, epoch])
11 :  else
12 :    (accept, p⃗) ← (false, ⊥)
13 :  Send (channelType, FetchProposals, (accept, gid, epoch)) to Sv
14 :  return (accept, p⃗)
```

FIGURE 5.49: The ideal metadata-hiding CGKA functionality $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$: functions for the adversary $\mathcal{S}$. If $\vec{c} = \bot$, $\vec{c}[\mathsf{index}]$ is defined to be $\bot$ for any index.

Input from the adversary $\mathcal{S}$

Input $(\texttt{PublishCommitAdv}, \mathsf{gid}, \mathsf{epoch}, c_0, \vec{c})$

1 : $\mathsf{channelType} \leftarrow (\text{'anon'}, \bot, \mathsf{Sv})$
2 : Send $(\mathsf{channelType}, \texttt{PublishCommitAdv}, \mathsf{gid}, \mathsf{epoch}, c_0, \vec{c})$ to $\mathcal{S}$ and receive $\texttt{accept}'$
3 : // The honest server accepts if $\mathcal{S}$ decides to accept and $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] = (\top, *)$.
4 : **if** $\texttt{ServerStat} =$ 'good' **then**
5 : $\quad \mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}]$
6 : $\quad$ **elseif** $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] = (\top, \mathsf{node\text{-}id}) \wedge \texttt{accept}'$ **then**
7 : $\quad\quad$ // If gid-epoch was initialized by an honest party, $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ checks authenticity is guaranteed.
8 : $\quad\quad$ **assert adv-access-allowed**$(\mathsf{node\text{-}id})$
9 : $\quad\quad$ $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] \leftarrow ((c_0, \vec{c}), \mathsf{node\text{-}id})$
10 : $\quad\quad$ // If injection succeeds, the next epoch is marked as adversarial initialized.
11 : $\quad\quad$ $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}+1] \leftarrow (\top, \text{'adv'})$
12 : $\quad$ **elseif** $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] = (\top, \text{'adv'}) \wedge \texttt{accept}'$ **then**
13 : $\quad\quad$ $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] \leftarrow ((c_0, \vec{c}), \text{'adv'})$
14 : $\quad\quad$ // If injection succeeds, the next epoch is marked as adversarial initialized.
15 : $\quad\quad$ $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}+1] \leftarrow (\top, \text{'adv'})$
16 : $\quad$ **else**
17 : $\quad\quad$ $\texttt{accept} \leftarrow \texttt{false}$
18 : $\quad$ Send $(\mathsf{channelType}, \texttt{PublishCommit}, (\texttt{accept}, \mathsf{gid}, \mathsf{epoch}, c_0, \vec{c}))$ to Sv
19 : $\quad$ **return** $\texttt{accept}$

Input $(\texttt{FetchCommitAdv}, \mathsf{gid}, \mathsf{epoch}, \mathsf{index})$

1 : $\mathsf{channelType} \leftarrow (\text{'anon'}, \bot, \mathsf{Sv})$
2 : Send $(\mathsf{channelType}, \texttt{FetchCommitAdv}, \mathsf{gid}, \mathsf{epoch}, \mathsf{index})$ to $\mathcal{S}$ and receive $\texttt{accept}'$
3 : // The honest server accepts if $\mathcal{S}$ decides to accept and $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] = ((c_0', \vec{c}'), *)$.
4 : **if** $\texttt{ServerStat} =$ 'good' **then**
5 : $\quad$ **if** $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] = ((c_0', \vec{c}'), \mathsf{node\text{-}id}) \wedge \texttt{accept}'$ **then**
6 : $\quad\quad$ // If gid-epoch was initialized by an honest party, $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ checks authenticity is guaranteed.
7 : $\quad\quad$ **assert adv-access-allowed**$(\mathsf{node\text{-}id})$
8 : $\quad\quad$ $(\texttt{accept}, c_0, \widehat{c}, \vec{p}) \leftarrow (\texttt{true}, c_0', \vec{c}'[\mathsf{index}], \mathsf{PropDB}[\mathsf{gid}, \mathsf{epoch}])$
9 : $\quad$ **if** $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] = ((c_0', \vec{c}'), \text{'adv'}) \wedge \texttt{accept}'$ **then**
10 : $\quad\quad$ $(\texttt{accept}, c_0, \widehat{c}, \vec{p}) \leftarrow (\texttt{true}, c_0', \vec{c}'[\mathsf{index}], \mathsf{PropDB}[\mathsf{gid}, \mathsf{epoch}])$
11 : $\quad$ **else**
12 : $\quad\quad$ $(\texttt{accept}, c_0, \widehat{c}, \vec{p}) \leftarrow (\texttt{false}, \bot, \bot, \bot)$
13 : $\quad$ Send $(\mathsf{channelType}, \texttt{PublishProposal}, (\texttt{accept}, \mathsf{gid}, \mathsf{epoch}, \mathsf{index})$ to Sv
14 : $\quad$ **return** $(\texttt{accept}, c_0, \widehat{c}, \vec{p})$

FIGURE 5.50: The ideal metadata-hiding CGKA functionality $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$: functions for the adversary $\mathcal{S}$. If $\vec{c} = \bot$, $\vec{c}[\mathsf{index}]$ is defined to be $\bot$ for any index.

Input $(\texttt{Expose}, \mathsf{id})$

1 : **if** $\mathsf{Ptr}[\mathsf{id}] \neq \perp$ **then**
2 : $\quad \mathsf{Node}[\mathsf{Ptr}[\mathsf{id}]].\mathsf{exp} \mathrel{+}\!\!\leftarrow \mathsf{id}$
3 : $\quad$ `*update-stat-after-exp`(id) // Pending secrets are marked as exposed.
4 : $\quad \mathsf{svk} \leftarrow \mathsf{Node}[\mathsf{Ptr}[\mathsf{id}]].\mathsf{mem}[\mathsf{id}]$
5 : $\quad$ Send $(\texttt{exposed}, \mathsf{id}, \mathsf{svk})$ to $\mathcal{F}_{\text{AS}}$
6 : $\quad$ Send $(\mathsf{Ptr}[\mathsf{id}], \mathsf{Node}[\mathsf{Ptr}[\mathsf{id}]])$ to $\mathcal{S}$ // All information stored in $\mathsf{Node}[\mathsf{Ptr}[\mathsf{id}]]$ is sent to $\mathcal{S}$.
7 : Send $(\texttt{exposed}, \mathsf{id})$ to $\mathcal{F}_{\text{KS}}$
8 : **restrict** $\forall$node-id :
9 : $\quad$ **if** $\mathsf{Node}[\text{node-id}].\mathsf{chall} = \texttt{true}$ **then safe**$(\text{node-id}) = \texttt{true}$
10 : $\quad$ **if** $\mathsf{Node}[\text{node-id}].\mathsf{conthide} = \texttt{true}$ **then safe**$(\text{node-id}) = \texttt{true}$
11 : $\quad$ **if** $\mathsf{Node}[\text{node-id}].\mathsf{dbinit} = \texttt{true}$ **then safe**$(\text{node-id}) = \texttt{true}$

Input `CorruptServer`

1 : // Once corrupted, the server remains corrupted.
2 : `ServerStat` $\leftarrow$ 'adv'

`*mark-next-db-initialized-epoch`(node-id)

1 : **if safe**(node-id) **then**
2 : $\quad \mathsf{Node}[\text{node-id}].\mathsf{dbinit} \leftarrow \texttt{true}$
3 : **else**
4 : $\quad \mathsf{Node}[\text{node-id}].\mathsf{dbinit} \leftarrow \texttt{false}$

FIGURE 5.51: The metadata-hiding CGKA functionality $\mathcal{F}^{\mathsf{mh}}_{\text{CGKA}}$: Corruptions from the adversary $\mathcal{S}$. The difference between those of $\mathcal{F}^{\mathsf{ctxt}}_{\text{CGKA}}$ is highlighted in gray. $\mathcal{S}$ can call `CorruptServer` only once.

5.6.3 Security of the Wrapper Protocol W^{mh}

The following theorem proves that the wrapper protocol W^{mh} UC-realizes the ideal functionality $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ in the $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$-hybrid model.

Theorem 5.6.1. *Assuming that* SIG' *is* EUF-CMA *secure,* PRF *is a secure pseudorandom function, and* PRP *is a secure pseudorandom permutation, the protocol* W^{mh} *UC-realizes the ideal functionality* $\mathcal{F}^{\mathsf{mh}}_{CGKA}$ *in the* $\mathcal{F}^{\mathsf{ctxt}}_{CGKA}$*-hybrid model, where the safety predicates and leakage functions for* $\mathcal{F}^{\mathsf{mh}}_{CGKA}$ *are defined in Figures 5.31, 5.32 and 5.41.*

Proof. We consider the following sequence of hybrids. While the environment $\mathcal{Z}$ interacts with W^{mh} in Hybrid 1, it interacts with the ideal functionality $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ in Hybrid 6. Below, we first define all the hybrids and then explain how the simulators are defined.

Hybrid 1. This is the real-world execution of the protocol, where we make a syntactic change. We consider a simulator $\mathcal{S}_1$ that interacts with a dummy functionality $\mathcal{F}_{\mathsf{dummy}}$. $\mathcal{F}_{\mathsf{dummy}}$ sits between the environment $\mathcal{Z}$ and $\mathcal{S}_1$, and simply routs all messages without any modification. $\mathcal{S}_1$ internally simulates the real-world parties and adversary $\mathcal{A}$ by routing all the messages sent from $\mathcal{F}_{\mathsf{dummy}}$; from $\mathcal{A}$'s point of view, $\mathcal{S}_1$ is the environment $\mathcal{Z}$.

Hybrid 2. In this hybrid, we replace the dummy functionality $\mathcal{F}_{\mathsf{dummy}}$ by the ideal functionality $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ except that we replace the functions used within `Create`,`Propose`,`Commit`,`Process`, and `Join` by those defined in Figures 5.52 to 5.56. We call this modified ideal functionality $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA},2}$. In words, $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA},2}$ includes all the descriptions of the ideal functionality $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ for static metadata-hiding CGKA and outsources any other checks performed by the wrapper protocol W^{mh} to the simulator $\mathcal{S}_2$. Namely, these correspond to the party-server interaction. In this hybrid, all consistency and security regarding the static metadata are guaranteed. The description of $\mathcal{S}_2$ is provided in Lemma 5.6.2.

Hybrid 3. In this hybrid, we add all the missing consistency checks regarding the wrapper protocol W^{mh} of the ideal functionality $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ into $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA},2}$. We cal this ideal functionality $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA},3}$. More precisely, $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA},3}$ is identical to $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$ except that **random-index** (resp. **adv-access-allowed**) always returns false (resp. true). It only takes care of the correctness guarantees and does not guarantee any security properties regarding the dynamic metadata. Simulator $\mathcal{S}_3$ is defined identically to $\mathcal{S}_2$.

Hybrid 4. In this hybrid, we change how the randomness used to derive the permutation key and group signature key are generated. The simulator $\mathcal{S}_4$ is identical to $\mathcal{S}_3$ except that, rather than generating $\mathsf{permKey} \leftarrow \mathsf{PRF}(\mathsf{k_{mh}}, \text{'perm'})$ and $\mathsf{authKey} \leftarrow \mathsf{PRF}(\mathsf{k_{mh}}, \text{'auth'})$ (which occurs during `Create` or `Commit`), if **safe** is true for that epoch, then it samples a random $\mathsf{permKey}$ and $\mathsf{authKey}$. In this hybrid, we use $\mathcal{F}_{\mathsf{CGKA},4} := \mathcal{F}_{\mathsf{CGKA},3}$.

Hybrid 5. In this hybrid, we add the missing security guarantee on the randomness of the party's index. Namely, we modify $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA},4}$ to use the original **random-index** predicate, denoted as $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA},5}$. $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA},5}$ permutes the indices with a random permutation if predicate **random-index** is true. Simulator $\mathcal{S}_5$ is identical to $\mathcal{S}_4$.

Hybrid 6. In this hybrid, we add the missing security guarantee when an adversary tries to access the honest server. Namely, we modify $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA},5}$ to use the original predicate **adv-access-allowed**, denoted as $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA},6}$. $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA},6}$ halts if an adversary succeeds to fetch or publish a proposal or commit message without knowing the corresponding group secret key. Simulator $\mathcal{S}_6$ is identical to $\mathcal{S}_5$. At this point, $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA},6}$ is identical to the ideal functionality $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA}}$.

We show indistinguishability of Hybrids 1 to 6 in Lemmata 5.6.2 to 5.6.4, 5.6.6 and 5.6.8. This completes the proof of the main theorem. □

From Hybrid 1 to 2: Lemma 5.6.2.

Lemma 5.6.2. *Hybrid 1 and Hybrid 2 are perfectly indistinguishable.*

Proof. We first provide the description of $\mathcal{S}_2$. Whenever $\mathcal{S}_2$ is invoked on an input from $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ called within $\mathcal{F}_{\text{CGKA},2}^{\text{mh}}$,[37] it simply relays the same input to the adversary $\mathcal{A}$. Since W^{mh} is constructed in the $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$-hybrid model, these inputs are exactly what $\mathcal{A}$ was provided in Hybrid 1. $\mathcal{S}_2$ returns to $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ whatever provided by $\mathcal{A}$. It remains to describe how $\mathcal{S}_2$ answers `RegisterGroup`, and publish and fetch queries sent from $\mathcal{F}_{\text{CGKA},2}^{\text{mh}}$. To this end, we first make a detour and explain how k_{mh} is set in Hybrids 1 and 2.

In Hybrid 1, k_{mh} for a commit node node-id is created when $\mathcal{S}_1$ invokes some party id with $\mathsf{Ptr}[\mathsf{id}] = \mathsf{node\text{-}id}$ on a `Create` or a `Commit`. When id queries `Create`, id performs a query $\texttt{Key}_{\texttt{mh}}$ to $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ within `*init-states`; when id queries `Commit`, id performs a query $\texttt{NextKey}_{\texttt{mh}}$ within `PublishCommit`. If $\textbf{safe}(\mathsf{node\text{-}id}) = \texttt{true}$, then $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ samples a random key k_{mh}. Otherwise, $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ asks $\mathcal{A}$ to provide the key. These are the only two places where a new k_{mh} is set — $\texttt{Key}_{\texttt{mh}}$ and $\texttt{NextKey}_{\texttt{mh}}$ are also queried during `Propose`, `Process`, and `Join` but k_{mh} will already be defined.

In Hybrid 2, however, $\mathcal{S}_2$ can no longer directly invoke a party id when the static metadata is hidden. For example, when $\mathcal{Z}$ invokes party $\mathsf{id}_{\mathsf{creator}}$ on a `Create` while ***leak-create** is activated (i.e., the predicate **safe** is true), then $\mathcal{S}_2$ is only provided with $(|\mathsf{id}_{\mathsf{creator}}|, |\mathsf{svk}|)$ from $\mathcal{F}_{\text{CGKA},2}^{\text{mh}}$. Without knowing $\mathsf{id}_{\mathsf{creator}}$, $\mathcal{S}_2$ cannot invoke $\mathsf{id}_{\mathsf{creator}}$ to execute `*init-states`. This in particular means that it cannot set k_{mh} like $\mathcal{S}_1$ did above by invoking $\texttt{Key}_{\texttt{mh}}$ or $\texttt{NextKey}_{\texttt{mh}}$ of $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$.

With this in mind, we now finish the description of $\mathcal{S}_2$. $\mathcal{S}_2$ maintains a list $L_{\mathsf{k}_{\mathsf{mh}}}$ to store the key k_{mh} generated at node node-id, i.e., $L_{\mathsf{k}_{\mathsf{mh}}}[\mathsf{node\text{-}id}] = \mathsf{k}_{\mathsf{mh}}$. Notice that when $\mathcal{S}_2$ needs to simulate a `RegisterGroup`, or a fetch or publish query, it is given either $(\mathsf{gid}, \mathsf{epoch})$ if the server Sv is honest and $\mathsf{Ptr}[\mathsf{id}]$ if Sv is malicious. Since there is no fork in the group when Sv is honest, $(\mathsf{gid}, \mathsf{epoch})$ uniquely defines node-id. Notably, even if $\mathcal{S}_2$ does not know who the calling party id is, it knows which group and epoch (or $\mathsf{Ptr}[\mathsf{id}]$ in case of a fork) id is included in, and thus, knows which $\mathsf{k}_{\mathsf{mh}} = L_{\mathsf{k}_{\mathsf{mh}}}[\mathsf{node\text{-}id}]$ to use if it exists.

As explained above, when $\mathcal{Z}$ invokes some party to perform `Propose`, `Process`, or `Join`, k_{mh} is already defined. Thus, $\mathcal{S}_2$ simply uses $\mathsf{k}_{\mathsf{mh}} = L_{\mathsf{k}_{\mathsf{mh}}}[\mathsf{node\text{-}id}]$ to perform the same simulation as $\mathcal{S}_1$, where recall id is not used during a publish or fetch protocol. When the environment $\mathcal{Z}$ invokes $\mathsf{id}_{\mathsf{creator}}$ on $(\texttt{Create}, \mathsf{svk})$, this is when a new k_{mh} is generated, i.e., $L_{\mathsf{k}_{\mathsf{mh}}}[\mathsf{Ptr}[\mathsf{id}_{\mathsf{creator}}]]$ is still undefined. $\mathcal{S}_2$ checks if $\textbf{safe}(0) = \texttt{false}$ — which it can do since it can simulate an identical semantics of the history graph maintained within $\mathcal{F}_{\text{CGKA},2}^{\text{mh}}$ — and asks $\mathcal{A}$ for k_{mh}. This is identical to what $\mathcal{S}_1$ did. Otherwise, $\mathcal{S}_2$ samples a random k_{mh} on its own. In either case, $\mathcal{S}_2$ stores $L_{\mathsf{k}_{\mathsf{mh}}}[0] \leftarrow \mathsf{k}_{\mathsf{mh}}$ and uses k_{mh} to perform the `RegisterGroup` protocol. The only difference between the previous hybrid is that $\mathcal{S}_2$ samples k_{mh} when $\textbf{safe}(0) = \texttt{true}$, rather than letting $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ sampling it. However, since $\texttt{*mark-next-db-initialized-epoch}(0)$ is called and due to the restriction on $\mathcal{A}$, $\mathcal{A}$ cannot compromise k_{mh} after `Create` was invoked. Therefore, from the view of $\mathcal{Z}$, k_{mh} is distributed identically in both hybrids. Thus, the simulation of `Create` is perfectly indistinguishable from Hybrid 1. The case when the environment $\mathcal{Z}$ invokes id on $(\texttt{Commit}, \mathsf{svk})$ is proven analogously to $(\texttt{Create}, \mathsf{svk})$. This completes the proof. □

From Hybrid 2 to 3: Lemma 5.6.3.

