論文 / 著書情報
Article / Book Information

題目(和文)	分散型秘匿 RAM および秘匿データ構造に関する研究
Title(English)	A Study on Distributed Oblivious RAM and Oblivious Data Structure
著者(和文)	市川敦謙
Author(English)	Atsunori Ichikawa
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12705号, 授与年月日:2024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12705号, Conferred date:2024/3/26, Degree Type:Course doctor, Examiner:,,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

A Study on Distributed Oblivious RAM and Oblivious Data Structure

Atsunori Ichikawa

Supervisor: Professor Wakaha Ogata

Department of Information and Communications Engineering
School of Engineering
Tokyo Institute of Technology

Abstract

A distributed oblivious RAM (DORAM) is a method for accessing a secret-shared memory while hiding the accessed locations. DORAMs are the key tool for secure multiparty computation (MPC) for RAM programs that avoids expensive RAM-to-circuit transformations. Oblivious Priority Queue (OPQ) is also known as a primitive closely related to (D)ORAM, that enables to introduce priority queue algorithms into secure computation.

We present new and improved 3-party DORAM protocols. For a logical memory of size N and for each logical operation, our DORAM requires $O(\log N)$ local CPU computation steps. This is known to be asymptotically optimal. Our DORAM satisfies passive security in the honest majority setting. Our technique results with concretely-efficient protocols and does not use expensive cryptography (such as re-randomizable or homomorphic encryption). Specifically, our DORAM is 25X faster than the known most efficient DORAM in the same setting.

Moreover, we extend our technique to handle malicious attackers at the expense of using slightly larger blocks (i.e., $\omega((\lambda + b) \log N)$ vs. $\lambda + b$ where $b = \Omega(\log N)$ is original block size). To the best of our knowledge, this is the first concretely-efficient maliciously secure DORAM.

Technically, our construction relies on a novel concretely-efficient 3-party oblivious permutation protocol. We combine it with efficient non-oblivious hashing techniques (i.e., Cuckoo hashing) to get a distributed oblivious hash table. From this, we build a full-fledged DORAM using a distributed variant of the hierarchical approach of Goldreich and Ostrovsky (J. ACM '96). These ideas, and especially the permutation protocol, are of independent interest.

Regarding an OPQ, we construct a novel perfectly secure OPQ that can simulate all PQ functions, inserting an element, extracting the top-priority one, deleting an element, and modifying the priority of an element, in $O(\log^2)$ work per query. Our scheme supports the same functions as the state-of-the-art statistical/computational OPQ with higher security, or achieve the same efficiency as the known perfectly secure OPQ with more functions.

Contents

Chapter 1

Introduction

Secure multiparty computation (MPC) is a method that enables mutually distrustful parties to jointly compute an arbitrary function over their private inputs. Since breakthrough feasibility results in the 80s, the quest for practically efficient MPC protocols is a central research area in cryptography. Efficiency is measured in terms of computation and/or communication, as a function of the size of the representation of the function that needs to be computed.

There are several common ways to represent computation, typically the circuit model or Random Access Machine (RAM) model. The former is known as a *MPC-friendly* model that can be simulated by gate-by-gate computation via traditional circuit-based MPC (e.g., [3, 16]). On the other hand, the latter is hard to simulate via MPC straightforwardly since RAM programs change their behavior according to its intermediate results that should be secret in MPC[1]. Indeed, any function can be computed in either of the models and a representation in one model can be translated to the other, thus any RAM program is *compatible* with circuit-based MPC *in principle*. However, such translations have a cost: a RAM program of size N can be turned into a circuit of size $O(N^3 \log N)$ [57]. Therefore, due to efficiency reasons, it would be highly desirable to be able to perform secure computation for RAM programs *directly*, i.e., without the translation.