[37] Note that this is not strictly true since $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$ is not defined within $\mathcal{F}_{\text{CGKA},2}^{\text{mh}}$. $\mathcal{F}_{\text{CGKA},2}^{\text{mh}}$ merely has most of the codes included in $\mathcal{F}_{\text{CGKA}}^{\text{ctxt}}$. We use this informal wording for simplicity.

Subroutine `RegisterGroup`(gid, epoch)

channelType $\leftarrow$ ('anon', $\perp$, Sv) // Connect to the server via anonymous channel
req epoch $= 0$
Send (channelType, `RegisterGroup`, gid, 0) to $\mathcal{S}$ and receive `accept`
if `ServerStat` $=$ 'good' **then**
  Send (`accept`, gid, 0) to Sv
return `accept`

Subroutine `PublishProposal`(gid, epoch, p)

channelType $\leftarrow$ ('anon', $\perp$, Sv) // Connect to the server via anonymous channel
if `ServerStat` $=$ 'good' **then**
  Send (channelType, `PublishProposal`, gid, epoch, p) to $\mathcal{S}$ and receive `accept`
  Send (`accept`, gid, epoch, p) to Sv
else
  Send (channelType, `PublishProposal`, Ptr[id], p) to $\mathcal{S}$ and receive `accept`
return `accept`

Subroutine `FetchProposals`(gid, epoch)

channelType $\leftarrow$ ('anon', $\perp$, Sv) // Connect to the server via anonymous channel
if `ServerStat` $=$ 'good' **then**
  Send (channelType, `FetchProposals`, gid, epoch) to $\mathcal{S}$ and receive (`accept`, $\vec{p}$)
  Send (`accept`, gid, epoch) to Sv
else
  Send (channelType, `FetchProposals`, Ptr[id]) to $\mathcal{S}$ and
    receive (`accept`, $\vec{p}$)
return (`accept`, $\vec{p}$)

FIGURE 5.52: Subroutines for publishing and fetching proposal messages used in Hybrid 2.

Subroutine $\texttt{PublishCommit}(\mathsf{gid}, \mathsf{epoch}, c_0, \vec{c}_{\mathsf{perm}})$

1 : $\mathsf{channelType} \leftarrow (\text{'anon'}, \perp, \mathsf{Sv})$ // Connect to the server via anonymous channel
2 : **if** $\texttt{ServerStat} = \text{'good'}$ **then**
3 : Send $(\mathsf{channelType}, \texttt{PublishCommit}, \mathsf{gid}, \mathsf{epoch}, c_0, \vec{c}_{\mathsf{perm}})$ to $\mathcal{S}$ and receive $\texttt{accept}$
4 : Send $(\texttt{accept}, \mathsf{gid}, \mathsf{epoch}, c_0, \vec{c}_{\mathsf{perm}})$ to Sv
5 : **else**
6 : Send $(\mathsf{channelType}, \texttt{PublishCommit}, \mathsf{Ptr}[\mathsf{id}], c_0, \vec{c}_{\mathsf{perm}})$ to $\mathcal{S}$ and receive $\texttt{accept}$
7 : **return** $\texttt{accept}$

Subroutine $\texttt{FetchCommit}(\mathsf{gid}, \mathsf{epoch}, \mathsf{index}_{\widehat{c}})$

1 : $\mathsf{channelType} \leftarrow (\text{'anon'}, \perp, \mathsf{Sv})$ // Connect to the server via anonymous channel
2 : **if** $\texttt{ServerStat} = \text{'good'}$ **then**
3 : Send $(\mathsf{channelType}, \texttt{FetchCommit}, \mathsf{gid}, \mathsf{epoch}, \mathsf{index}_{\widehat{c}})$ to $\mathcal{S}$ and receive $(\texttt{accept}, c_0, \widehat{c}, \vec{p})$
4 : Send $(\texttt{accept}, \mathsf{gid}, \mathsf{epoch}, \mathsf{index}_{\widehat{c}})$ to Sv
5 : **else**
6 : Send $(\mathsf{channelType}, \texttt{FetchCommit}, \mathsf{Ptr}[\mathsf{id}], \mathsf{index}_{\widehat{c}})$ to $\mathcal{S}$ and receive $(\texttt{accept}, c_0, \widehat{c}, \vec{p})$
7 : **return** $(\texttt{accept}, c_0, \widehat{c}, \vec{p})$

FIGURE 5.53: Subroutines for publishing and fetching commit messages used in Hybrid 2.

Input from a party id **or adversary** $\mathcal{S}$
Input $\texttt{PublishWelcome}(\mathsf{id}_t, \widehat{w})$

1 : // Connect to the server via anonymous channel
2 : $\mathsf{channelType} \leftarrow (\text{'anon'}, \perp, \mathsf{Sv})$
3 : Send $(\mathsf{channelType}, \texttt{PublishWelcome}, \mathsf{id}_t, \widehat{w})$ to $\mathcal{S}$ and receive $\texttt{accept}$
4 : **if** $\texttt{ServerStat} = \text{'good'}$ **then**
5 : Send $(\texttt{accept}, \mathsf{id}_t, \widehat{w})$ to Sv
6 : **return** $\texttt{accept}$

Input $\texttt{FetchWelcome}(\mathsf{id})$

1 : // Connect to the server via authenticated channel
2 : $\mathsf{channelType} \leftarrow (\text{'auth'}, \mathsf{id}, \mathsf{Sv})$
3 : Send $(\mathsf{channelType}, \texttt{FetchWelcome})$ to $\mathcal{S}$ and receive $(\texttt{accept}, \widehat{w})$
4 : **if** $\texttt{ServerStat} = \text{'good'}$ **then**
5 : Send $(\texttt{accept}, \mathsf{id})$ to Sv
6 : **return** $(\texttt{accept}, \widehat{w})$

FIGURE 5.54: Subroutines for publish and fetch welcome messages used in Hybrid 2.

***permuted-commit-index**(node-id, id)

1 : **assert** Node[node-id] $\neq \perp$
2 : // If $\texttt{flag}_{\mathsf{selDL}}$ is false, there is no need to permute indexes.
3 : **if** $\neg\texttt{flag}_{\mathsf{selDL}}$ **then**
4 : **return** $\perp$
5 : mem $\leftarrow$ Node[node-id].mem
6 : **assert** id $\in$ mem
7 : Send (*permuted-commit-index, Ptr[id], id) to $\mathcal{S}$ and receive $\text{index}_{\widehat{c}}$
8 : **return** $\text{index}_{\widehat{c}}$

***permute-commit**(node-id, $\vec{c}$)

1 : **assert** Node[node-id] $\neq \perp$
2 : // If $\texttt{flag}_{\mathsf{selDL}}$ is false,
3 : // no party-dependent commitment can exist.
4 : **if** $\neg\texttt{flag}_{\mathsf{selDL}}$ **then**
5 : **assert** $\vec{c} = \perp$
6 : **return** $\perp$
7 : Send (*permute-commit, Ptr[id], $\vec{c}$) to $\mathcal{S}$ and receive $\vec{c}_{\mathsf{perm}}$
8 : **return** $\vec{c}_{\mathsf{perm}}$

FIGURE 5.55: Helper functions: permute functions used in Hybrid 2.

Input from the adversary $\mathcal{S}$

Input (PublishProposalAdv, gid, epoch, p)

1 : channelType $\leftarrow$ ('anon', $\perp$, Sv)
2 : Send (channelType, PublishProposalAdv, gid, epoch, p) to $\mathcal{S}$ and receive accept
3 : **if** ServerStat = 'good' **then**
4 : Send (channelType, PublishProposal, (accept, gid, epoch, p)) to Sv
5 : **return** accept

Input (FetchProposalsAdv, gid, epoch)

1 : channelType $\leftarrow$ ('anon', $\perp$, Sv)
2 : Send (channelType, FetchProposalsAdv, gid, epoch) to $\mathcal{S}$ and receive (accept, $\vec{p}$)
3 : **if** ServerStat = 'good' **then**
4 : **return** (accept, $\vec{p}$)

Input (PublishCommitAdv, gid, epoch, $c_0, \vec{c}$)

1 : channelType $\leftarrow$ ('anon', $\perp$, Sv)
2 : Send (channelType, PublishCommitAdv, gid, epoch, $c_0, \vec{c}$) to $\mathcal{S}$ and receive accept
3 : **if** ServerStat = 'good' **then**
4 : **return** accept

Input (FetchCommitAdv, gid, epoch, index)

1 : channelType $\leftarrow$ ('anon', $\perp$, Sv)
2 : Send (channelType, FetchCommitAdv, gid, epoch, index) to $\mathcal{S}$ and receive (accept, $c_0, \widehat{c}, \vec{p}$)
3 : **if** ServerStat = 'good' **then**
4 : **return** (accept, $c_0, \widehat{c}, \vec{p}$)

FIGURE 5.56: Publish functions used by the adversary in Hybrid 2.

Lemma 5.6.3. *Hybrid 2 and Hybrid 3 are indistinguishable assuming the correctness of* SIG'.

Proof. The only difference between Hybrids 2 and 3 is that $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA},3}$ in Hybrid 3 mandates correctness of the `RegisterGroup`, and publish and fetch protocols when the server Sv is honest. For instance, in Hybrid 2, during the `PublishProposal` protocol, $\mathcal{S}_2$ simulated the interaction between the party id and the honest Sv using the key $\mathsf{k_{mh}}$ stored in the list $L_{\mathsf{k_{mh}}}$. $\mathcal{S}_2$ then returned $\texttt{accept}'$ output by Sv to $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA},2}$, and in particular, this notified the environment $\mathcal{Z}$ that $\mathcal{S}$ output $\texttt{accept}'$. In particular, $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA},2}$ did not model any notion of correctness of the `PublishProposal` protocol.

On the other hand, in Hybrid 3, if $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] = (\top, \mathsf{node\text{-}id})$ and $\mathsf{node\text{-}id} = \mathsf{Ptr}[\mathsf{id}]$ (i.e., the group statement gvk is created honestly and id holds the corresponding signing key gsk), then $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA},3}$ always sends $\texttt{accept} = \texttt{true}$ to $\mathcal{Z}$. This models the fact that if the new epoch ($\mathsf{node\text{-}id}$) is initialized by an honest party, then any member assigned on the same commit node (i.e., $\mathsf{node\text{-}id} = \mathsf{Ptr}[\mathsf{id}]$) can upload a proposal. It is clear that this holds assuming that SIG' is correct, i.e., a properly generated signature is always accepted. $\mathcal{F}^{\mathsf{mh}}_{\mathrm{CGKA},3}$ always sends $\texttt{accept} = \texttt{false}$ to $\mathcal{Z}$ if (1) $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] \neq (\top, \mathsf{node\text{-}id})$ or (2) $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}] = (\top, \mathsf{node\text{-}id})$ and $\mathsf{node\text{-}id} \neq \mathsf{Ptr}[\mathsf{id}]$. For case (1), we can verify that the real server performs the same check as $\mathsf{ComDB}[\mathsf{gid}, \mathsf{epoch}]$ is equal to $(\mathsf{gvk}, \bot, \bot)$ or not, and thus $\mathcal{S}_2$ returned $\texttt{accept}' = \texttt{false}$. For case (2), if $\mathcal{S}_2$ returned $\texttt{accept}' = \texttt{true}$ in case (2), we can construct an adversary that breaks the EUF-CMA security of SIG' by using such $\mathcal{S}_2$. Thus $\mathcal{S}_2$ returned $\texttt{accept}' = \texttt{false}$ assuming the EUF-CMA security of SIG'. Therefore, the view to $\mathcal{Z}$ remain identical.

All other `RegisterGroup`, and publish and fetch protocols in Hybrid 3 can be checked to behave identically to those of Hybrid 2 conditioned on SIG' being correct and EUF-CMA secure. □

From Hybrid 3 to 4: Lemma 5.6.4.

Lemma 5.6.4. *Hybrid 3 and Hybrid 4 are indistinguishable assuming* PRF *is a secure pseudorandom function.*

Proof. We assume $\mathcal{Z}$ creates at most Q epochs (i.e., commit nodes $\mathsf{node\text{-}id}$ in the history graph). To show Lemma 5.6.4, we consider the following sub-hybrids between Hybrid 3 and Hybrid 4.

Hybrid 3-0 := Hybrid 3. This is identical to Hybrid 3. We use the functionality $\mathcal{F}_{\mathsf{CGKA},3}$, and the simulator $\mathcal{S}_{3\text{-}0} := \mathcal{S}_3$. When simulating protocol, $\mathcal{S}_{3\text{-}0}$ uses the group key $\mathsf{k_{mh}}$ stored in the list $L_{\mathsf{k_{mh}}}$ as PRF key.

Hybrid 3-*i*. i runs through $[Q]$. The simulator $\mathcal{S}_{3\text{-}i}$ is defined exactly as $\mathcal{S}_{3\text{-}(i-1)}$ except that when $\mathcal{S}_{3\text{-}(i-1)}$ generates a random $\mathsf{k_{mh}}$ for the i-th epoch where **safe** is true, it chooses a random $\mathsf{authKey}$ and $\mathsf{permKey}$ instead of deriving them from PRF and $\mathsf{k_{mh}}$. Note that we count epochs (i.e., $\mathsf{node\text{-}id}$) in the order in which `Create` or `Commit` is invoked. We show in Lemma 5.6.5 that Hybrid 3-$(i-1)$ and Hybrid 3-i are indistinguishable.

Hybrid 3-Q := Hybrid 4. In this hybrid, all $\mathsf{authKey}$ and $\mathsf{permKey}$ generated at epochs such that **safe** is true are chosen at random.

The indistinguishability between Hybrid 3 and Hybrid 4 is established by applying the following Lemma 5.6.5 for all $i \in [Q]$.

Lemma 5.6.5. *Hybrid 3-$(i-1)$ and Hybrid 3-i are indistinguishable assuming* PRF *is a secure pseudo-random function.*

Proof. The difference between Hybrid 3-$(i-1)$ and Hybrid 3-i is whether authKey and permKey at the i-th epoch (i.e., node-id) are chosen uniformly at random if **safe** is true. (If **safe** is false for the i-th epoch, two hybrids proceed identically.) Due to the modification we made in Hybrid 2, the group key k_{mh} is chosen uniformly at random. Moreover, due to the restriction on the adversary, k_{mh} generated when **safe** was true cannot be corrupted by the adversary. Thus, by the pseudorandomness of the PRF, the output of the PRF is indistinguishable from a random string for the input labels 'perm' and 'auth'. This implies Hybrid 3-$(i-1)$ and Hybrid 3-i are indistinguishable. More formally, if $\mathcal{Z}$ can distinguish the two hybrids, then there exists an adversary $\mathcal{B}$ that breaks the pseudorandomness of the PRF. We first explain the description of $\mathcal{B}$; then we evaluate $\mathcal{B}$'s advantage.

$\mathcal{B}$ simulates for $\mathcal{Z}$ the protocol executions between a party and the server as in $\mathcal{S}_{3\text{-}(i-1)}$, except for the evaluation of the PRF. To generate authKey and permKey, $\mathcal{B}$ queries $\mathsf{authKey} := \mathcal{F}(\text{'auth'})$ and $\mathsf{permKey} := \mathcal{F}(\text{'perm'})$ to its oracle $\mathcal{F}$.

We evaluate the success probability of $\mathcal{B}$. If the oracle $\mathcal{F}$ evaluates a pseudo-random function, then $\mathcal{Z}$'s view is identical to Hybrid 3-$(i-1)$. If $\mathcal{F}$ evaluates a truly random function, then $\mathcal{Z}$'s view is identical to Hybrid 3-i. Hence, if $\mathcal{Z}$ distinguishes Hybrid 3-$(i-1)$ and Hybrid 3-i with non-negligible probability, $\mathcal{B}$ breaks the pseudorandomness of the PRF with non-negligible probability. This contradicts the assumption that PRF is secure. Therefore, Hybrid 3-$(i-1)$ and Hybrid 3-i are indistinguishable. □

□

From Hybrid 4 to 5: Lemma 5.6.6. Hybrid 5 concerns the randomness of the access indix of group members. If **random-index** is true, then the ideal functionality $\mathcal{F}^{\mathsf{mh}}_{\mathsf{CGKA},5}$ randomizes the commit vector $\vec{c}$ and the index of each party in the group on behalf of the simulator. We prove in Lemma 5.6.6 that assuming that PRP is secure, Hybrid 4 and Hybrid 5 are indistinguishable.

Lemma 5.6.6. *Hybrid 4 and Hybrid 5 are indistinguishable assuming* PRP *is a secure pseudorandom permutation.*

Proof. We assume $\mathcal{Z}$ creates at most Q epochs (i.e., commit nodes node-id in the history graph). To show Lemma 5.6.6, we consider the following sub-hybrids between Hybrid 4 and Hybrid 5.

Hybrid 4-0 := Hybrid 4. This is identical to Hybrid 4. We use the functionality $\mathcal{F}_{\mathsf{CGKA},4}$, and the simulator $\mathcal{S}_{4\text{-}0} := \mathcal{S}_4$. In this hybrid, $\mathcal{F}_{\mathsf{CGKA},4}$ permutes the indices index and commit vectors $\vec{c}$ with a permutation chosen by $\mathcal{S}_{4\text{-}0}$, and $\mathcal{S}_{4\text{-}0}$ simulates the `PublishCommit` and `FetchCommit` protocols using the indices and commit messages provided from $\mathcal{F}_{\mathsf{CGKA},4\text{-}i}$.