There is a generic way to turn any RAM program into secure computation that simulates the same functionality but whose memory accesses do not reveal anything about the program's secret inputs except the input length. This is called an *Oblivious RAM* (ORAM), originally proposed by Goldreich and Ostrovsky [31]. The traditional setting for ORAMs is one client and one server. That is, an untrusted server has a large memory and a client can make accesses to it using a small trusted memory. Ostrovsky and Shoup [54] observed that, by simulating the client of an ORAM using traditional circuit-based MPC protocol, one can generically get an MPC for RAM programs. However, designing an efficient ORAM whose client behavior is "compatible" with circuit-based MPC is not at all obvious and has been (so far) sub-optimal in terms of the efficiency of the resulting protocol (see, e.g., [64]).

Due to the inherent inefficiency of circuit-based MPC for certain computations, there have been significant efforts in the last decade in building efficient MPCs for RAM programs, for example [49, 23, 64, 14, 9, 26]. By now, the common terminology for this problem is **Distributed Oblivious RAM** (DORAM)—informally, this is a protocol that allows parties to collectively maintain and perform reads/writes on a memory (a formal definition appears in Section 2.3).

Toward efficient MPC for RAM computation, reducing the complexity of DORAM is an ef-

[1] Consider the following toy example: Let a, b, c, d as inputs and assume a program that computes $\{$*if* $a = 0$ *then* $b \leftarrow bc$ *else* $b \leftarrow cd\}$. Now, the MPC circuit that simulates the program while hiding all inputs should always compute *both* bc and cd whatever a is since selecting either (b, c) or (c, d) directly leaks a is zero or not.

fective way. The complexity measure of DORAMs we are interested in is their *computational overhead*. That is, the maximal amount of CPU instructions[2] performed by each of the parties when serving a single access query.[3] Some prior works measure *bandwidth* overhead which accounts only for the maximal amount of bits communicated between the parties. Computational overhead is harder to optimize than communication overhead since an upper bound on the computational overhead implies an upper bound on the communication overhead, i.e., computational overhead $\geq$ communication overhead. The other direction is not necessarily true; indeed, some prior works (e.g., [26]) optimize communication overhead at the expense of increased computational overhead. In this work, we choose the more stringent measure. This is particularly important if we aim for concretely efficient and practically useful DORAMs.

A variant of a DORAM scheme due to Lu and Ostrovsky [49], suggested by Faber et al. [23],[4] gives a (3-party) DORAM with $O(\log N)$ computational overhead with block size[5] $\Omega(\log N)$. While the asymptotical overhead of this construction is optimal, the concrete efficiency is quite poor. The reason is that their scheme requires the parties to securely and jointly compute more than $100 \log N$ times of secure computation for pseudorandom functions[6] per access.

Later works attempt to present concretely efficient DORAMs. Wang et al. [64] and Faber et al. [23] proposed DORAMs with $O(\log N)$ computational overhead, but their block size is $\Omega(\log^2 N)$ which is less standard. Bunn et al. [9] constructed a DORAM with small hidden constants but poor asymptotic overhead ($\Omega(\sqrt{N})$). Most recently, Falk et al. [26] reduced the large constant factor of [49, 23] and achieved a scheme with $O(\log N)$ communication overhead with small hidden constant, but its computational overhead is sub-optimal $O(\log^2 N)$.

Moreover, all of the above schemes only guarantee security against a passive attacker, i.e., any single server cannot learn any non-trivial information about the others' inputs, as long as it follows the prescribed protocol. There are generic methods to boost security to the more standard setting of active security, where security holds even if a rouge server arbitrarily deviates from the protocol. However, these techniques do not preserve efficiency.

The current state of the art for 3-party DORAMs is summarized in Table 1.1. This brings us the main problems that we consider in this work:

Is there a 3-party DORAM that is asymptotically optimal in terms of computational overhead and concretely efficient? Additionally, is there an actively secure DORAM of the same overhead?

In this dissertation, we also consider *Oblivious Data Structures* (ODSs) that are cryptographic primitives relevant to (D)ORAMs. ODSs allow a client to store its data in an untrusted server while maintaining a certain structure and to operate the data in secret. They are used not only for a secure cloud system but also for secure graph algorithms [65, 40] and an oblivious streaming sampler for large-scale federated learning [60] in the context of secure computation.