Hybrid 4-i. i runs through $[Q]$. We use the functionality $\mathcal{F}_{\mathsf{CGKA},4\text{-}i}$ which is defined exactly as $\mathcal{F}_{\mathsf{CGKA},4\text{-}(i-1)}$ except that the indices index and commit vectors $\vec{c}$ are permutated with a truly random permutation if **random-index** is true for the i-th epoch. The simulator $\mathcal{S}_{4\text{-}i}$ is defined exactly as $\mathcal{S}_{4\text{-}(i-1)}$. Note that we count epochs (i.e., node-id) in the order in which `Create` or `Commit` is invoked. We show in Lemma 5.6.7 that Hybrid 4-$(i-1)$ and Hybrid 4-i are indistinguishable.

Hybrid 4-Q := Hybrid 5. We use the functionality $\mathcal{F}_{\mathsf{CGKA},4\text{-}Q}$ which permutes the indices index and commit vectors $\vec{c}$ with a random permutation for all epochs where **random-index** is true. $\mathcal{F}_{\mathsf{CGKA},4\text{-}Q}$ is identical to $\mathcal{F}_{\mathsf{CGKA},5}$.

Indistinguishability between Hybrid 4 and Hybrid 5 is established by applying the following Lemma 5.6.7 for all $i \in [Q]$.

Lemma 5.6.7. *Hybrid 4-$(i-1)$ and Hybrid 4-i are indistinguishable assuming* PRP *is a secure pseudorandom permutation.*

Proof. The difference between Hybrid 4-$(i-1)$ and Hybrid 4-i is whether index and $\vec{c}$ issued at the i-th epoch where **random-index** is true are permuted by a truly random permutation. (If **random-index** is false for the i-th epoch, two hybrids proceed identically.) Due to the modification we made in Hybrid 4, the i-th permutation key permKey is chosen uniformly at random if **random-index** (=**safe**) is true. Thus, by the pseudorandomness of the PRP, the output of the PRP is indistinguishable from that of a truly random permutation. This implies Hybrid 4-$(i-1)$ and Hybrid 4-i are indistinguishable. More formally, if $\mathcal{Z}$ can distinguish the two hybrids, then we can construct an adversary $\mathcal{B}$ that breaks the pseudorandomness of the PRP. We first explain the description of $\mathcal{B}$; then we evaluate $\mathcal{B}$'s advantage.

$\mathcal{B}$ simulates for $\mathcal{Z}$ the protocol executions between a party and the server as in $\mathcal{S}_{4\text{-}(i-1)}$, except for the evaluation of PRP at the i-th epoch. Rather than sampling a permutation key permKey for the i-th epoch to evaluate the permutation, $\mathcal{B}$ queries its challenge oracle to compute $\mathsf{index}' \leftarrow \mathcal{P}(\mathsf{index})$.

We evaluate the success probability of $\mathcal{B}$. If the oracle $\mathcal{P}$ evaluates a pseudorandom permutation, then $\mathcal{Z}$'s view is identical to Hybrid 4-$(i-1)$. Otherwise, if $\mathcal{P}$ evaluates a truly random permutation, then $\mathcal{Z}$'s view is identical to Hybrid 4-i. Hence, if $\mathcal{Z}$ distinguishes Hybrid 4-$(i-1)$ and Hybrid 4-i with non-negligible probability, $\mathcal{B}$ breaks the pseudorandomness of the PRP with non-negligible probability. This contradicts the assumption that PRP is secure. Therefore, Hybrid 4-$(i-1)$ and Hybrid 4-i are indistinguishable. □

□

From Hybrid 5 to 6: Lemma 5.6.8. Hybrid 6 concerns the group membership check performed by the honest server. When the server is honest (i.e., `ServerStat` = 'good') and **adv-access-allowed** is false (i.e., group secrets are not compromised), the functionality $\mathcal{F}^{\mathsf{mh}}_{\mathsf{CGKA},6}$ halts if an adversary succeeds to publish or fetch a proposal or commit messages. That is an adversary that does not know the group secret should not be able to publish or fetch contents from the honest server. We prove in Lemma 5.6.8 that if $\mathcal{Z}$ can distinguish the two hybrids, then we can construct an adversary that breaks the EUF-CMA security of SIG′. In other words, assuming that SIG′ is EUF-CMA secure, Hybrid 5 and Hybrid 6 are indistinguishable. Concretely, we show the following.

Lemma 5.6.8. *Hybrid 5 and Hybrid 6 are indistinguishable assuming* SIG′ *is* EUF-CMA *secure.*

Proof. We show that, if $\mathcal{Z}$ can distinguish the two hybrids, then there exists an adversary $\mathcal{B}$ that breaks the EUF-CMA security of SIG′. We first explain the description of $\mathcal{B}$ and how $\mathcal{B}$ extracts a valid signature forgery using $\mathcal{Z}$; we then show the validity of the forged signature and finally evaluate $\mathcal{B}$'s advantage.

$\mathcal{B}$ simulates for $\mathcal{Z}$ the protocol executions between a party and the server as in $\mathcal{S}_6$, except for the signature generation during the publish and fetch protocols. At the beginning of the game, $\mathcal{B}$ chooses an index $i \in [Q]$ at random, where Q is the total number of epochs (i.e., commit nodes node-id) that $\mathcal{Z}$ creates. $\mathcal{B}$ aborts if **adv-access-allowed** is `true` for the i-th epoch, and otherwise embeds the challenge signing key svk^* in the i-th group signing key gvk. Here, if **adv-access-allowed** is false (i.e, **safe** is true) for the i-th epoch, the group signing key is generated with a uniformly random authKey due to the modification we made in Hybrid 4. Therefore, $\mathcal{B}$ perfectly simulates svk^* as in Hybrid 5. Moreover, note that once **safe** is true, the restricted adversary cannot compromise the i-th group signing key. Whenever an honest party at the i-th epoch publishes or fetches proposal/commit messages, $\mathcal{B}$ uses the signing oracle to sign the challenge message sent from the server. For the other epochs, $\mathcal{B}$ generates gvk and signs as in the

previous hybrid. If the adversary succeeds in making the server accept at execution of a publish and fetch proposal/commit protocols in the i-th epoch, $\mathcal{B}$ retrieves the pair of challenge message and response signature (ch, σ) from the successful transcript, and submits (ch, σ) as the forgery.

Let us analyze the success probability of $\mathcal{B}$. First, by noticing the only way that $\mathcal{Z}$ can distinguish the two hybrids is by triggering the **assert** condition on **adv-access-allowed**, there must exist at least one epoch for which the adversary succeeds in the protocol `PublishProposalAdv`, where **adv-access-allowed** is false. Since the choice $i \leftarrow\$ [Q]$ of $\mathcal{B}$ is information-theoretically hidden from $\mathcal{Z}$ and the adversary, the guess made by $\mathcal{B}$ is correct with probability at least $1/Q$. Conditioning on the guess being correct, (ch, σ) is a valid message and signature pair for gvk. Moreover, since the server is honest and the challenge message space is exponentially large, the server never picks the same challenge message ch. Therefore, (ch, σ) is a pair of message and signature that $\mathcal{B}$ did not query to the signing oracle and constitutes a valid forgery. In summary, if $\mathcal{Z}$ distinguishes the two hybrids with non-negligible probability ϵ, $\mathcal{B}$ wins the game with probability at least ϵ/Q, which is also non-negligible. This contradicts the assumption that SIG' is EUF-CMA secure. Thus, Hybrid 5 and Hybrid 6 are indistinguishable. □

5.7 Instantiation and Efficiency of Proposed Metadata-Hiding CGKA

We now discuss how to instantiate W^{mh} and Chained $\mathsf{CmPKE}^{\mathsf{ctxt}}$. We target the so-called "NIST Level I"[38], which informally states that breaking the protocol is no easier than key-recovery on a block cipher with a 128-bit key (e.g. AES128). This provides a meaningful baseline to discuss post-quantum security and ignoring quantum attacks corresponds to a classical security level of 128 bits. Given a cryptographic object x, we note $|\mathsf{x}|$ its size in bytes.

5.7.1 Instantiation

The main cryptographic primitives used in the proposed protocols are: (a) two signature schemes SIG and SIG', (b) a multi-recipient PKE mPKE, and finally (c) symmetric primitives: a pseudo-random function PRF, a symmetric encryption scheme SKE and a pseudo-random permutation PRP.

Multi-recipient PKE. For pre-quantum security, one may combine Kurosawa's multi-recipient variant [Kur02] of ElGamal with the transform of [Kat+20], whose decomposability property makes it amenable to the selective downloading performed by Chained $\mathsf{CmPKE}^{\mathsf{ctxt}}$. For post-quantum security, one may readily use one of the mPKEs proposed in [Has+21b]: Ilum512, Bilbo640 or a SIKEp434-based mPKE. Their performance profiles can be found in Table 4.6.

Signature schemes. We choose signature schemes that complement our chosen mPKEs nicely, either by having similar performances, being based on similar assumptions, or both:

- The ECDSA standard is based on similar assumptions as of the ElGamal-based mPKE, and it has a similar performance profile as well.
- The NIST PQC finalist Falcon [Pre+20] complements our SIKEp434-based mPKE, as they both have small communication costs.
- The NIST PQC finalist Dilithium [Lyu+20] is based on Module-LWE (plus Module-SIS), just like Ilum512.

[38] `https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/evaluation-criteria/security-(evaluation-criteria)`

- $\mathsf{SPHINCS}^+$ [Hul+20] and Bilbo640 are both based on assumptions perceived as conservative (hash-based assumptions and unstructured LWE), and they both have comparatively larger communication costs than other schemes.

For simplicity, we consider that SIG and SIG' use the same scheme, but using distinct schemes may lead to interesting trade-offs.

Pseudo-random permutation. We require a PRP in order to permute the set of $[N]$ group members. There are at least two viable approaches:

- *Shuffling*. One may use a shuffling algorithm whose randomness is provided by passing $\mathsf{permKey}$ into a PRF. If cache attacks on group members' devices are not a concern, the Fisher-Yates shuffle is a good choice since it is simple, performs $N-1$ swaps, and its entropy consumption is optimal. If cache attacks are a concern, one may need to use so-called *oblivious* algorithms. For example, with the Thorp shuffle [MRS18], each member may compute their permuted index in time $O(\log N)$.
- *Sorting*. This approach assigns to each member id a pseudo-random value $r_{\mathsf{id}} = \mathsf{H}(\mathsf{permKey}, \mathsf{id})$, then sorts the id's according to the r_{id}'s. The sorting step can be done obliviously in time $O(N \log^2 N)$ using sorting networks, for example, Batcher networks. Assuming H is collision-resistant, this provides a PRP over $[N]$.

While the solutions proposed above are not optimal when used in `*permuted-commit-index` (each group member needs to shuffle/sort *all* indices before finding their position), we expect them to be significantly less costly than public-key cryptographic operations, even for concretely large groups.

5.7.2 Efficiency

We now study the impact of W^{mh} on bandwidth efficiency, by applying it to Chained $\mathsf{CmPKE}^{\mathsf{ctxt}}$. The results are summarized in Table 5.8, with the overhead of W^{mh} represented in bold red font **(+X)**.

TABLE 5.8: Bandwidth overhead of our wrapper W^{mh} applied to Chained $\mathsf{CmPKE}^{\mathsf{ctxt}}$, for a group of N members in terms of public key cryptography elements. The nominal bandwidth cost of Chained $\mathsf{CmPKE}^{\mathsf{ctxt}}$ is written in normal font (X). The overhead of W^{mh} is written in bold red **(+X)**. U (resp. A, R) stands for the number of 'upd' (resp. 'add', 'rem') proposals published during the last epoch.

	Upload					Download				
Procedure	$\lvert\mathsf{ek}\rvert$	$\lvert\mathsf{ct}_0\rvert$	$\lvert\widehat{\mathsf{ct}}_{\mathsf{id}}\rvert$	$\lvert\mathsf{sig}\rvert$	$\lvert\mathsf{svk}\rvert$	$\lvert\mathsf{ek}\rvert$	$\lvert\mathsf{ct}_0\rvert$	$\lvert\widehat{\mathsf{ct}}_{\mathsf{id}}\rvert$	$\lvert\mathsf{sig}\rvert$	$\lvert\mathsf{svk}\rvert$
`Propose-'upd'`	1			2 **(+1)**						
`Propose-'add'`	1			1 **(+1)**	1					
`Propose-'rem'`				1 **(+1)**						
`Commit`	1	1	N	2 **(+2)**	**(+1)**	U+A			2U+A+R	A
`Process`				**(+1)**		U+A+1	1	1	2U+A+R+2	A
Applications messages				1 **(+1)**						

Let us make two observations. First, W^{mh} adds the same overhead to any CGKA protocol realizing the ideal functionality $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$ (e.g., our Chained $\mathsf{CmPKE}^{\mathsf{ctxt}}$ and MLSCiphertext[39]). Second, as is obvious in

[39] Although no formal proof exists, it is believed that MLSCiphertext realizes $\mathcal{F}^{\mathsf{ctxt}}_{\mathrm{CGKA}}$.

TABLE 5.9: Concrete overhead of W^{mh} in terms of public key cryptography elements, when instantiated with: (a) ElGamal-mPKE + ECDSA (E+E), (b) SIKEp434-mPKE + Falcon (S+F), (c) Ilum512 + Dilithium (I+D), (d) Bilbo640 + SPHINCS$^+$ (B+S). We assume a group of $N = 256$ members, and numbers for `Commit` and `Process` assume no proposal was made during the last epoch. The nominal bandwidth cost of Chained CmPKE$^{\mathsf{ctxt}}$ is written in normal font (X), and the overhead of W^{mh} is written in bold red (**+X**). All sizes are in bytes.

Procedure	E+E		S+F		I+D		B+S	
`Propose-'upd'`	160	**+64**	1 662	**+666**	5 608	**+2 420**	44 416	**+17 088**
`Propose-'add'`	128	**+64**	1 893	**+666**	4 500	**+2 420**	27 360	**+17 088**
`Propose-'rem'`	64	**+64**	666	**+666**	2 420	**+2 420**	17 088	**+17 088**
`Commit`	8 384	**+160**	6 088	**+2 229**	18 600	**+6 152**	60 800	**+34 208**
`Process`	224	**+64**	2 008	**+666**	6 360	**+2 420**	54 680	**+17 088**
Applications messages	64	**+64**	666	**+666**	2 420	**+2 420**	17 088	**+17 088**

Table 5.8, the bandwidth overhead is added only in the *upload* direction, since the additional signatures serve to convince the server that a user is a legitimate group member.

For a more concrete perspective on the numbers provided in Table 5.8, we study the impact of applying W^{mh} over Chained CmPKE$^{\mathsf{ctxt}}$, when these protocols are instantiated with the four mPKE-signature pairs selected in Section 5.7.1. The results are given in Table 5.9. For our examples, W^{mh} increases the bandwidth cost of `Propose-'upd'`, `Propose-'add'`, `Propose-'rem'`, `Commit`, `Process` and application messages by at most 44%, 63%, 100%, 57%, 39% and 100%, respectively.[40] We believe this to be a very reasonable overhead if protecting metadata is considered important.

5.8 Limitation of Efficient Metadata-Hiding CGKA

We conclude this work by discussing some inherent limitations of metadata-hiding CGKAs. Our model and solution hide metadata from messages but allow leaking message size. This may reveal information about the structure and activity of the group. While remaining informal, our discussion highlights that the nature and extent of this information depend on the inner workings of the CGKA, as well as the precise topology of the group at a given time (especially for tree-based CGKAs).

5.8.1 Chained CmPKE and TreeKEM

We first discuss Chained CmPKE [Has+21b], as well as variants of TreeKEM *without server assistance* [Bar+22; Alw+20a; Alw+21a; AJM22; Kle+21], meaning that the server forwards messages to group members without editing them.

Leakage from proposal messages. When the size of proposal messages depends on their type, these messages leak the proposal type (i.e., 'upd', 'add', 'rem'). For Chained CmPKE, this is evident from Table 5.9. Even though this can be fixed by padding, we note that at least for Chained CmPKE, commit messages subsequent to a successful 'add' (resp. 'rem') will increase (resp. decrease) by $|\widehat{\mathsf{ct}}_{\mathsf{id}}|$ bytes. For TreeKEM, a

[40] For simplicity, numbers for `Commit` and `Process` in Table 5.9 consider an idealized setting where no proposal was made during the last epoch. In practice, this is not the case and the bandwidth overhead of W^{mh} for `Commit` and `Process` will be even lower in percentage.

less obvious yet similar correlation exists. On the bright side, the size of proposals is independent of the group size and the sender's identity.

Leakage from commit messages.

Chained CmPKE [Has+21b]. The size of an uploaded commit message is affine in the group size N. Thus, the server can infer N from commit messages. Chained CmPKE allows selective downloading but the wrapper W^{mh} hides the index of each party-dependent message by randomizing indices via per-epoch random permutation. Thus, from the server's point of view, indices look random.

TreeKEM [Bar+22; Alw+20a; Alw+21a; AJM22]. The number of PKE ciphertexts in a commit message depends on the topology of the ratchet tree and the sender's position in this tree, therefore the size of a commit message can leak information about both elements. We provide two examples.

First, suppose that the tree is full (i.e., no blank node) but the group size is not a power of two. Since TreeKEM uses left-balanced binary trees, parties assigned to leaves "at the left" of the tree will send longer commit messages than the ones "on the right". Hence the server can easily partition the set of parties into two groups depending on the length of commit messages.

Second, we consider the tree in Figure 5.57, which has some blank nodes. In this tree, the number of cryptographic materials sent by each party is as follows: parties 1 and 2 (resp. 3 and 4, resp. 5 and 6, resp. 7) send 3 encryption keys + 5 ciphertexts (resp. $3+4$, resp. $3+3$, resp. $2+2$ encryption keys and ciphertexts).

Tainted TreeKEM [Kle+21]. In this variant of TreeKEM, commit messages contain cryptographic materials related to tainted nodes, nodes managed by a party other than on the direct path. Message size becomes larger if the sender manages more tainted nodes.