ODSs are also often described as a computation model in which a trusted CPU manages a data structure in dishonest storage. In contrast to plain (non-oblivious) data structures, in ODS algorithms, the CPU should perform additional storage accesses to obfuscate the *access patterns*

[2]We model parties as RAM machines that can perform word-level addition and standard Boolean operations at unit cost.

[3]As commonly done, we sometimes settle for overhead in an amortized sense, that is, we measure the average overhead over a sequence of requests. Known schemes can be made worst-case ("de-amortized") [54, 6].

[4]The protocol of Lu and Ostrovsky [49] is in the multi-party setting where there are two non-communicating servers and a single trusted lightweight client (see Section 1.2).Faber et al. [23] observed that the client in [49]'s scheme can be efficiently simulated by an MPC.

[5]We call an element stored in memory as a *block*. Note that the block size is at least $\lceil \log N \rceil$ bits if there are N blocks.

[6]Note that a secure pseudorandom function requires a large circuit that can be a bottleneck of complexity.

that contain information about data and its access pattern. Thus, efficiency of an ODS is measured by its *total work* (or simply *work*), which indicates the number of data elements that the CPU interacts with the storage to perform one ODS operation.

Specifically, an *Oblivious Priority Queue* (OPQ) is one of the most popular ODSs that obliviously simulates a *priority queue* (PQ). A *queue* is a first-in first-out data structure, and a PQ is a queue that allows each data elements to be assigned a *priority*, and the elements can be retrieved in order of priority.

Through this work, we consider a PQ that supports the following four operations:

- **Inserting** an element with its *key* that indicates the priority of the element.
- **Extracting** the top-priority element, i.e., the first inserted element among ones with the smallest key in the PQ.
- **Deleting** a specified element of the PQ.
- **Modifying** the key of a specified element.

The first OPQ, proposed by Toft [63], supports only two operations, the *insertion* and *extraction*, each of which requires (amortized) $O(\log^2 N)$ total work where N is the capacity of the OPQ. His OPQ is perfectly secure but leaks the operation in progress. I.e., it is free from probabilistic data structure corruption and leakage of access patterns *in each operation*, but the adversary can observe whether the insertion or extraction is performed now *in a sequence of operations*. The recent work of Shi [60] improves the cost to optimal $O(\log N)$ work per query and supports all four operations described above while hiding the operations, but is statistically secure.

There are significant tradeoffs between perfect and statistical (or computational) security. For ORAM, achieving perfect security increase the total work considerably, e.g., there are numerous results of ORAMs that realize optimal $O(\log N)$ overhead as mentioned above, the best perfectly secure ORAM at present has $O(\log^3 N/\log\log N)$ overhead [13, 15]. On the other hand, for OPQ, there are no perfectly secure schemes other than Toft's work, and unfortunately, his scheme supports minimal operations and is less efficient than the recent works. It is an important theoretical question to clarify the boundary between perfect and other security, and the work for a perfectly secure OPQ is one step in that study direction.

Moreover, regardless of the tradeoffs, we stress that perfect security is worth enough even in practice. The modifying-key operation of PQ is, for example, used for Dijkstra's algorithm, and hence part of the known OPQs [40, 60, 39] realizes a secure computation for graphs. However, these OPQs have a failure probability that may not be negligible when N is small, so their advantage of efficiency cannot be maximized with a small graph.

In light of the above, we consider the following question:

Is there a perfectly secure OPQ that supports the modifying-key operation and others, in practical efficiency (at least asymptotically the same as Toft's OPQ [63])?

1.1 Contributions of this dissertation

1.1.1 An Optimal 3-Party DORAM

We present an asymptotically optimal and concretely efficient 3-party DORAM in the honest majority setting. Specifically, our DORAM has the computational overhead of $O(\log N)$ and the

Table 1.1: Summary of known *3-party DORAMs in the honest majority setting* together with our own schemes. Let N be the number of input elements and also treated as the statistical security parameter, $b = \Omega(\log N)$ be the size of the input element, and λ be the computational security parameter. The first column points to the paper that obtained the DORAM. The second column states the communication overhead of the proposed construction. The third column states the computational overhead of the proposed construction. The fourth column states whether the security guarantee is for passive or active attackers. The fifth column mentions the block size used in the construction. Lastly, the sixth column states whether the hidden constants are considered large or small.