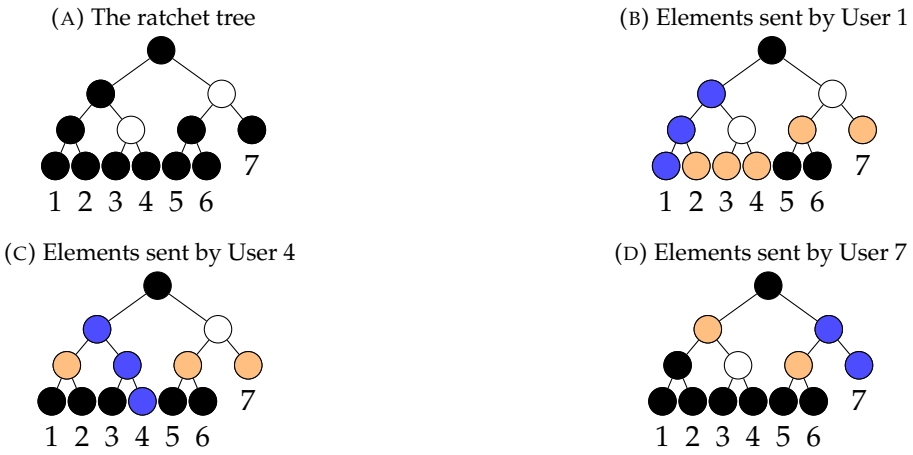


FIGURE 5.57: Example of the TreeKEM ratchet tree with blank nodes. The numbers indicate party identities. Figures 5.57b to 5.57d: sending encryption keys ● and ciphertexts ● when party 1, 4 or 7 commits, respectively.

Leakage from welcome messages. In both protocols, welcome messages leak the receiver's identity, the key package's hash, and the group size. Note that the receiver's identity must be in the clear for the server to deliver welcome messages. Key package hashes are used by the receivers to identify which decryption key is necessary to process the welcome message.

First of all, the server always gets to know when a welcome message was fetched. In addition, if the welcome message did not hide the dynamic metadata (i.e., uploader's identity), then the server can infer

that the party creating the welcome message and the party fetching it belong to the same group. Since groups are created by adding new members through a welcome message, this minor leakage of metadata can be used to trace the entire group member.

However, even if the welcome message hides the dynamic metadata, the server may still be able to link welcome messages to a specific group in some scenarios. For example, assume a party at an insecure $\mathsf{epoch}_{\mathsf{cur}}$ commits an add proposal and moves to a secure $\mathsf{epoch}_{\mathsf{next}}$. Using a client-anonymized authenticated channel to upload the welcome message, the party uploading the welcome message remains anonymous, thus protecting against the above attack. However, since the server gets to see the key package included in the add proposal issued at the insecure $\mathsf{epoch}_{\mathsf{cur}}$, the server can link this key package to the key package hash included in the welcome message to infer that the new member joins the group related to $\mathsf{epoch}_{\mathsf{cur}}$. In particular, while the key package hash included in the welcome message is good for efficiency, it may have non-trivial side effects.

A simple way to prevent such information leakage is to remove key package hashes from welcome messages. However, this would require recipients to try decrypting with all their registered decryption keys since they no longer can determine which decryption key can be used to decrypt the received welcome message. Finally, we note that welcome messages will always leak the size of the group to which the party was newly added. This is because a welcome message in essence sends all the current group states to the newly added party.

5.8.2 Server-Aided Variants of TreeKEM

We discuss recent efficient variations of TreeKEM [Alw+22d; Alw+22c][41]. These allow the server to perform special tasks, e.g., editing signatures or ratchet trees in addition to delivering messages. They improve efficiency by revealing metadata to the server. Devising metadata-hiding variants of these schemes would likely require devising ways to perform these editing operations in an oblivious way, similarly to the PRP-based solution, we proposed for Chained $\mathsf{CmPKE}^{\mathsf{ctxt}}$.

SAIK[42] [Alw+22d]. This TreeKEM variant uses reducible signatures, a variant of redactable signatures, to allow parties to selectively download parts of commit messages while still guaranteeing their validity with the same signature that was initially uploaded by the sender. This improves the overall communication cost. However, the server needs to know the identities of both the sender and receiver in order to reduce the commit message before forwarding it to the receiving party. In SAIK, this reduction is more involved than the selective downloading of Chained CmPKE. Even assuming that the reduction itself can be performed obliviously, the size of the reduced message depends on the positions of both the sender and receiver, so it would still leak information about both parties.

CoCoA[43] [Alw+22c]. This variant allows the server to merge multiple commit messages for the purpose of reducing communication costs. The server keeps the public part of the group state and creates the next group state by merging concurrently issued commit messages. Then, it forwards to each group member the part of the new state they need. This results in $O(\log N)$ upload and download costs, and a $O(\log^2 N)$ total cost. However, the server needs to know the public part of group states, which in particular includes the group member list.

[41] This work analyzes the initial ePrint versions [Alw+21b; Alw+22b].
[42] SAIK stands for Server-Aided Insider-Secure TreeKEM.
[43] CoCoA stands for COncurrent COntinuous group key Agreement.

One can say that these schemes improve efficiency by giving more information to the server. In other words, there is a trade-off between efficiency and privacy. We view it as an interesting research direction to construct more efficient (and practical) CGKA protocols while still protecting metadata.

Chapter 6

Conclusion

This work constructed post-quantum key exchange protocols for secure group messaging and analyze their efficiency and security. Below is the summary of the contributions.

Post-Quantum Authenticated Key Exchange for Signal Protocol. We focus on the Signal protocol and proposed new generic construction of authenticated key exchange that can be used as a replacement for the X3DH protocol. We cast the X3DH protocol as an AKE protocol and formalized its security model based on the vast prior work on AKE protocols. Then, we proposed a generic construction based on key encapsulation mechanisms (KEM) and signature schemes. Since such primitives can be instantiated from well-established post-quantum assumptions, we obtain the first post-quantum secure replacement of the X3DH protocol. Moreover, we evaluated the communication and computation costs of our protocol instantiated from the NIST PQC standardization candidates and showed that it is sufficiently efficient in the real world. This result implies that the first post-quantum Signal protocol is realized from our proposal and the already proposed post-quantum double ratchet protocol. This post-quantum Signal protocol ensures the confidentiality of messages even if large-scale quantum computers will be realized.

Continuous Group Key Agreement via Post-Quantum Multi-Recipient PKE. We design a new post-quantum continuous group key agreement (CGKA) protocol, called Chained CmPKE, from multi-recipient public key encryption (mPKE). Compared with MLS's TreeKEM, Chained CmPKE offers small total communication costs for each user when all users update their key materials. In addition, to reduce the uploading costs of Chained CmPKE, we proposed new lattice-based mPKE schemes. Thanks to this new mPKE, the concrete uploading costs of Chained CmPKE become smaller than TreeKEM's costs if the group size is about hundreds, despite our costs scaling linearly in the group size and TreeKEM's costs scaling logarithmically in the group size. Since Chained CmPKE offers small communication costs, it can efficiently update keys even for mobile phones that may have an upper limit on available data or slow communication speeds (e.g., 3G network).

MetaData-Hiding Continuous Group Key Agreement. We proposed a metadata-hiding CGKA protocol that allows users to continuously key exchange while hiding metadata such as receiver information. We proposed a simple and generic wrapper protocol that converts any non-metadata-hiding CGKA protocol into a metadata-hiding one with minimum overhead. Also, we proved the modified version of Chained CmPKE called Chained $\mathsf{CmPKE}^{\mathsf{ctxt}}$ plus the wrapper protocol is the desirable metadata-hiding CGKA protocol. The proposed protocol enhances users' privacy in secure messaging. This protocol allows users to perform cryptographically secure communications that both messages and conversational partners are kept secret against third parties. That is, even if governments obtain the communication transcripts stored on the server, it is not possible to know who the user was conversing with and the content of the conversation.

Bibliography

[AR04] Dorit Aharonov and Oded Regev. "Lattice Problems in NP cap coNP". In: *45th FOCS*. IEEE Computer Society Press, Oct. 2004, pp. 362–371. DOI: 10.1109/FOCS.2004.35 (cit. on p. 70).

[Ala+20] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, John Kelsey, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. *NISTIR 8309 - Status Report on the Secind Round of the NIST Post-Quantum Cryptography Standardization Process*. 2020. URL: https://csrc.nist.gov/publications/detail/nistir/8309/final (cit. on p. 171).

[ASB14] Janaka Alawatugoda, Douglas Stebila, and Colin Boyd. "Modelling after-the-fact leakage for key exchange". In: *ASIACCS 14*. Ed. by Shiho Moriai, Trent Jaeger, and Kouichi Sakurai. ACM Press, June 2014, pp. 207–216 (cit. on pp. 25, 26).

[Alb+22] Ange Albertini, Thai Duong, Shay Gueron, Stefan Kölbl, Atul Luykx, and Sophie Schmieg. "How to Abuse and Fix Authenticated Encryption Without Key Commitment". In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 3291–3308. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/albertini (cit. on pp. 13, 82–84, 181).

[Alb+15a] Martin R. Albrecht, Carlos Cid, Jean-Charles Faugère, Robert Fitzpatrick, and Ludovic Perret. "Algebraic Algorithms for LWE Problems". In: *ACM Commun. Comput. Algebra* 49.2 (Aug. 2015), p. 62. DOI: 10.1145/2815111.2815158 (cit. on pp. 83, 178).

[Alb+15b] Martin R. Albrecht, Carlos Cid, Jean-Charles Faugère, Robert Fitzpatrick, and Ludovic Perret. "On the complexity of the BKW algorithm on LWE". In: *Designs, Codes and Cryptography* 74.2 (Feb. 2015), pp. 325–354. DOI: 10.1007/s10623-013-9864-x (cit. on p. 176).

[Alb+20] Martin R. Albrecht, Vlad Gheorghiu, Eamonn W. Postlethwaite, and John M. Schanck. "Estimating Quantum Speedups for Lattice Sieves". In: *ASIACRYPT 2020, Part II*. Ed. by Shiho Moriai and Huaxiong Wang. Vol. 12492. LNCS. Springer, Heidelberg, Dec. 2020, pp. 583–613. DOI: 10.1007/978-3-030-64834-3_20 (cit. on p. 176).

[APS15] Martin R. Albrecht, Rachel Player, and Sam Scott. "On the concrete hardness of Learning with Errors". In: *Journal of Mathematical Cryptology* 9.3 (2015), pp. 169–203. DOI: doi:10.1515/jmc-2015-0016 (cit. on p. 176).

[Alk+20] Erdem Alkim, Hülya Evkan, Norman Lahr, Ruben Niederhagen, and Richard Petri. "ISA Extensions for Finite Field Arithmetic". In: *IACR TCHES* 2020.3 (2020). https://tches.iacr.org/index.php/TCHES/article/view/8589, pp. 219–242. DOI: 10.13154/tches.v2020.i3.219-242 (cit. on p. 173).

[Alw+22a] Joël Alwen, Benedikt Auerbach, Miguel Cueto Noval, Karen Klein, Guillermo Pascual-Perez, and Krzysztof Pietrzak. *DeCAF: Decentralizable Continuous Group Key Agreement with Fast Healing*. Cryptology ePrint Archive, Report 2022/559. https://eprint.iacr.org/2022/559. 2022 (cit. on p. 4).

[Alw+22b] Joël Alwen, Benedikt Auerbach, Miguel Cueto Noval, Karen Klein, Guillermo Pascual-Perez, Krzysztof Pietrzak, and Michael Walter. *CoCoA: Concurrent Continuous Group Key Agreement*. Cryptology ePrint Archive, Report 2022/251. https://eprint.iacr.org/2022/251. 2022 (cit. on p. 301).

[Alw+22c] Joël Alwen, Benedikt Auerbach, Miguel Cueto Noval, Karen Klein, Guillermo Pascual-Perez, Krzysztof Pietrzak, and Michael Walter. "CoCoA: Concurrent Continuous Group Key Agreement". In: *EUROCRYPT 2022, Part II*. Ed. by Orr Dunkelman and Stefan Dziembowski. Vol. 13276. LNCS. Springer, Heidelberg, May 2022, pp. 815–844. DOI: 10.1007/978-3-031-07085-3_28 (cit. on pp. 4, 194, 195, 197, 301).

[ACD19] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. "The Double Ratchet: Security Notions, Proofs, and Modularization for the Signal Protocol". In: *EUROCRYPT 2019, Part I*. Ed. by Yuval Ishai and Vincent Rijmen. Vol. 11476. LNCS. Springer, Heidelberg, May 2019, pp. 129–158. DOI: 10.1007/978-3-030-17653-2_5 (cit. on pp. 2, 5, 79, 194).

[Alw+20a] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. "Security Analysis and Improvements for the IETF MLS Standard for Group Messaging". In: *CRYPTO 2020, Part I*. Ed. by Daniele Micciancio and Thomas Ristenpart. Vol. 12170. LNCS. Springer, Heidelberg, Aug. 2020, pp. 248–277. DOI: 10.1007/978-3-030-56784-2_9 (cit. on pp. 4, 6, 79, 194, 195, 197, 299, 300).

[Alw+21a] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. "Modular Design of Secure Group Messaging Protocols and the Security of MLS". In: *ACM CCS 2021*. Ed. by Giovanni Vigna and Elaine Shi. ACM Press, Nov. 2021, pp. 1463–1483. DOI: 10.1145/3460120.3484820 (cit. on pp. 4, 194, 195, 197, 199, 203, 236, 240, 299, 300).

[Alw+20b] Joël Alwen, Sandro Coretti, Daniel Jost, and Marta Mularczyk. "Continuous Group Key Agreement with Active Security". In: *TCC 2020, Part II*. Ed. by Rafael Pass and Krzysztof Pietrzak. Vol. 12551. LNCS. Springer, Heidelberg, Nov. 2020, pp. 261–290. DOI: 10.1007/978-3-030-64378-2_10 (cit. on pp. 4, 6, 79, 83, 86, 93–95, 99, 107, 108, 123, 125, 132, 186, 194, 195, 197, 198, 200–203).

[Alw+21b] Joël Alwen, Dominik Hartmann, Eike Kiltz, and Marta Mularczyk. *Server-Aided Continuous Group Key Agreement*. Cryptology ePrint Archive, Report 2021/1456. https://eprint.iacr.org/2021/1456. 2021 (cit. on p. 301).

[Alw+22d] Joël Alwen, Dominik Hartmann, Eike Kiltz, and Marta Mularczyk. "Server-Aided Continuous Group Key Agreement". In: *ACM CCS 2022*. Ed. by Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi. ACM Press, Nov. 2022, pp. 69–82. DOI: 10.1145/3548606.3560632 (cit. on pp. 194–198, 201, 203, 236, 301).

[AJM22] Joël Alwen, Daniel Jost, and Marta Mularczyk. "On the Insider Security of MLS". In: *CRYPTO 2022, Part II*. Ed. by Yevgeniy Dodis and Thomas Shrimpton. Vol. 13508. LNCS. Springer, Heidelberg, Aug. 2022, pp. 34–68. DOI: 10.1007/978-3-031-15979-4_2 (cit. on pp. 4, 6, 79, 83, 93, 94, 96, 99, 107–109, 111, 113, 116–121, 123, 125, 126, 132, 186, 194–203, 206, 236, 299, 300).

[ADR02] Jee Hea An, Yevgeniy Dodis, and Tal Rabin. "On the Security of Joint Signature and Encryption". In: *EUROCRYPT 2002*. Ed. by Lars R. Knudsen. Vol. 2332. LNCS. Springer, Heidelberg, Apr. 2002, pp. 83–107. DOI: 10.1007/3-540-46035-7_6 (cit. on p. 14).

[App+09] Benny Applebaum, David Cash, Chris Peikert, and Amit Sahai. "Fast Cryptographic Primitives and Circular-Secure Encryption Based on Hard Learning Problems". In: *CRYPTO 2009*. Ed. by Shai Halevi. Vol. 5677. LNCS. Springer, Heidelberg, Aug. 2009, pp. 595–618. DOI: 10.1007/978-3-642-03356-8_35 (cit. on p. 179).

[AG11] Sanjeev Arora and Rong Ge. "New Algorithms for Learning in Presence of Errors". In: *ICALP 2011, Part I*. Ed. by Luca Aceto, Monika Henzinger, and Jiri Sgall. Vol. 6755. LNCS. Springer, Heidelberg, July 2011, pp. 403–415. DOI: 10.1007/978-3-642-22006-7_34 (cit. on pp. 83, 178).

[Bad+15] Christoph Bader, Dennis Hofheinz, Tibor Jager, Eike Kiltz, and Yong Li. "Tightly-Secure Authenticated Key Exchange". In: *TCC 2015, Part I*. Ed. by Yevgeniy Dodis and Jesper Buus Nielsen. Vol. 9014. LNCS. Springer, Heidelberg, Mar. 2015, pp. 629–658. DOI: 10.1007/978-3-662-46494-6_26 (cit. on pp. 31–33).

[BG14] Shi Bai and Steven D. Galbraith. "Lattice Decoding Attacks on Binary LWE". In: *ACISP 14*. Ed. by Willy Susilo and Yi Mu. Vol. 8544. LNCS. Springer, Heidelberg, July 2014, pp. 322–337. DOI: 10.1007/978-3-319-08344-5_21 (cit. on p. 176).

[BRV20] Fatih Balli, Paul Rösler, and Serge Vaudenay. "Determining the Core Primitive for Optimally Secure Ratcheting". In: *ASIACRYPT 2020, Part III*. Ed. by Shiho Moriai and Huaxiong Wang. Vol. 12493. LNCS. Springer, Heidelberg, Dec. 2020, pp. 621–650. DOI: 10.1007/978-3-030-64840-4_21 (cit. on p. 2).

[BF07] Manuel Barbosa and Pooya Farshim. "Randomness Reuse: Extensions and Improvements". In: *11th IMA International Conference on Cryptography and Coding*. Ed. by Steven D. Galbraith. Vol. 4887. LNCS. Springer, Heidelberg, Dec. 2007, pp. 257–276 (cit. on p. 17).

[Bar18] Richard Barnes. *[MLS] Efficiency and "Ampelmann trees"*. IETF Mail Archive. 2018. URL: https://mailarchive.ietf.org/arch/msg/mls/INcV28Jth25m_l__NMmQIYp13Po/ (cit. on p. 110).