Ref.	Communication	Computation	Security	Block size	Hidden const.
[54]	$O(\log^3 N)$	$O(\log^3 N)$	Passive	b	Large
[49, 23]	$O(\log N)$	$O(\log N)$	Passive	$\lambda + b$	Large
[64, 23]	$O(\log N)$	$O(\log N)$	Passive	$b + \Omega(\log^2 N)$	Large
[9]	$O(\sqrt{N})$	$O(\sqrt{N})$	Passive	b	Small
[26]	$O(\log N)$	$O(\log^2 N)$	Passive	$\lambda + b$	Small
[24]	$O(\log N)$	$O(\log N)$	Active	$\omega((\lambda + b)\log N)$	Small
Our	$O(\log N)$	$O(\log N)$	Passive	$\lambda + b$	Small
Our	$O(\log N)$	$O(\log N)$	Active	$\omega((\lambda + b)\log N)$	Small

hidden constant is rather small. Our DORAM requires at most $4 \log N$ oblivious pseudorandom function (OPRF) calls per query (amortized). This is significantly more efficient than the known optimal DORAM of Lu and Ostrovsky [49, 23] that requires at least $100 \log N$ calls per query, and is about 2 times greater than that of the DORAM of Falk et al. [26] but the DORAM of Falk et al. requires $O(\log^2 N)$ *computational* cost in addition to the communication cost (see Appendix Afor more detailed comparisons). This protocol is secure against passive (honest-but-curious) attackers.

A distributed oblivious permutation: Our main technical novelty is a new (concretely efficient and asymptotically optimal) 3-party oblivious permutation protocol. Our protocol can apply any permutation to the data with communication of $4nb+2n\lceil \log n \rceil$ bits and $12nb+2n\lceil \log n \rceil$ local computation steps where n is the number of data elements to be shuffled and b is the bit-length of each data element. We also construct a procedure to invert that permutation. This procedure requires $8nb + 2n\lceil \log n \rceil$ bits of communication and $19nb + 2n\lceil \log n \rceil$ steps of local computation.

A distributed oblivious hash table: Our DORAM construction is modular and, at a high level, is reminiscent of the hierarchical ORAM technique of Goldreich and Ostrovsky [31]. Recall that [31]'s hierarchical method basically reduces the problem of maintaining a memory to the problem of building a static hash table (supporting only lookups after an initial build). To this end, we implement a concretely efficient *distributed* oblivious hashing scheme. This is the first concretely efficient and asymptotically optimal distributed oblivious hash table construction. To store n data blocks of size b bits into a distributed hash table, each party needs to perform at most $O(n \cdot (\lambda + b + \lceil \log n \rceil))$ local CPU computation steps, and our lookup protocol requires $O(\lambda + b) + O(\sigma \cdot b)$ local CPU computation steps, where the first term is for a lookup in a main table and the second for a linear scan of a σ-size stash. The storage size of the hash table is $O(nb)$. We obtain our distributed oblivious hash table by first randomly permuting the data to be hashed (using the above-mentioned permutation protocol) and then simply invoking an off-the-shelf (non-oblivious) distributed hashing technique.

As a starting point of our study of ODS, we are motivated in the question of whether; *Is there a perfectly secure OPQ that supports the modifying-key operation and others, in practical efficiency*

Table 1.2: A comparison table for the known OPQs and ours. N denotes the capacity of OPQs.

Ref.	Total work	Operations	Op. hiding	Security
[63]	$O(\log^2 N)$	Insert, ExtractMin	✗	Perfect
[65]	$O(\log^2 N)$	Insert, ExtractMin	✓	Statistical
[40]	$O(\log^2 N)$	Insert, ExtractMin, ModKey	✓	Statistical
[60]	$O(\log N)$	Insert, ExtractMin, Delete, ModKey	✓	Statistical
[39]	$O(\log N)$	Insert, ExtractMin, Delete, ModKey	✓	Computational
Ours	$O(\log^2 N)$	Insert, ExtractMin, Delete, ModKey	✓	**Perfect**

(at least asymptotically the same as Toft's OPQ [63])?

Active security: We extend our passively secure schemes from above to be *actively* secure, without hurting efficiency, except that we rely on somewhat larger blocks. Specifically, we get a 3-party DORAM with $O(\log N)$ computational overhead and block size $\omega((\lambda + b) \log N)$. As far as we know, this is the first result of achieving active security for DORAM with practical efficiency guarantees. We do not know how to achieve similar concrete efficiency guarantees with logarithmic size blocks and we leave it as an exciting open problem.

1.1.2 A Perfectly Secure Oblivious Priority Queue

We show a novel *perfectly secure* OPQ that has the same functionalities as the recent schemes. I.e., our scheme supports *inserting, extracting, deleting*, and *modifying* an element[7] while hiding the operation in progress, and also it never impairs the correctness and also never leaks its access patterns. Similar to the known perfectly secure OPQ [63], our OPQ requires (amortized) $O(\log^2 N)$ total work per operation. We show a comparison for the known OPQs and ours in Table 1.2.

1.2 Related Works

Standard ORAM. The first ORAM, i.e., in the client-server model, was introduced by Goldreich and Ostrovsky [30, 53, 31]. Their best scheme had poly-logarithmic overhead in the memory size. They also proved a lower bound for a restricted class of schemes saying that logarithmic overhead is necessary [8]. Larsen and Nielsen [45] (see also [38, 42]) showed that logarithmic overhead is inherent for all schemes as long as they support "online" accesses.

There have been significant efforts to improve the overhead of ORAMs, both asymptotically and concretely [32, 61, 43, 62, 64, 12, 56, 4, 6]. A few years ago, Asharov et al. [4] (building on Patel et al. [56]) got an asymptotically optimal ORAM with logarithmic overhead. The constants underlying their scheme are enormous (due to the use of certain expander graphs; see also [22]) which makes it far from being practically useful. The best ORAM for practical purposes is still based on Path ORAM [62], a technique that suffers from $\log^2 N$ overhead in the memory size.

Multi-server ORAM. Another model that received significant attention is the *multi-server* ORAM setting. Here, there are multiple servers and a single client. The client can communicate with each server but the servers cannot communicate amongst themselves. As in the client-server setting of ORAM, the client has a small trusted memory. The adversarial model is that only some fraction of the servers are corrupted, making it possible for more efficient solutions than in the

[7]The scheme of [60] supports *finding the top-priority, increasing* and *decreasing a key* as well as inserting and deleting. We note that *finding* operation can be realized by *extract-then-insert*, and *increasing* and *decreasing* can be realized the general modifying operation.

standard single-server ORAM setting. This model was first introduced by Lu and Ostrovsky [49] who proposed a two-server ORAM with logarithmic communication overhead. By now, we have schemes with the same asymptotic overhead in the single-server setting [4]. Assuming larger block size, Kushilevitz and Mour [44] achieved a multi-server ORAM with sub-logarithmic communication from the client but poly-logarithmic work by the servers. Hoang et al. [34] proposed a multi-server ORAM with constant client-server communication overhead using secret sharing and MPC techniques. Lower bounds for this setting were proven by Larsen et al. [46], arguing that (up to constants) the multi-server setting is (asymptotically) as hard as the single server setting.

Actively secure ORAM. Since Ren et al. [58] introduced an actively secure ORAM in the standard model, various actively secure ORAMs have been proposed. Devadas et al. [21] proposed a generic construction of ORAMs that guarantees the correctness of metadata in the standard model. Also, Blass et al. [7] and Maffei et al. [50] proposed standard ORAMs (with multiple clients) that force the dishonest server or clients to behave correctly. Hoang et al. [33] construct a maliciously secure multi-server ORAM that detects servers that deviate from the protocol. Concurrently to this work, Falk et al. [24] proposed an actively secure DORAM with the same asymptotic cost as our construction.