[Bar+20] Richard Barnes, Benjamin Beurdouche, Jon Millican, Emad Omara, Katriel Cohn-Gordon, and Raphael Robert. *The Messaging Layer Security (MLS) Protocol*. Internet-Draft draft-ietf-mls-protocol-11. Work in Progress. Internet Engineering Task Force, Dec. 2020. 88 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-mls-protocol-11 (cit. on pp. 6, 79).

[Bar+22] Richard Barnes, Benjamin Beurdouche, Jon Millican, Emad Omara, Katriel Cohn-Gordon, and Raphael Robert. *The Messaging Layer Security (MLS) Protocol*. Internet-Draft draft-ietf-mls-protocol-13. Work in Progress. Internet Engineering Task Force, 2022. 124 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-mls-protocol-13 (cit. on pp. 193, 194, 221, 222, 233, 299, 300).

[Bel06] Mihir Bellare. "New Proofs for NMAC and HMAC: Security without Collision-Resistance". In: *CRYPTO 2006*. Ed. by Cynthia Dwork. Vol. 4117. LNCS. Springer, Heidelberg, Aug. 2006, pp. 602–619. DOI: 10.1007/11818175_36 (cit. on p. 51).

[Bel15] Mihir Bellare. "New Proofs for NMAC and HMAC: Security without Collision Resistance". In: *Journal of Cryptology* 28.4 (Oct. 2015), pp. 844–878. DOI: 10.1007/s00145-014-9185-x (cit. on p. 51).

[BBS03] Mihir Bellare, Alexandra Boldyreva, and Jessica Staddon. "Randomness Re-use in Multi-recipient Encryption Schemeas". In: *PKC 2003*. Ed. by Yvo Desmedt. Vol. 2567. LNCS. Springer, Heidelberg, Jan. 2003, pp. 85–99. DOI: 10.1007/3-540-36288-6_7 (cit. on p. 82).

[Bel+98] Mihir Bellare, Anand Desai, David Pointcheval, and Phillip Rogaway. "Relations Among Notions of Security for Public-Key Encryption Schemes". In: *CRYPTO'98*. Ed. by Hugo Krawczyk. Vol. 1462. LNCS. Springer, Heidelberg, Aug. 1998, pp. 26–45. DOI: 10.1007/BFb0055718 (cit. on pp. 27, 56, 59).

[BN00] Mihir Bellare and Chanathip Namprempre. "Authenticated Encryption: Relations among notions and analysis of the generic composition paradigm". In: *ASIACRYPT 2000*. Ed. by Tatsuaki Okamoto. Vol. 1976. LNCS. Springer, Heidelberg, Dec. 2000, pp. 531–545. DOI: 10.1007/3-540-44448-3_41 (cit. on p. 12).

[BP04] Mihir Bellare and Adriana Palacio. "Towards Plaintext-Aware Public-Key Encryption without Random Oracles". In: *ASIACRYPT 2004*. Ed. by Pil Joong Lee. Vol. 3329. LNCS. Springer, Heidelberg, Dec. 2004, pp. 48–62. DOI: 10.1007/978-3-540-30539-2_4 (cit. on pp. 11, 27, 56, 59).

[BR94] Mihir Bellare and Phillip Rogaway. "Entity Authentication and Key Distribution". In: *CRYPTO'93*. Ed. by Douglas R. Stinson. Vol. 773. LNCS. Springer, Heidelberg, Aug. 1994, pp. 232–249. DOI: 10.1007/3-540-48329-2_21 (cit. on pp. 24, 31, 32).

[BR95] Mihir Bellare and Phillip Rogaway. "Optimal Asymmetric Encryption". In: *EUROCRYPT'94*. Ed. by Alfredo De Santis. Vol. 950. LNCS. Springer, Heidelberg, May 1995, pp. 92–111. DOI: 10.1007/BFb0053428 (cit. on pp. 11, 27, 56, 59).

[Bel+17] Mihir Bellare, Asha Camper Singh, Joseph Jaeger, Maya Nyayapati, and Igors Stepanovs. "Ratcheted Encryption and Key Exchange: The Security of Messaging". In: *CRYPTO 2017, Part III*. Ed. by Jonathan Katz and Hovav Shacham. Vol. 10403. LNCS. Springer, Heidelberg, Aug. 2017, pp. 619–650. DOI: 10.1007/978-3-319-63697-9_21 (cit. on p. 2).

[BKM06] Adam Bender, Jonathan Katz, and Ruggero Morselli. "Ring Signatures: Stronger Definitions, and Constructions Without Random Oracles". In: *TCC 2006*. Ed. by Shai Halevi and Tal Rabin. Vol. 3876. LNCS. Springer, Heidelberg, Mar. 2006, pp. 60–79. DOI: 10.1007/11681878_4 (cit. on p. 16).

[Ber06] Daniel J. Bernstein. "Curve25519: New Diffie-Hellman Speed Records". In: *PKC 2006*. Ed. by Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin. Vol. 3958. LNCS. Springer, Heidelberg, Apr. 2006, pp. 207–228. DOI: 10.1007/11745853_14 (cit. on p. 22).

[Ber+20] Daniel J. Bernstein, Billy Bob Brumley, Ming-Shing Chen, Chitchanok Chuengsatiansup, Tanja Lange, Adrian Marotzke, Bo-Yuan Peng, Nicola Tuveri, Christine van Vredendaal, and Bo-Yin Yang. *NTRU Prime*. Tech. rep. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions. National Institute of Standards and Technology, 2020 (cit. on pp. 83, 110, 171, 172, 174, 181, 183).

[BL20] Daniel J. Bernstein and Tanja Lange. "McTiny: Fast High-Confidence Post-Quantum Key Erasure for Tiny Network Servers". In: *USENIX Security 2020*. Ed. by Srdjan Capkun and Franziska Roesner. USENIX Association, Aug. 2020, pp. 1731–1748 (cit. on p. 5).

[BKP20] Ward Beullens, Shuichi Katsumata, and Federico Pintore. "Calamari and Falafl: Logarithmic (Linkable) Ring Signatures from Isogenies and Lattices". In: *ASIACRYPT 2020, Part II*. Ed. by Shiho Moriai and Huaxiong Wang. Vol. 12492. LNCS. Springer, Heidelberg, Dec. 2020, pp. 464–492. DOI: 10.1007/978-3-030-64834-3_16 (cit. on pp. 25, 70).

[BKV19] Ward Beullens, Thorsten Kleinjung, and Frederik Vercauteren. "CSI-FiSh: Efficient Isogeny Based Signatures Through Class Group Computations". In: *ASIACRYPT 2019, Part I*. Ed. by Steven D. Galbraith and Shiho Moriai. Vol. 11921. LNCS. Springer, Heidelberg, Dec. 2019, pp. 227–247. DOI: 10.1007/978-3-030-34578-5_9 (cit. on p. 70).

[Beu20] Benjamin Beurdouche. "Formal Verification for High Assurance Security Software in F*". English. PhD thesis. 2020 (cit. on p. 81).

[BBR18] Karthikeyan Bhargavan, Richard Barnes, and Eric Rescorla. *TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups A protocol proposal for Messaging Layer Security (MLS)*. Research Report. Inria Paris, May 2018. URL: https://hal.inria.fr/hal-02425247 (cit. on pp. 4, 6, 79, 194).

[BBN19a] Karthikeyan Bhargavan, Benjamin Beurdouche, and Prasad Naldurg. *Formal Models and Verified Protocols for Group Messaging: Attacks and Proofs for IETF MLS*. Research Report. Inria Paris, Dec. 2019. URL: https://hal.inria.fr/hal-02425229 (cit. on pp. 4, 79, 81, 107).

[BBN19b] Karthikeyan Bhargavan, Benjamin Beurdouche, and Prasad Naldurg. *Formal Models and Verified Protocols for Group Messaging: Attacks and Proofs for IETF MLS*. Research Report. Inria Paris, Dec. 2019. URL: https://hal.inria.fr/hal-02425229 (cit. on p. 194).

[Bie+22a] Alexander Bienstock, Yevgeniy Dodis, Sanjam Garg, Garrison Grogan, Mohammad Hajiabadi, and Paul Rösler. *On the Worst-Case Inefficiency of CGKA*. Cryptology ePrint Archive, Report 2022/1237. https://eprint.iacr.org/2022/1237. 2022 (cit. on p. 4).

[BDR20] Alexander Bienstock, Yevgeniy Dodis, and Paul Rösler. "On the Price of Concurrency in Group Ratcheting Protocols". In: *TCC 2020, Part II*. Ed. by Rafael Pass and Krzysztof Pietrzak. Vol. 12551. LNCS. Springer, Heidelberg, Nov. 2020, pp. 198–228. DOI: 10.1007/978-3-030-64378-2_8 (cit. on p. 4).

[Bie+22b] Alexander Bienstock, Jaiden Fairoze, Sanjam Garg, Pratyay Mukherjee, and Srinivasan Raghuraman. "A More Complete Analysis of the Signal Double Ratchet Algorithm". In: *CRYPTO 2022, Part I*. Ed. by Yevgeniy Dodis and Thomas Shrimpton. Vol. 13507. LNCS. Springer, Heidelberg, Aug. 2022, pp. 784–813. DOI: 10.1007/978-3-031-15802-5_27 (cit. on p. 2).

[BS20a] Nina Bindel and John M. Schanck. "Decryption Failure Is More Likely After Success". In: *Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020*. Ed. by Jintai Ding and Jean-Pierre Tillich. Springer, Heidelberg, 2020, pp. 206–225. DOI: 10.1007/978-3-030-44223-1_12 (cit. on p. 179).

[BJM97] Simon Blake-Wilson, Don Johnson, and Alfred Menezes. "Key Agreement Protocols and Their Security Analysis". In: *6th IMA International Conference on Cryptography and Coding*. Ed. by Michael Darnell. Vol. 1355. LNCS. Springer, Heidelberg, Dec. 1997, pp. 30–45 (cit. on pp. 3, 31).

[BM99] Simon Blake-Wilson and Alfred Menezes. "Unknown Key-Share Attacks on the Station-to-Station (STS) Protocol". In: *PKC'99*. Ed. by Hideki Imai and Yuliang Zheng. Vol. 1560. LNCS. Springer, Heidelberg, Mar. 1999, pp. 154–170. DOI: 10.1007/3-540-49162-7_12 (cit. on p. 30).

[BKW00] Avrim Blum, Adam Kalai, and Hal Wasserman. "Noise-tolerant learning, the parity problem, and the statistical query model". In: *32nd ACM STOC*. ACM Press, May 2000, pp. 435–440. DOI: 10.1145/335305.335355 (cit. on pp. 83, 176).

[BFM88] Manuel Blum, Paul Feldman, and Silvio Micali. "Non-Interactive Zero-Knowledge and Its Applications (Extended Abstract)". In: *20th ACM STOC*. ACM Press, May 1988, pp. 103–112. DOI: 10.1145/62212.62222 (cit. on p. 17).

[BS20b] Xavier Bonnetain and André Schrottenloher. "Quantum Security Analysis of CSIDH". In: *EUROCRYPT 2020, Part II*. Ed. by Anne Canteaut and Yuval Ishai. Vol. 12106. LNCS. Springer, Heidelberg, May 2020, pp. 493–522. DOI: 10.1007/978-3-030-45724-2_17 (cit. on p. 24).

[BK10] Zvika Brakerski and Yael Tauman Kalai. *A Framework for Efficient Signatures, Ring Signatures and Identity Based Encryption in the Standard Model*. Cryptology ePrint Archive, Report 2010/086. https://eprint.iacr.org/2010/086. 2010 (cit. on p. 70).

[Bre+22] Jacqueline Brendel, Rune Fiedler, Felix Günther, Christian Janson, and Douglas Stebila. "Post-quantum Asynchronous Deniable Key Exchange and the Signal Handshake". In: *Public-Key Cryptography – PKC 2022*. Ed. by Goichiro Hanaoka, Junji Shikata, and Yohei Watanabe. Cham: Springer International Publishing, 2022, pp. 3–34. DOI: 10.1007/978-3-030-97131-1_1 (cit. on pp. 26, 27, 59, 70, 73, 75, 76).

[Bre+20] Jacqueline Brendel, Marc Fischlin, Felix Günther, Christian Janson, and Douglas Stebila. "Towards Post-Quantum Security for Signal's X3DH Handshake". In: *SAC 2020*. Ed. by Orr Dunkelman, Michael J. Jacobson Jr., and Colin O'Flynn. Vol. 12804. LNCS. Springer, Heidelberg, Oct. 2020, pp. 404–430. DOI: 10.1007/978-3-030-81652-0_16 (cit. on pp. 3, 5, 24).

[Bre22] Thomas Brewster. *Meet The Secretive Surveillance Wizards Helping The FBI And ICE Wiretap Facebook And Google Users*. Forbes. 2022. URL: https://www.forbes.com/sites/thomasbrewster/2022/02/23/meet-the-secretive-surveillance-wizards-helping-the-fbi-and-ice-wiretap-facebook-and-google-users/ (cit. on p. 2).

[BCK22] Chris Brzuska, Eric Cornelissen, and Konrad Kohbrok. "Security Analysis of the MLS Key Derivation". In: *2022 IEEE Symposium on Security and Privacy (S&P)*. 2022, pp. 2535–2553. DOI: 10.1109/SP46214.2022.9833678 (cit. on pp. 4, 194, 195, 197).

[Brz+18] Chris Brzuska, Antoine Delignat-Lavaud, Cédric Fournet, Konrad Kohbrok, and Markulf Kohlweiss. "State Separation for Code-Based Game-Playing Proofs". In: *ASIACRYPT 2018, Part III*. Ed. by Thomas Peyrin and Steven Galbraith. Vol. 11274. LNCS. Springer, Heidelberg, Dec. 2018, pp. 222–249. DOI: 10.1007/978-3-030-03332-3_9 (cit. on p. 4).

[Bud+20] Alessandro Budroni, Qian Guo, Thomas Johansson, Erik Mårtensson, and Paul Stankovski Wagner. "Making the BKW Algorithm Practical for LWE". In: *INDOCRYPT 2020*. Ed. by Karthikeyan Bhargavan, Elisabeth Oswald, and Manoj Prabhakaran. Vol. 12578. LNCS. Springer, Heidelberg, Dec. 2020, pp. 417–439. DOI: 10.1007/978-3-030-65277-7_19 (cit. on pp. 176, 177).

[Cab21] Cable.co.uk. *Worldwide Mobile Data Pricing 2021 | 1GB Cost in 230 Countries*. 2021. URL: https://www.cable.co.uk/mobiles/worldwide-data-pricing/ (cit. on p. 80).

[CDV21] Andrea Caforio, F. Betül Durak, and Serge Vaudenay. "Beyond Security and Efficiency: On-Demand Ratcheting with Security Awareness". In: *PKC 2021, Part II*. Ed. by Juan Garay. Vol. 12711. LNCS. Springer, Heidelberg, May 2021, pp. 649–677. DOI: 10.1007/978-3-030-75248-4_23 (cit. on p. 2).

[Cam+16] Jan Camenisch, Robert R. Enderlein, Stephan Krenn, Ralf Küsters, and Daniel Rausch. "Universal Composition with Responsive Environments". In: *ASIACRYPT 2016, Part II*. Ed. by Jung Hee Cheon and Tsuyoshi Takagi. Vol. 10032. LNCS. Springer, Heidelberg, Dec. 2016, pp. 807–840. DOI: 10.1007/978-3-662-53890-6_27 (cit. on p. 94).

[Can01] Ran Canetti. "Universally Composable Security: A New Paradigm for Cryptographic Protocols". In: *42nd FOCS*. IEEE Computer Society Press, Oct. 2001, pp. 136–145. DOI: 10.1109/SFCS.2001.959888 (cit. on pp. 6, 94, 201, 202).

[Can+07] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. "Universally Composable Security with Global Setup". In: *TCC 2007*. Ed. by Salil P. Vadhan. Vol. 4392. LNCS. Springer, Heidelberg, Feb. 2007, pp. 61–85. DOI: 10.1007/978-3-540-70936-7_4 (cit. on pp. 94, 201).

[Can+96] Ran Canetti, Uriel Feige, Oded Goldreich, and Moni Naor. "Adaptively Secure Multi-Party Computation". In: *28th ACM STOC*. ACM Press, May 1996, pp. 639–648. DOI: 10.1145/237814.238015 (cit. on p. 235).

[Can+22] Ran Canetti, Palak Jain, Marika Swanberg, and Mayank Varia. "Universally Composable End-to-End Secure Messaging". In: *CRYPTO 2022, Part II*. Ed. by Yevgeniy Dodis and Thomas Shrimpton. Vol. 13508. LNCS. Springer, Heidelberg, Aug. 2022, pp. 3–33. DOI: 10.1007/978-3-031-15979-4_1 (cit. on p. 2).

[CK01] Ran Canetti and Hugo Krawczyk. "Analysis of Key-Exchange Protocols and Their Use for Building Secure Channels". In: *EUROCRYPT 2001*. Ed. by Birgit Pfitzmann. Vol. 2045. LNCS. Springer, Heidelberg, May 2001, pp. 453–474. DOI: 10.1007/3-540-44987-6_28 (cit. on pp. 21, 23, 28, 30–32).

[CK02] Ran Canetti and Hugo Krawczyk. "Security Analysis of IKE's Signature-based Key-Exchange Protocol". In: *CRYPTO 2002*. Ed. by Moti Yung. Vol. 2442. LNCS. https://eprint.iacr.org/2002/120/. Springer, Heidelberg, Aug. 2002, pp. 143–161. DOI: 10.1007/3-540-45708-9_10 (cit. on pp. 24, 31).

[Car15] Bjorn Carey. *Stanford computer scientists show telephone metadata can reveal surprisingly sensitive personal information*. May 2015. URL: https://news.stanford.edu/2016/05/16/stanford-computer-scientists-show-telephone-metadata-can-reveal-surprisingly-sensitive-personal-information/ (cit. on p. 2).

[CKS08] David Cash, Eike Kiltz, and Victor Shoup. "The Twin Diffie-Hellman Problem and Applications". In: *EUROCRYPT 2008*. Ed. by Nigel P. Smart. Vol. 4965. LNCS. Springer, Heidelberg, Apr. 2008, pp. 127–145. DOI: 10.1007/978-3-540-78967-3_8 (cit. on p. 22).