Oblivious Priority Queue. The first OPQ was introduced by Toft [63]. That supports *insertion* and *extraction* of elements with $O(\log^2 N)$ total work per query and achieves perfect security but leaks which operation is in progress. Subsequently, Wang et al. [65] and Keller and Scholl [40] each constructed an OPQ with the same communication cost as Toft's OPQ, but which can hide the running operations. Moreover, the scheme of Keller and Scholl also supports *modifying a key*. However, these two OPQs are statistically secure, i.e., they impair the correctness of their structures or leak their access patterns stochastically (but in negligible probability). Jafargholi et al. [39] achieved an OPQ of the optimal $O(\log N)$ work per operation that supports all four operations described above while hiding the operations, depending on computationally secure hash functions. Shi [60] achieves an optimal statistically secure OPQ with the similar property as Jafargholi et al., but not relying on the existence of hash functions.

1.3 Organization

We give the preliminaries used throughout this thesis in Chapter 2. In Chapter 3, we describe our passively secure oblivious permutation and oblivious hashing protocols. We combine them to get our passively secure DORAM in Chapter 4. The actively secure extension is described in Chapter 5. Lastly, in Chapter 6.3, we show our perfectly secure oblivious priority queue.

Chapter 2

Preliminaries

In this chapter, we introduce notations, definitions, and building blocks that are used throughout this dissertation. For an integer $n \in \mathbb{N}$, we denote by $[n]$ the set $\{0, \ldots, n-1\}$. By $\|$ we denote the operation of string concatenation.

2.1 Pseudorandom Functions

Throughout this work, the computational security parameter is denoted λ, and it is given as input to algorithms in unary (i.e., as 1^λ). For the statistical security parameter, we use the input size N. A function $\mathsf{negl}\colon \mathbb{N} \to \mathbb{R}^+$ is *negligible* if for every constant $c > 0$ there exists an integer N_c such that $\mathsf{negl}(\lambda) < \lambda^{-c}$ for all $\lambda > N_c$. Two sequences of random variables $X = \{X_\lambda\}_{\lambda\in\mathbb{N}}$ and $Y = \{Y_\lambda\}_{\lambda\in\mathbb{N}}$ are *computationally indistinguishable* if for any probabilistic polynomial-time algorithm $\mathcal{A}$, there exists a negligible function $\mathsf{negl}(\cdot)$ such that $\left|\Pr[\mathcal{A}(1^\lambda, X_\lambda) = 1] - \Pr[\mathcal{A}(1^\lambda, Y_\lambda) = 1]\right| \leq \mathsf{negl}(\lambda)$ for all $\lambda \in \mathbb{N}$. We say that $X \equiv Y$ for such two sequences if they define *identical* random variables for every $\lambda \in \mathbb{N}$.

Definition 1 (Pseudorandom functions (PRFs))**.** *Let* F *be an efficiently computable function family indexed by keys* $\mathsf{sk} \in \{0,1\}^\lambda$*, where each* F_{sk} *takes as input a value* $x \in \{0,1\}^{n(\lambda)}$ *and outputs a value* $y \in \{0,1\}^{m(\lambda)}$*. A function family* F *is a secure* pseudorandom function *(PRF) if for every (non-uniform) probabilistic polynomial-time algorithm* $\mathcal{A}$*, there is a negligible function* $\mathsf{negl}(\cdot)$ *such that*

$$\left| \Pr_{\mathsf{sk}\leftarrow\$\{0,1\}^\lambda} \left[\mathcal{A}^{\mathsf{F}_{\mathsf{sk}}(\cdot)}(1^\lambda) = 1\right] - \Pr_{u\leftarrow\$U_\lambda} \left[\mathcal{A}^{u(\cdot)}(1^\lambda) = 1\right] \right| \leq \mathsf{negl}(\lambda),$$

for all $\lambda \in \mathbb{N}$*, where* U_λ *is the set of all functions mapping* $\{0,1\}^{n(\lambda)}$ *to* $\{0,1\}^{m(\lambda)}$*.*

It is known that one-way functions are existentially equivalent to PRFs for any polynomials $n(\cdot)$ and $m(\cdot)$ [35, 51]. Our construction will employ PRFs in several places and we present each part modularly with its own PRF, but note that the whole DORAM construction can be implemented with a single PRF from which we can implicitly derive all other PRFs.