[CD22] Wouter Castryck and Thomas Decru. *An efficient key recovery attack on SIDH (preliminary version)*. Cryptology ePrint Archive, Report 2022/975. https://eprint.iacr.org/2022/975. 2022 (cit. on p. 27).

[CMZ14] Melissa Chase, Sarah Meiklejohn, and Greg Zaverucha. "Algebraic MACs and Keyed-Verification Anonymous Credentials". In: *ACM CCS 2014*. Ed. by Gail-Joon Ahn, Moti Yung, and Ninghui Li. ACM Press, Nov. 2014, pp. 1205–1216. DOI: 10.1145/2660267.2660328 (cit. on p. 194).

[CPZ20] Melissa Chase, Trevor Perrin, and Greg Zaverucha. "The Signal Private Group System and Anonymous Credentials Supporting Efficient Verifiable Encryption". In: *ACM CCS 2020*. Ed. by Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna. ACM Press, Nov. 2020, pp. 1445–1459. DOI: 10.1145/3372297.3417887 (cit. on pp. 4, 194–196).

[Cha82] David Chaum. “Blind Signatures for Untraceable Payments”. In: *CRYPTO’82*. Ed. by David Chaum, Ronald L. Rivest, and Alan T. Sherman. Plenum Press, New York, USA, 1982, pp. 199–203 (cit. on pp. 3, 197, 236, 258).

[Cho+95] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. “Private Information Retrieval”. In: *36th FOCS*. IEEE Computer Society Press, Oct. 1995, pp. 41–50. DOI: 10.1109/SFCS.1995.492461 (cit. on p. 261).

[Coh+17] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. “A Formal Security Analysis of the Signal Messaging Protocol”. In: *IEEE European Symposium on Security and Privacy (EuroS&P)*. 2017, pp. 451–466. DOI: 10.1109/EuroSP.2017.27 (cit. on pp. 2, 3, 5, 79, 194).

[Coh+20] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. “A Formal Security Analysis of the Signal Messaging Protocol”. In: *Journal of Cryptology* 33.4 (Oct. 2020), pp. 1914–1983. DOI: 10.1007/s00145-020-09360-1 (cit. on pp. 2, 3, 5).

[CCG16] Katriel Cohn-Gordon, Cas Cremers, and Luke Garratt. “On Post-compromise Security”. In: *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*. 2016, pp. 164–178. DOI: 10.1109/CSF.2016.19 (cit. on pp. 79, 194).

[Coh+18] Katriel Cohn-Gordon, Cas Cremers, Luke Garratt, Jon Millican, and Kevin Milner. “On Ends-to-Ends Encryption: Asynchronous Group Messaging with Strong Security Guarantees”. In: *ACM CCS 2018*. Ed. by David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang. ACM Press, Oct. 2018, pp. 1802–1819. DOI: 10.1145/3243734.3243747 (cit. on pp. 3, 4, 194).

[Coh+19] Katriel Cohn-Gordon, Cas Cremers, Kristian Gjøsteen, Håkon Jacobsen, and Tibor Jager. “Highly Efficient Key Exchange Protocols with Optimal Tightness”. In: *CRYPTO 2019, Part III*. Ed. by Alexandra Boldyreva and Daniele Micciancio. Vol. 11694. LNCS. Springer, Heidelberg, Aug. 2019, pp. 767–797. DOI: 10.1007/978-3-030-26954-8_25 (cit. on pp. 23, 24, 27, 31–33).

[CS03] Ronald Cramer and Victor Shoup. “Design and Analysis of Practical Public-Key Encryption Schemes Secure against Adaptive Chosen Ciphertext Attack”. In: *SIAM Journal on Computing* 33.1 (2003), pp. 167–226 (cit. on p. 11).

[Cre11] Cas Cremers. “Examining indistinguishability-based security models for key exchange protocols: the case of CK, CK-HMQV, and eCK”. In: *ASIACCS 11*. Ed. by Bruce S. N. Cheung, Lucas Chi Kwong Hui, Ravi S. Sandhu, and Duncan S. Wong. ACM Press, Mar. 2011, pp. 80–91 (cit. on p. 32).

[Cre09] Cas J. F. Cremers. “Session-state Reveal Is Stronger Than Ephemeral Key Reveal: Attacking the NAXOS Authenticated Key Exchange Protocol”. In: *ACNS 09*. Ed. by Michel Abdalla, David Pointcheval, Pierre-Alain Fouque, and Damien Vergnaud. Vol. 5536. LNCS. Springer, Heidelberg, June 2009, pp. 20–33. DOI: 10.1007/978-3-642-01957-9_2 (cit. on p. 32).

[CF12] Cas J. F. Cremers and Michele Feltz. “Beyond eCK: Perfect Forward Secrecy under Actor Compromise and Ephemeral-Key Reveal”. In: *ESORICS 2012*. Ed. by Sara Foresti, Moti Yung, and Fabio Martinelli. Vol. 7459. LNCS. Springer, Heidelberg, Sept. 2012, pp. 734–751. DOI: 10.1007/978-3-642-33167-1_42 (cit. on pp. 31, 32).

[DAn+20] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, Frederik Vercauteren, Jose Maria Bermudo Mera, Michiel Van Beirendonck, and Andrea Basso. *SABER*. Tech. rep. available at `https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions`. National Institute of Standards and Technology, 2020 (cit. on pp. 172, 179).

[DRV20] Jan-Pieter D'Anvers, Mélissa Rossi, and Fernando Virdia. "(One) Failure Is Not an Option: Bootstrapping the Search for Failures in Lattice-Based Encryption Schemes". In: *EUROCRYPT 2020, Part III*. Ed. by Anne Canteaut and Yuval Ishai. Vol. 12107. LNCS. Springer, Heidelberg, May 2020, pp. 3–33. DOI: `10.1007/978-3-030-45727-3_1` (cit. on p. 179).

[DAn+19] Jan-Pieter D'Anvers, Marcel Tiepelt, Frederik Vercauteren, and Ingrid Verbauwhede. "Timing Attacks on Error Correcting Codes in Post-Quantum Schemes". In: *Proceedings of ACM Workshop on Theory of Implementation Security Workshop*. TIS'19. London, United Kingdom: Association for Computing Machinery, 2019, pp. 2–9. DOI: `10.1145/3338467.3358948`. URL: `https://doi.org/10.1145/3338467.3358948` (cit. on p. 173).

[DVV19] Jan-Pieter D'Anvers, Frederik Vercauteren, and Ingrid Verbauwhede. "The Impact of Error Dependencies on Ring/Mod-LWE/LWR Based Schemes". In: *Post-Quantum Cryptography - 10th International Conference, PQCrypto 2019*. Ed. by Jintai Ding and Rainer Steinwandt. Springer, Heidelberg, 2019, pp. 103–115. DOI: `10.1007/978-3-030-25510-7_6` (cit. on p. 173).

[Dac+20] Dana Dachman-Soled, Léo Ducas, Huijing Gong, and Mélissa Rossi. "LWE with Side Information: Attacks and Concrete Security Estimation". In: *CRYPTO 2020, Part II*. Ed. by Daniele Micciancio and Thomas Ristenpart. Vol. 12171. LNCS. Springer, Heidelberg, Aug. 2020, pp. 329–358. DOI: `10.1007/978-3-030-56880-1_12` (cit. on p. 176).

[dFW20] Cyprien de Saint Guilhem, Marc Fischlin, and Bogdan Warinschi. "Authentication in Key-Exchange: Definitions, Relations and Composition". In: *CSF 2020 Computer Security Foundations Symposium*. Ed. by Limin Jia and Ralf Küsters. IEEE Computer Society Press, 2020, pp. 288–303. DOI: `10.1109/CSF49147.2020.00028` (cit. on pp. 28, 32, 33).

[DGK06] Mario Di Raimondo, Rosario Gennaro, and Hugo Krawczyk. "Deniable authentication and key exchange". In: *ACM CCS 2006*. Ed. by Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati. ACM Press, Oct. 2006, pp. 400–409. DOI: `10.1145/1180405.1180454` (cit. on pp. 11, 22, 25, 27, 56–59, 71).

[DVW92] Whitfield Diffie, Paul C. Van Oorschot, and Michael J. Wiener. "Authentication and authenticated key exchanges". In: *Designs, Codes and Cryptography* 2.2 (June 1992), pp. 107–125. DOI: `10.1007/BF00124891` (cit. on p. 30).

[DMS04] Roger Dingledine, Nick Mathewson, and Paul F. Syverson. "Tor: The Second-Generation Onion Router". In: *USENIX Security 2004*. Ed. by Matt Blaze. USENIX Association, Aug. 2004, pp. 303–320 (cit. on pp. 194, 195, 258).

[DG22] Samuel Dobson and Steven D. Galbraith. "Post-Quantum Signal Key Agreement from SIDH". In: *Post-Quantum Cryptography*. Ed. by Jung Hee Cheon and Thomas Johansson. Cham: Springer International Publishing, 2022, pp. 422–450 (cit. on pp. 26, 27).

[Dod+18] Yevgeniy Dodis, Paul Grubbs, Thomas Ristenpart, and Joanne Woodage. "Fast Message Franking: From Invisible Salamanders to Encryptment". In: *CRYPTO 2018, Part I*. Ed. by Hovav Shacham and Alexandra Boldyreva. Vol. 10991. LNCS. Springer, Heidelberg, Aug. 2018, pp. 155–186. DOI: `10.1007/978-3-319-96884-1_6` (cit. on pp. 13, 82–84).

[Dod+09] Yevgeniy Dodis, Jonathan Katz, Adam Smith, and Shabsi Walfish. "Composability and On-Line Deniability of Authentication". In: *TCC 2009*. Ed. by Omer Reingold. Vol. 5444. LNCS. Springer, Heidelberg, Mar. 2009, pp. 146–162. DOI: 10.1007/978-3-642-00457-5_10 (cit. on pp. 27, 58, 59, 71).

[DRS04] Yevgeniy Dodis, Leonid Reyzin, and Adam Smith. "Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data". In: *EUROCRYPT 2004*. Ed. by Christian Cachin and Jan Camenisch. Vol. 3027. LNCS. Springer, Heidelberg, May 2004, pp. 523–540. DOI: 10.1007/978-3-540-24676-3_31 (cit. on p. 15).

[Dow+22] Benjamin Dowling, Eduard Hauck, Doreen Riepel, and Paul Rösler. *Strongly Anonymous Ratcheted Key Exchange*. Cryptology ePrint Archive, Report 2022/1187. https://eprint.iacr.org/2022/1187. 2022 (cit. on p. 2).

[Duc18] Léo Ducas. "Shortest Vector from Lattice Sieving: A Few Dimensions for Free". In: *EUROCRYPT 2018, Part I*. Ed. by Jesper Buus Nielsen and Vincent Rijmen. Vol. 10820. LNCS. Springer, Heidelberg, Apr. 2018, pp. 125–145. DOI: 10.1007/978-3-319-78381-9_5 (cit. on p. 176).

[DV19] F. Betül Durak and Serge Vaudenay. "Bidirectional Asynchronous Ratcheted Key Agreement with Linear Complexity". In: *IWSEC 19*. Ed. by Nuttapong Attrapadung and Takeshi Yagi. Vol. 11689. LNCS. Springer, Heidelberg, Aug. 2019, pp. 343–362. DOI: 10.1007/978-3-030-26834-3_20 (cit. on p. 2).

[ElG85] Taher ElGamal. "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms". In: *IEEE Transactions on Information Theory* 31 (1985), pp. 469–472 (cit. on p. 5).

[Esg+19a] Muhammed F. Esgin, Ron Steinfeld, Joseph K. Liu, and Dongxi Liu. "Lattice-Based Zero-Knowledge Proofs: New Techniques for Shorter and Faster Constructions and Applications". In: *CRYPTO 2019, Part I*. Ed. by Alexandra Boldyreva and Daniele Micciancio. Vol. 11692. LNCS. Springer, Heidelberg, Aug. 2019, pp. 115–146. DOI: 10.1007/978-3-030-26948-7_5 (cit. on p. 70).

[Esg+19b] Muhammed F. Esgin, Ron Steinfeld, Amin Sakzad, Joseph K. Liu, and Dongxi Liu. "Short Lattice-Based One-out-of-Many Proofs and Applications to Ring Signatures". In: *ACNS 19*. Ed. by Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung. Vol. 11464. LNCS. Springer, Heidelberg, June 2019, pp. 67–88. DOI: 10.1007/978-3-030-21568-2_4 (cit. on p. 70).

[Esg+19c] Muhammed F. Esgin, Raymond K. Zhao, Ron Steinfeld, Joseph K. Liu, and Dongxi Liu. "MatRiCT: Efficient, Scalable and Post-Quantum Blockchain Confidential Transactions Protocol". In: *ACM CCS 2019*. Ed. by Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz. ACM Press, Nov. 2019, pp. 567–584. DOI: 10.1145/3319535.3354200 (cit. on p. 70).

[FOR17] Pooya Farshim, Claudio Orlandi, and Răzvan Roşie. "Security of Symmetric Primitives under Incorrect Usage of Keys". In: *IACR Trans. Symm. Cryptol.* 2017.1 (2017), pp. 449–473. DOI: 10.13154/tosc.v2017.i1.449-473 (cit. on pp. 13, 82, 83).

[FLS90] Uriel Feige, Dror Lapidot, and Adi Shamir. "Multiple Non-Interactive Zero Knowledge Proofs Based on a Single Random String (Extended Abstract)". In: *31st FOCS*. IEEE Computer Society Press, Oct. 1990, pp. 308–317. DOI: 10.1109/FSCS.1990.89549 (cit. on p. 17).

[FJP14] Luca De Feo, David Jao, and Jérôme Plû. "Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies". In: *Journal of Mathematical Cryptology* 8.3 (2014), pp. 209–247. DOI: doi:10.1515/jmc-2012-0015 (cit. on p. 174).

[Fis05] Marc Fischlin. "Communication-Efficient Non-interactive Proofs of Knowledge with Online Extractors". In: *CRYPTO 2005*. Ed. by Victor Shoup. Vol. 3621. LNCS. Springer, Heidelberg, Aug. 2005, pp. 152–168. DOI: 10.1007/11535218_10 (cit. on p. 71).

[Fli15] Ola Flisbäck. *Stalking anyone on Telegram*. Dec. 2015. URL: https://oflisback.github.io/telegram-stalking/ (cit. on p. 2).

[FPZ08] Pierre-Alain Fouque, David Pointcheval, and Sébastien Zimmer. "HMAC is a randomness extractor and applications to TLS". In: *ASIACCS 08*. Ed. by Masayuki Abe and Virgil Gligor. ACM Press, Mar. 2008, pp. 21–32 (cit. on p. 51).

[Fre+13] Eduarda S. V. Freire, Dennis Hofheinz, Eike Kiltz, and Kenneth G. Paterson. "Non-Interactive Key Exchange". In: *PKC 2013*. Ed. by Kaoru Kurosawa and Goichiro Hanaoka. Vol. 7778. LNCS. Springer, Heidelberg, Feb. 2013, pp. 254–271. DOI: 10.1007/978-3-642-36362-7_17 (cit. on p. 22).

[Fuj+12] Atsushi Fujioka, Koutarou Suzuki, Keita Xagawa, and Kazuki Yoneyama. "Strongly Secure Authenticated Key Exchange from Factoring, Codes, and Lattices". In: *PKC 2012*. Ed. by Marc Fischlin, Johannes Buchmann, and Mark Manulis. Vol. 7293. LNCS. Springer, Heidelberg, May 2012, pp. 467–484. DOI: 10.1007/978-3-642-30057-8_28 (cit. on pp. 21, 23–25, 31, 32).

[Fuj+13] Atsushi Fujioka, Koutarou Suzuki, Keita Xagawa, and Kazuki Yoneyama. "Practical and post-quantum authenticated key exchange from one-way secure key encapsulation mechanism". In: *ASIACCS 13*. Ed. by Kefei Chen, Qi Xie, Weidong Qiu, Ninghui Li, and Wen-Guey Tzeng. ACM Press, May 2013, pp. 83–94 (cit. on p. 24).

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. "Secure Integration of Asymmetric and Symmetric Encryption Schemes". In: *CRYPTO'99*. Ed. by Michael J. Wiener. Vol. 1666. LNCS. Springer, Heidelberg, Aug. 1999, pp. 537–554. DOI: 10.1007/3-540-48405-1_34 (cit. on p. 18).

[GJ18] Kristian Gjøsteen and Tibor Jager. "Practical and Tightly-Secure Digital Signatures and Authenticated Key Exchange". In: *CRYPTO 2018, Part II*. Ed. by Hovav Shacham and Alexandra Boldyreva. Vol. 10992. LNCS. Springer, Heidelberg, Aug. 2018, pp. 95–125. DOI: 10.1007/978-3-319-96881-0_4 (cit. on pp. 23, 24, 27, 31–33).

[GGM84] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. "How to Construct Random Functions (Extended Abstract)". In: *25th FOCS*. IEEE Computer Society Press, Oct. 1984, pp. 464–479. DOI: 10.1109/SFCS.1984.715949 (cit. on p. 15).

[GMR88] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. "A Digital Signature Scheme Secure Against Adaptive Chosen-message Attacks". In: *SIAM Journal on Computing* 17.2 (Apr. 1988), pp. 281–308 (cit. on p. 14).

[GM13] Glenn Greenwald and Ewen MacAskill. *NSA Prism program taps in to user data of Apple, Google and others*. The Guardian. 2013. URL: https://www.theguardian.com/world/2013/jun/06/us-tech-giants-nsa-data (cit. on p. 1).

[GLR17] Paul Grubbs, Jiahui Lu, and Thomas Ristenpart. "Message Franking via Committing Authenticated Encryption". In: *CRYPTO 2017, Part III*. Ed. by Jonathan Katz and Hovav Shacham. Vol. 10403. LNCS. Springer, Heidelberg, Aug. 2017, pp. 66–97. DOI: 10.1007/978-3-319-63697-9_3 (cit. on pp. 13, 82, 83).

[Gua] Guardian Project. *Orbot: Proxy with Tor*. URL: https://guardianproject.info/apps/org.torproject.android/ (visited on 04/15/2022) (cit. on pp. 194, 195, 258).

[Guo+17] Qian Guo, Thomas Johansson, Erik Mårtensson, and Paul Stankovski. "Coded-BKW with Sieving". In: *ASIACRYPT 2017, Part I*. Ed. by Tsuyoshi Takagi and Thomas Peyrin. Vol. 10624. LNCS. Springer, Heidelberg, Dec. 2017, pp. 323–346. DOI: 10.1007/978-3-319-70694-8_12 (cit. on p. 177).

[Guo+19] Qian Guo, Thomas Johansson, Erik Mårtensson, and Paul Stankovski Wagner. "On the Asymptotics of Solving the LWE Problem Using Coded-BKW With Sieving". In: *IEEE Transactions on Information Theory* 65.8 (2019), pp. 5243–5259. DOI: 10.1109/TIT.2019.2906233 (cit. on p. 177).

[GJS15] Qian Guo, Thomas Johansson, and Paul Stankovski. "Coded-BKW: Solving LWE Using Lattice Codes". In: *CRYPTO 2015, Part I*. Ed. by Rosario Gennaro and Matthew J. B. Robshaw. Vol. 9215. LNCS. Springer, Heidelberg, Aug. 2015, pp. 23–42. DOI: 10.1007/978-3-662-47989-6_2 (cit. on p. 177).

[GJS16] Qian Guo, Thomas Johansson, and Paul Stankovski. "A Key Recovery Attack on MDPC with CCA Security Using Decoding Errors". In: *ASIACRYPT 2016, Part I*. Ed. by Jung Hee Cheon and Tsuyoshi Takagi. Vol. 10031. LNCS. Springer, Heidelberg, Dec. 2016, pp. 789–815. DOI: 10.1007/978-3-662-53887-6_29 (cit. on p. 179).

[GJY19] Qian Guo, Thomas Johansson, and Jing Yang. "A Novel CCA Attack Using Decryption Errors Against LAC". In: *ASIACRYPT 2019, Part I*. Ed. by Steven D. Galbraith and Shiho Moriai. Vol. 11921. LNCS. Springer, Heidelberg, Dec. 2019, pp. 82–111. DOI: 10.1007/978-3-030-34578-5_4 (cit. on p. 173).

[GMW21] Qian Guo, Erik Mårtensson, and Paul Stankovski Wagner. "On the Sample Complexity of solving LWE using BKW-Style Algorithms". In: *2021 IEEE International Symposium on Information Theory (ISIT)*. 2021, pp. 2405–2410. DOI: 10.1109/ISIT45174.2021.9518190 (cit. on pp. 176, 177).

[Guo+20] Siyao Guo, Pritish Kamath, Alon Rosen, and Katerina Sotiraki. "Limits on the Efficiency of (Ring) LWE Based Non-interactive Key Exchange". In: *PKC 2020, Part I*. Ed. by Aggelos Kiayias, Markulf Kohlweiss, Petros Wallden, and Vassilis Zikas. Vol. 12110. LNCS. Springer, Heidelberg, May 2020, pp. 374–395. DOI: 10.1007/978-3-030-45374-9_13 (cit. on p. 3).

[HGS99] Chris Hall, Ian Goldberg, and Bruce Schneier. "Reaction Attacks against several Public-Key Cryptosystems". In: *ICICS 99*. Ed. by Vijay Varadharajan and Yi Mu. Vol. 1726. LNCS. Springer, Heidelberg, Nov. 1999, pp. 2–12 (cit. on p. 179).

[Has+21a] Keitaro Hashimoto, Shuichi Katsumata, Kris Kwiatkowski, and Thomas Prest. "An Efficient and Generic Construction for Signal's Handshake (X3DH): Post-Quantum, State Leakage Secure, and Deniable". In: *PKC 2021, Part II*. Ed. by Juan Garay. Vol. 12711. LNCS. Springer, Heidelberg, May 2021, pp. 410–440. DOI: 10.1007/978-3-030-75248-4_15 (cit. on pp. 21, 26, 59, 69).

[Has+22] Keitaro Hashimoto, Shuichi Katsumata, Kris Kwiatkowski, and Thomas Prest. "An Efficient and Generic Construction for Signal's Handshake (X3DH): Post-quantum, State Leakage Secure, and Deniable". In: *Journal of Cryptology* 35.3 (May 2022). DOI: 10.1007/s00145-022-09427-1 (cit. on p. 21).

[Has+21b] Keitaro Hashimoto, Shuichi Katsumata, Eamonn Postlethwaite, Thomas Prest, and Bas Westerbaan. "A Concrete Treatment of Efficient Continuous Group Key Agreement via Multi-Recipient PKEs". In: *ACM CCS 2021*. Ed. by Giovanni Vigna and Elaine Shi. ACM Press, Nov. 2021, pp. 1441–1462. DOI: 10.1145/3460120.3484817 (cit. on pp. 6, 79, 194–203, 207, 209, 210, 221, 235, 236, 240–248, 297, 299, 300).

[Has+21c] Keitaro Hashimoto, Shuichi Katsumata, Eamonn Postlethwaite, Thomas Prest, and Bas Westerbaan. *A Concrete Treatment of Efficient Continuous Group Key Agreement via Multi-Recipient PKEs*. Cryptology ePrint Archive, Report 2021/1407. https://eprint.iacr.org/2021/1407. 2021 (cit. on pp. 79, 202, 203, 206, 207, 209, 210, 216, 221–227, 231, 233, 234, 236, 238, 253).

[HKP22a] Keitaro Hashimoto, Shuichi Katsumata, and Thomas Prest. "How to Hide MetaData in MLS-Like Secure Group Messaging: Simple, Modular, and Post-Quantum". In: *ACM CCS 2022*. Ed. by Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi. ACM Press, Nov. 2022, pp. 1399–1412. DOI: 10.1145/3548606.3560679 (cit. on p. 193).

[HKP22b] Keitaro Hashimoto, Shuichi Katsumata, and Thomas Prest. *How to Hide MetaData in MLS-Like Secure Group Messaging: Simple, Modular, and Post-Quantum*. Cryptology ePrint Archive, Paper 2022/1533. 2022. DOI: 10.1145/3548606.3560679. URL: https://eprint.iacr.org/2022/1533 (cit. on p. 193).

[HKM18] Gottfried Herold, Elena Kirshanova, and Alexander May. "On the asymptotic complexity of solving LWE". In: *Designs, Codes and Cryptography* 86.1 (Jan. 2018), pp. 55–83. DOI: 10.1007/s10623-016-0326-0 (cit. on p. 177).

[Höv+20] Kathrin Hövelmanns, Eike Kiltz, Sven Schäge, and Dominique Unruh. "Generic Authenticated Key Exchange in the Quantum Random Oracle Model". In: *PKC 2020, Part II*. Ed. by Aggelos Kiayias, Markulf Kohlweiss, Petros Wallden, and Vassilis Zikas. Vol. 12111. LNCS. Springer, Heidelberg, May 2020, pp. 389–422. DOI: 10.1007/978-3-030-45388-6_14 (cit. on pp. 24, 31, 32).

[How+20] James Howe, Thomas Prest, Thomas Ricosset, and Mélissa Rossi. "Isochronous Gaussian Sampling: From Inception to Implementation". In: *Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020*. Ed. by Jintai Ding and Jean-Pierre Tillich. Springer, Heidelberg, 2020, pp. 53–71. DOI: 10.1007/978-3-030-44223-1_5 (cit. on p. 180).

[Hul+20] Andreas Hulsing, Daniel J. Bernstein, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Panos Kampanakis, Stefan Kolbl, Tanja Lange, Martin M Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, Peter Schwabe, Jean-Philippe Aumasson, Bas Westerbaan, and Ward Beullens. *SPHINCS+*. Tech. rep. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions. National Institute of Standards and Technology, 2020 (cit. on p. 298).

[Hül+21] Andreas Hülsing, Kai-Chun Ning, Peter Schwabe, Florian Weber, and Philip R. Zimmermann. "Post-quantum WireGuard". In: *2021 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2021, pp. 304–321. DOI: 10.1109/SP40001.2021.00030 (cit. on p. 5).

[Jag+21] Tibor Jager, Eike Kiltz, Doreen Riepel, and Sven Schäge. "Tightly-Secure Authenticated Key Exchange, Revisited". In: *EUROCRYPT 2021, Part I*. Ed. by Anne Canteaut and François-Xavier Standaert. Vol. 12696. LNCS. Springer, Heidelberg, Oct. 2021, pp. 117–146. DOI: 10.1007/978-3-030-77870-5_5 (cit. on pp. 31–33).

[Jao+20] David Jao, Reza Azarderakhsh, Matthew Campagna, Craig Costello, Luca De Feo, Basil Hess, Amir Jalali, Brian Koziel, Brian LaMacchia, Patrick Longa, Michael Naehrig, Joost Renes, Vladimir Soukharev, David Urbanik, Geovandro Pereira, Koray Karabina, and Aaron Hutchinson. *SIKE*. Tech. rep. available at `https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions`. National Institute of Standards and Technology, 2020 (cit. on pp. 171, 183).

[Jaq+20] Samuel Jaques, Michael Naehrig, Martin Roetteler, and Fernando Virdia. "Implementing Grover Oracles for Quantum Key Search on AES and LowMC". In: *EUROCRYPT 2020, Part II*. Ed. by Anne Canteaut and Yuval Ishai. Vol. 12106. LNCS. Springer, Heidelberg, May 2020, pp. 280–310. DOI: 10.1007/978-3-030-45724-2_10 (cit. on pp. 178, 179).

[JMM19a] Daniel Jost, Ueli Maurer, and Marta Mularczyk. "A Unified and Composable Take on Ratcheting". In: *TCC 2019, Part II*. Ed. by Dennis Hofheinz and Alon Rosen. Vol. 11892. LNCS. Springer, Heidelberg, Dec. 2019, pp. 180–210. DOI: 10.1007/978-3-030-36033-7_7 (cit. on pp. 2, 202).

[JMM19b] Daniel Jost, Ueli Maurer, and Marta Mularczyk. "Efficient Ratcheting: Almost-Optimal Guarantees for Secure Messaging". In: *EUROCRYPT 2019, Part I*. Ed. by Yuval Ishai and Vincent Rijmen. Vol. 11476. LNCS. Springer, Heidelberg, May 2019, pp. 159–188. DOI: 10.1007/978-3-030-17653-2_6 (cit. on p. 2).

[Kan87] Ravi Kannan. "Minkowski's Convex Body Theorem and Integer Programming". In: *Mathematics of Operations Research* 12.3 (1987), pp. 415–440. URL: `http://www.jstor.org/stable/3689974` (visited on 09/14/2022) (cit. on p. 176).

[Kat+20] Shuichi Katsumata, Kris Kwiatkowski, Federico Pintore, and Thomas Prest. "Scalable Ciphertext Compression Techniques for Post-quantum KEMs and Their Applications". In: *ASIACRYPT 2020, Part I*. Ed. by Shiho Moriai and Huaxiong Wang. Vol. 12491. LNCS. Springer, Heidelberg, Dec. 2020, pp. 289–320. DOI: 10.1007/978-3-030-64837-4_10 (cit. on pp. 17–19, 82, 83, 110, 171, 172, 174, 179, 180, 183, 297).

[KW03] Jonathan Katz and Nan Wang. "Efficiency Improvements for Signature Schemes with Tight Security Reductions". In: *ACM CCS 2003*. Ed. by Sushil Jajodia, Vijayalakshmi Atluri, and Trent Jaeger. ACM Press, Oct. 2003, pp. 155–164. DOI: 10.1145/948109.948132 (cit. on p. 91).

[Kaw+20] Tomoki Kawashima, Katsuyuki Takashima, Yusuke Aikawa, and Tsuyoshi Takagi. "An Efficient Authenticated Key Exchange from Random Self-reducibility on CSIDH". In: *ICISC 20*. Ed. by Deukjo Hong. Vol. 12593. LNCS. Springer, Heidelberg, Dec. 2020, pp. 58–84. DOI: 10.1007/978-3-030-68890-5_4 (cit. on pp. 24, 31).

[Kle+21] Karen Klein, Guillermo Pascual-Perez, Michael Walter, Chethan Kamath, Margarita Capretto, Miguel Cueto, Ilia Markov, Michelle Yeo, Joël Alwen, and Krzysztof Pietrzak. "Keep the Dirt: Tainted TreeKEM, Adaptively and Actively Secure Continuous Group Key Agreement". In: *2021 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2021, pp. 268–284. DOI: 10.1109/SP40001.2021.00035 (cit. on pp. 4, 79, 80, 125, 186, 194, 195, 197, 299, 300).

[KGV20] Bor de Kock, Kristian Gjøsteen, and Mattia Veroni. "Practical Isogeny-Based Key-Exchange with Optimal Tightness". In: *Selected Areas in Cryptography*. Ed. by Orr Dunkelman, Michael J. Jacobson Jr., and Colin O'Flynn. Cham: Springer International Publishing, 2020, pp. 451–479 (cit. on pp. 24, 31).

[Kra05] Hugo Krawczyk. "HMQV: A High-Performance Secure Diffie-Hellman Protocol". In: *CRYPTO 2005*. Ed. by Victor Shoup. Vol. 3621. LNCS. Springer, Heidelberg, Aug. 2005, pp. 546–566. DOI: 10.1007/11535218_33 (cit. on pp. 21, 23, 31, 32).

[Kra10] Hugo Krawczyk. "Cryptographic Extraction and Key Derivation: The HKDF Scheme". In: *CRYPTO 2010*. Ed. by Tal Rabin. Vol. 6223. LNCS. Springer, Heidelberg, Aug. 2010, pp. 631–648. DOI: 10.1007/978-3-642-14623-7_34 (cit. on p. 20).

[Kro21] Andy Kroll. *FBI Document Says the Feds Can Get Your WhatsApp Data - in Real Time*. Rolling Stone. 2021. URL: https://www.rollingstone.com/politics/politics-features/whatsapp-imessage-facebook-apple-fbi-privacy-1261816/ (cit. on p. 2).

[Kur02] Kaoru Kurosawa. "Multi-recipient Public-Key Encryption with Shortened Ciphertext". In: *PKC 2002*. Ed. by David Naccache and Pascal Paillier. Vol. 2274. LNCS. Springer, Heidelberg, Feb. 2002, pp. 48–63. DOI: 10.1007/3-540-45664-3_4 (cit. on pp. 17, 82, 297).

[KF14] Kaoru Kurosawa and Jun Furukawa. "2-Pass Key Exchange Protocols from CPA-Secure KEM". In: *CT-RSA 2014*. Ed. by Josh Benaloh. Vol. 8366. LNCS. Springer, Heidelberg, Feb. 2014, pp. 385–401. DOI: 10.1007/978-3-319-04852-9_20 (cit. on pp. 21, 24, 26).

[Kwi20] Kris Kwiatkowski. *An Efficient and Generic Construction for Signal's Handshake (X3DH): Post-Quantum, State Leakage Secure, and Deniable. Proof of concept implementation*. Dec. 2020. URL: https://github.com/post-quantum-cryptography/post-quantum-state-leakage-secure-ake (cit. on pp. 22, 50, 52, 55).

[LLM07] Brian A. LaMacchia, Kristin Lauter, and Anton Mityagin. "Stronger Security of Authenticated Key Exchange". In: *ProvSec 2007*. Ed. by Willy Susilo, Joseph K. Liu, and Yi Mu. Vol. 4784. LNCS. Springer, Heidelberg, Nov. 2007, pp. 1–16 (cit. on pp. 21, 23–25, 31, 32).

[Lau+21] Ben Laurie, Adam Langley, Emilia Kasper, Eran Messeri, and Rob Stradling. *Certificate Transparency Version 2.0*. Internet-Draft draft-ietf-trans-rfc6962-bis-35. Work in Progress. Internet Engineering Task Force, Mar. 2021. 60 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-trans-rfc6962-bis-35 (cit. on p. 82).

[LS17] Yong Li and Sven Schäge. "No-Match Attacks and Robust Partnering Definitions: Defining Trivial Attacks for Security Protocols is Not Trivial". In: *ACM CCS 2017*. Ed. by Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu. ACM Press, Oct. 2017, pp. 1343–1360. DOI: 10.1145/3133956.3134006 (cit. on pp. 30, 32).

[LP11] Richard Lindner and Chris Peikert. "Better Key Sizes (and Attacks) for LWE-Based Encryption". In: *CT-RSA 2011*. Ed. by Aggelos Kiayias. Vol. 6558. LNCS. Springer, Heidelberg, Feb. 2011, pp. 319–339. DOI: 10.1007/978-3-642-19074-2_21 (cit. on p. 171).

[LAZ19] Xingye Lu, Man Ho Au, and Zhenfei Zhang. "Raptor: A Practical Lattice-Based (Linkable) Ring Signature". In: *ACNS 19*. Ed. by Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung. Vol. 11464. LNCS. Springer, Heidelberg, June 2019, pp. 110–130. DOI: 10.1007/978-3-030-21568-2_6 (cit. on pp. 25, 70).

[LR88] Michael Luby and Charles Rackoff. "How to construct pseudorandom permutations from pseudorandom functions". In: *SIAM Journal on Computing* 17.2 (1988) (cit. on p. 15).

[Lyu+20] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. *CRYSTALS-DILITHIUM*. Tech. rep. available at `https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions`. National Institute of Standards and Technology, 2020 (cit. on pp. 70, 181, 297).

[MG19] Alexandra Ma and Ben Gilbert. *Facebook understood how dangerous the Trump-linked data firm Cambridge Analytica could be much earlier than it previously said. Here's everything that's happened up until now*. INSIDER. [Online; accessed 8-January-2023]. 2019. URL: `https://www.businessinsider.com/cambridge-analytica-a-guide-to-the-trump-linked-data-firm-that-harvested-50-million-facebook-profiles-2018-3` (cit. on p. 1).

[MP16a] Moxie Marlinspike and Trevor Perrin. *The double ratchet algorithm*. Nov. 2016. URL: `https://signal.org/docs/specifications/doubleratchet/` (cit. on pp. 2, 193).

[MP16b] Moxie Marlinspike and Trevor Perrin. *The X3DH key agreement protocol*. Nov. 2016. URL: `https://signal.org/docs/specifications/x3dh/` (cit. on pp. 2, 3, 5, 22, 24, 31, 48, 52).

[Mar+21] Ian Martiny, Gabriel Kaptchuk, Adam J. Aviv, Daniel S. Roche, and Eric Wustrow. "Improving Signal's Sealed Sender". In: *NDSS 2021*. The Internet Society, Feb. 2021 (cit. on pp. 3, 194, 197).

[McG+15] Susan E. McGregor, Polina Charters, Tobin Holliday, and Franziska Roesner. "Investigating the Computer Security Practices and Needs of Journalists". In: *USENIX Security 2015*. Ed. by Jaeyeon Jung and Thorsten Holz. USENIX Association, Aug. 2015, pp. 399–414 (cit. on p. 2).

[MRC16] Susan E. McGregor, Franziska Roesner, and Kelly Caine. "Individual versus Organizational Computer Security and Privacy Concerns in Journalism". In: *PoPETs* 2016.4 (Oct. 2016), pp. 418–435. DOI: `10.1515/popets-2016-0048` (cit. on p. 2).

[Moh21] Vaishnavi Krishna Mohan. *WhatsApp's New Privacy Policy: Collecting Metadata and Its Implications*. Jan. 2021. URL: `https://www.globalviews360.com/articles/whatsapps-new-privacy-policy-collecting-metadata-and-its-implications` (cit. on p. 2).

[MRS18] Ben Morris, Phillip Rogaway, and Till Stegers. "Deterministic Encryption with the Thorp Shuffle". In: *Journal of Cryptology* 31.2 (Apr. 2018), pp. 521–536. DOI: `10.1007/s00145-017-9262-z` (cit. on p. 298).

[MSs12] Steven Myers, Mona Sergi, and abhi shelat. "Blackbox Construction of a More Than Non-Malleable CCA1 Encryption Scheme from Plaintext Awareness". In: *SCN 12*. Ed. by Ivan Visconti and Roberto De Prisco. Vol. 7485. LNCS. Springer, Heidelberg, Sept. 2012, pp. 149–165. DOI: `10.1007/978-3-642-32928-9_9` (cit. on pp. 11, 59).

[Nae+20] Michael Naehrig, Erdem Alkim, Joppe Bos, Léo Ducas, Karen Easterbrook, Brian LaMacchia, Patrick Longa, Ilya Mironov, Valeria Nikolaenko, Christopher Peikert, Ananth Raghunathan, and Douglas Stebila. *FrodoKEM*. Tech. rep. available at `https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions`. National Institute of Standards and Technology, 2020 (cit. on pp. 83, 110, 171, 172, 174, 176, 178, 180, 181, 183).

[NZ96] Noam Nisan and David Zuckerman. "Randomness is Linear in Space". In: *Journal of Computer and System Sciences* 52.1 (1996), pp. 43–52. DOI: `https://doi.org/10.1006/jcss.1996.0004` (cit. on p. 15).

[NIS17] NIST. *Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process*. Accessed: 2021-04-16. 2017. URL: https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf (cit. on pp. 178, 179).

[Oma+21] Emad Omara, Benjamin Beurdouche, Eric Rescorla, Srinivas Inguva, Albert Kwon, and Alan Duric. *The Messaging Layer Security (MLS) Architecture*. Internet-Draft draft-ietf-mls-architecture-06. Work in Progress. Internet Engineering Task Force, Mar. 2021. 33 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-mls-architecture-06 (cit. on pp. 6, 79, 175).

[Ops13] Kurt Opsahl. *Why Metadata Matters*. June 2013. URL: https://www.eff.org/deeplinks/2013/06/why-metadata-matters (cit. on p. 2).

[PST20] Christian Paquin, Douglas Stebila, and Goutam Tamvada. "Benchmarking Post-quantum Cryptography in TLS". In: *Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020*. Ed. by Jintai Ding and Jean-Pierre Tillich. Springer, Heidelberg, 2020, pp. 72–91. DOI: 10.1007/978-3-030-44223-1_5 (cit. on p. 53).

[Pas03] Rafael Pass. "On Deniability in the Common Reference String and Random Oracle Model". In: *CRYPTO 2003*. Ed. by Dan Boneh. Vol. 2729. LNCS. Springer, Heidelberg, Aug. 2003, pp. 316–337. DOI: 10.1007/978-3-540-45146-4_19 (cit. on p. 58).

[Pei20] Chris Peikert. "He Gives C-Sieves on the CSIDH". In: *EUROCRYPT 2020, Part II*. Ed. by Anne Canteaut and Yuval Ishai. Vol. 12106. LNCS. Springer, Heidelberg, May 2020, pp. 463–492. DOI: 10.1007/978-3-030-45724-2_16 (cit. on p. 24).

[Per16] Trevor Perrin. *The XEdDSA and VXEdDSA Signature Schemes*. Oct. 2016. URL: https://signal.org/docs/specifications/xeddsa/ (cit. on p. 22).

[Per18] Trevor Perrin. *The Noise Protocol Framework*. The Noise Protocol Framework. July 2018. URL: http://www.noiseprotocol.org/noise.pdf (cit. on p. 193).

[PR18] Bertram Poettering and Paul Rösler. "Towards Bidirectional Ratcheted Key Exchange". In: *CRYPTO 2018, Part I*. Ed. by Hovav Shacham and Alexandra Boldyreva. Vol. 10991. LNCS. Springer, Heidelberg, Aug. 2018, pp. 3–32. DOI: 10.1007/978-3-319-96884-1_1 (cit. on p. 2).

[PS14] David Pointcheval and Olivier Sanders. "Forward Secure Non-Interactive Key Exchange". In: *SCN 14*. Ed. by Michel Abdalla and Roberto De Prisco. Vol. 8642. LNCS. Springer, Heidelberg, Sept. 2014, pp. 21–39. DOI: 10.1007/978-3-319-10879-7_2 (cit. on p. 22).

[Pre+20] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. *FALCON*. Tech. rep. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions. National Institute of Standards and Technology, 2020 (cit. on pp. 70, 297).

[Pre+22] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. *FALCON*. Tech. rep. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022. National Institute of Standards and Technology, 2022 (cit. on p. 5).

[Sav13] Charlie Savage. *Court Rejects Appeal Bid by Writer in Leak Case*. The New York Times. Oct. 2013. URL: http://www.nytimes.%20com/2013/10/16/us/court-rejects-appealbid-by-writer-in-leak-case.html (cit. on p. 2).

[Sch90] Claus-Peter Schnorr. "Efficient Identification and Signatures for Smart Cards (Abstract) (Rump Session)". In: *EUROCRYPT'89*. Ed. by Jean-Jacques Quisquater and Joos Vandewalle. Vol. 434. LNCS. Springer, Heidelberg, Apr. 1990, pp. 688–689. DOI: 10.1007/3-540-46885-4_68 (cit. on p. 5).

[Sch+20] Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, and Damien Stehlé. *CRYSTALS-KYBER*. Tech. rep. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions. National Institute of Standards and Technology, 2020 (cit. on pp. 83, 110, 171–174, 176, 179, 183).

[Sch+22] Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, Damien Stehlé, and Jintai Ding. *CRYSTALS-KYBER*. Tech. rep. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022. National Institute of Standards and Technology, 2022 (cit. on p. 5).

[SSW20] Peter Schwabe, Douglas Stebila, and Thom Wiggers. "Post-Quantum TLS Without Handshake Signatures". In: *ACM CCS 2020*. Ed. by Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna. ACM Press, Nov. 2020, pp. 1461–1480. DOI: 10.1145/3372297.3423350 (cit. on p. 5).

[Sho94] Peter W. Shor. "Polynominal time algorithms for discrete logarithms and factoring on a quantum computer". In: *Algorithmic Number Theory, First International Symposium, ANTS-I, Ithaca, NY, USA, May 6-9, 1994, Proceedings*. Ed. by Leonard M. Adleman and Ming-Deh A. Huang. Vol. 877. Lecture Notes in Computer Science. Springer, 1994, p. 289. DOI: 10.1007/3-540-58691-1_68. URL: https://doi.org/10.1007/3-540-58691-1%5C_68 (cit. on p. 4).

[Sho97] Victor Shoup. "Lower Bounds for Discrete Logarithms and Related Problems". In: *EUROCRYPT'97*. Ed. by Walter Fumy. Vol. 1233. LNCS. Springer, Heidelberg, May 1997, pp. 256–266. DOI: 10.1007/3-540-69053-0_18 (cit. on p. 195).

[Sho00] Victor Shoup. "Using Hash Functions as a Hedge against Chosen Ciphertext Attack". In: *EUROCRYPT 2000*. Ed. by Bart Preneel. Vol. 1807. LNCS. Springer, Heidelberg, May 2000, pp. 275–288. DOI: 10.1007/3-540-45539-6_19 (cit. on p. 11).

[Sig] Signal. *Signal protocol: Technical Documentation*. URL: https://signal.org/docs/ (cit. on pp. 2, 31).

[Sig18] Signal Blog. *Technology preview: Sealed sender for Signal*. Oct. 2018. URL: https://signal.org/blog/sealed-sender/ (cit. on pp. 3, 194).

[Sig19] Signal Blog. *Technology Preview: Signal Private Group System*. Dec. 2019. URL: https://signal.org/blog/signal-private-group-system/ (cit. on pp. 4, 194–196).

[Sma05] Nigel P. Smart. "Efficient Key Encapsulation to Multiple Parties". In: *SCN 04*. Ed. by Carlo Blundo and Stelvio Cimato. Vol. 3352. LNCS. Springer, Heidelberg, Sept. 2005, pp. 208–219. DOI: 10.1007/978-3-540-30598-9_15 (cit. on p. 17).

[Spe21] Speedtest. *Speedtest Global Index — Internet Speed around the world*. July 2021. URL: https://www.speedtest.net/global-index (cit. on p. 80).

[Sta20] Katherine E. Stange. *Algebraic aspects of solving Ring-LWE, including ring-based improvements in the Blum-Kalai-Wasserman algorithm*. 2020. arXiv: 1902.07140 [cs.CR] (cit. on p. 178).

[Tya+21] Nirvan Tyagi, Julia Len, Ian Miers, and Thomas Ristenpart. *Orca: Blocklisting in Sender-Anonymous Messaging*. Cryptology ePrint Archive, Report 2021/1380. https://eprint.iacr.org/2021/1380. 2021 (cit. on p. 3).

[UG15] Nik Unger and Ian Goldberg. "Deniable Key Exchanges for Secure Messaging". In: *ACM CCS 2015*. Ed. by Indrajit Ray, Ninghui Li, and Christopher Kruegel. ACM Press, Oct. 2015, pp. 1211–1223. DOI: 10.1145/2810103.2813616 (cit. on pp. 3, 27, 58, 59, 71).

[UG18] Nik Unger and Ian Goldberg. "Improved Strongly Deniable Authenticated Key Exchanges for Secure Messaging". In: *PoPETs* 2018.1 (Jan. 2018), pp. 21–66. DOI: 10.1515/popets-2018-0003 (cit. on pp. 3, 27, 58, 59, 71).

[Vat+20] Nihal Vatandas, Rosario Gennaro, Bertrand Ithurburn, and Hugo Krawczyk. "On the Cryptographic Deniability of the Signal Protocol". In: *ACNS 20, Part II*. Ed. by Mauro Conti, Jianying Zhou, Emiliano Casalicchio, and Angelo Spognardi. Vol. 12147. LNCS. Springer, Heidelberg, Oct. 2020, pp. 188–209. DOI: 10.1007/978-3-030-57878-7_10 (cit. on pp. 3, 22, 25, 27, 56, 59, 71).

[Wei19] Matthew Weidner. *Group messaging for secure asynchronous collaboration*. MPhil dissertation. University of Cambridge, Cambridge, UK, 2019 (cit. on pp. 4, 79).

[Wei+21] Matthew Weidner, Martin Kleppmann, Daniel Hugenroth, and Alastair R. Beresford. "Key Agreement for Decentralized Secure Group Messaging with Strong Security Guarantees". In: *ACM CCS 2021*. Ed. by Giovanni Vigna and Elaine Shi. ACM Press, Nov. 2021, pp. 2024–2045. DOI: 10.1145/3460120.3484542 (cit. on pp. 4, 195, 197).

[Wik22] Wikipedia contributors. *PRISM — Wikipedia, The Free Encyclopedia*. [Online; accessed 8-January-2023]. 2022. URL: https://en.wikipedia.org/w/index.php?title=PRISM&oldid=1111536426 (cit. on p. 1).

[XL21] Yufei Xing and Shuguo Li. "A Compact Hardware Implementation of CCA-Secure Key Exchange Mechanism CRYSTALS-KYBER on FPGA". In: *IACR TCHES* 2021.2 (2021). https://tches.iacr.org/index.php/TCHES/article/view/8797, pp. 328–356. DOI: 10.46586/tches.v2021.i2.328-356 (cit. on p. 173).

[Xue+20] Haiyang Xue, Man Ho Au, Rupeng Yang, Bei Liang, and Haodong Jiang. *Compact Authenticated Key Exchange in the Quantum Random Oracle Model*. Cryptology ePrint Archive, Report 2020/1282. https://eprint.iacr.org/2020/1282. 2020 (cit. on p. 24).

[Xue+18] Haiyang Xue, Xianhui Lu, Bao Li, Bei Liang, and Jingnan He. "Understanding and Constructing AKE via Double-Key Key Encapsulation Mechanism". In: *ASIACRYPT 2018, Part II*. Ed. by Thomas Peyrin and Steven Galbraith. Vol. 11273. LNCS. Springer, Heidelberg, Dec. 2018, pp. 158–189. DOI: 10.1007/978-3-030-03329-3_6 (cit. on p. 24).

[Yan14] Zheng Yang. "Modelling Simultaneous Mutual Authentication for Authenticated Key Exchange". In: *Foundations and Practice of Security*. Ed. by Jean Luc Danger, Mourad Debbabi, Jean-Yves Marion, Joaquin Garcia-Alfaro, and Nur Zincir Heywood. Cham: Springer International Publishing, 2014, pp. 46–62. DOI: 10.1007/978-3-319-05302-8_4 (cit. on p. 33).

[YCL18] Zheng Yang, Yu Chen, and Song Luo. "Two-Message Key Exchange with Strong Security from Ideal Lattices". In: *CT-RSA 2018*. Ed. by Nigel P. Smart. Vol. 10808. LNCS. Springer, Heidelberg, Apr. 2018, pp. 98–115. DOI: 10.1007/978-3-319-76953-0_6 (cit. on pp. 21, 24, 26).

[YZ10] Andrew Chi-Chih Yao and Yunlei Zhao. "Deniable Internet Key Exchange". In: *ACNS 10*. Ed. by Jianying Zhou and Moti Yung. Vol. 6123. LNCS. Springer, Heidelberg, June 2010, pp. 329–348. DOI: 10.1007/978-3-642-13708-2_20 (cit. on pp. 22, 25, 27, 56, 59, 71).

[Yue+21] Tsz Hon Yuen, Muhammed F. Esgin, Joseph K. Liu, Man Ho Au, and Zhimin Ding. "DualRing: Generic Construction of Ring Signatures with Efficient Instantiations". In: *CRYPTO 2021, Part I*. Ed. by Tal Malkin and Chris Peikert. Vol. 12825. LNCS. Virtual Event: Springer, Heidelberg, Aug. 2021, pp. 251–281. DOI: 10.1007/978-3-030-84242-0_10 (cit. on pp. 25, 70).

Author's Publications

List of Publications Related to This Dissertation

Journal Papers

[Has+22] Keitaro Hashimoto, Shuichi Katsumata, Kris Kwiatkowski, and Thomas Prest. "An Efficient and Generic Construction for Signal's Handshake (X3DH): Post-quantum, State Leakage Secure, and Deniable". In: *Journal of Cryptology* 35.3 (May 2022). DOI: 10.1007/s00145-022-09427-1.

Refereed Conference Papers

[Has+21a] Keitaro Hashimoto, Shuichi Katsumata, Kris Kwiatkowski, and Thomas Prest. "An Efficient and Generic Construction for Signal's Handshake (X3DH): Post-Quantum, State Leakage Secure, and Deniable". In: *PKC 2021, Part II*. Ed. by Juan Garay. Vol. 12711. LNCS. Springer, Heidelberg, May 2021, pp. 410–440. DOI: 10.1007/978-3-030-75248-4_15.

[Has+21b] Keitaro Hashimoto, Shuichi Katsumata, Eamonn Postlethwaite, Thomas Prest, and Bas Westerbaan. "A Concrete Treatment of Efficient Continuous Group Key Agreement via Multi-Recipient PKEs". In: *ACM CCS 2021*. Ed. by Giovanni Vigna and Elaine Shi. ACM Press, Nov. 2021, pp. 1441–1462. DOI: 10.1145/3460120.3484817.

[HKP22] Keitaro Hashimoto, Shuichi Katsumata, and Thomas Prest. "How to Hide MetaData in MLS-Like Secure Group Messaging: Simple, Modular, and Post-Quantum". In: *ACM CCS 2022*. Ed. by Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi. ACM Press, Nov. 2022, pp. 1399–1412. DOI: 10.1145/3548606.3560679.

Non-refereed Papers

[Has+21] Keitaro Hashimoto, Shuichi Katsumata, Kris Kwiatkowski, and Thomas Prest. "Design and Implementation of a Post-Quantum Authenticated Key Exchange Protocol for Signal". In: *2021 Symposium on Cryptography and Information Security*. (In Japanese). Jan. 2021.

List of Other Publications

Journal Papers

[HO19] Keitaro Hashimoto and Wakaha Ogata. "Unrestricted and compact certificateless aggregate signature scheme". In: *Information Sciences* 487 (2019), pp. 97–114. DOI: https://doi.org/10.1016/j.ins.2019.03.005.

Non-refereed Papers

[HO18] Keitaro Hashimoto and Wakaha Ogata. "A Study on Construction and Security of Certificateless Aggregate Signature Scheme". In: *2018 Symposium on Cryptography and Information Security*. (In Japanese). Jan. 2018.

[HOT20] Keitaro Hashimoto, Wakaha Ogata, and Toi Tomita. "Tight reduction for generic construction of certificateless signature and its instantiation from DDH assumption". In: *2020 Symposium on Cryptography and Information Security*. (In Japanese). Jan. 2020.