2.2 Secret Sharing Schemes

A (threshold) secret sharing scheme (SSS) is a technique for "splitting" a secret between a collection of m parties such that a set of parties of some predefined cardinality, say $t+1$ for $1 \leq t \leq m-1$, can reconstruct the secret while smaller sets cannot. A "piece" that is held by a party is called a

share. We refer to such a scheme as (t, m)-SSS. Shamir's scheme [59] or the so-called replicated secret sharing scheme [37] are well-known implementations of such schemes.

To share and reconstruct a secret, we introduce three functionalities as follows. We use the notations $[\![\cdot]\!]_i$ to represent the share of party $i \in [m]$.

- $\mathcal{F}_{\text{SHARE}}$ receives a secret s and distributes the shares among the parties; share $[\![s]\!]_i$ is sent to party i.
- $\mathcal{F}_{\text{REVEAL}}$ receives shares $[\![s]\!]_i$ from at least $t+1$ parties, recovers the secret s, and sends it to all parties.
- $\mathcal{F}^{\mathcal{P}}_{\text{REVEAL}}$ behaves the same as $\mathcal{F}_{\text{REVEAL}}$ except that it sends the recovered secret s only to parties $\in \mathcal{P}$.

Also, we use the notations $\langle\cdot\rangle_{i\in\{0,1\}}$ to represent the shares in a $(1,2)$-*additive* SSS, i.e., $a = \langle a\rangle_0 + \langle a\rangle_1$ for any secret a. Under Shamir's SSS [59] or the replicated SSS [37], any two parties $P_i, P_{i+1 \bmod 3}$ can convert their shares $[\![s]\!]_i, [\![s]\!]_{i+1 \bmod 3}$ to $\langle s\rangle_0, \langle s\rangle_1$ by performing local computation.

We extend the notation to sharing arrays. For an array $\mathbf{A}$ of length X, we denote its x-th element by $\mathbf{A}[x]$. When secret sharing such an array, we denote its sharing by $[\![\mathbf{A}]\!] := ([\![a_0]\!], \ldots, [\![a_{X-1}]\!])$ and $\langle\mathbf{A}\rangle = (\langle a_0\rangle, \ldots, \langle a_{X-1}\rangle)$, where $a_x = \mathbf{A}[x]$.

For concreteness and clarity of this work, we assume that all shares $[\![s]\!]$ are of the $(1,3)$-replicated SSS on the extension field $\mathbb{Z}_{2^\ell}$ as [3, 18], i.e., for any $s = \sum_{i=0}^{\ell-1} s_i 2^i$; $s_i \in \mathbb{Z}_2$, its share is of the form $[\![s]\!] = \sum_{i=0}^{\ell-1} [\![s_i]\!] 2^i$. In this setup, all shares have the following properties.

Linear homomorphism: The replicated SSS [37], by definition, supports share-to-share addition by performing only local operations on each party's shares. That is, for any $a, b, c \in \mathbb{Z}_{2^\ell}$, without any interaction, the parties can compute

$$[\![a]\!] + [\![b]\!] = [\![a+b]\!] \text{ and } c \times [\![a]\!] = [\![ca]\!].$$

We extend the above notation and operations to arrays of secrets. For any arrays $[\![\mathbf{A}]\!] = ([\![a_0]\!], \ldots, [\![a_{X-1}]\!])$ and $[\![\mathbf{B}]\!] = ([\![b_0]\!], \ldots, [\![b_{X-1}]\!])$ where $a_i, b_i \in \mathbb{Z}_{2^\ell}$, we denote entry-wise addition as $[\![\mathbf{A}]\!] + [\![\mathbf{B}]\!] := ([\![a_0]\!] + [\![b_0]\!], \ldots, [\![a_{X-1}]\!] + [\![b_{X-1}]\!])$ and entry-wise multiplication by a scalar as $c \times [\![\mathbf{A}]\!] := (c \times [\![a_0]\!], \ldots, c \times [\![a_{X-1}]\!])$.

Bit-decomposition: Since all shares are of the form $[\![s]\!] = \sum_{i=0}^{\ell-1} [\![s_i]\!] 2^i$ where each $[\![s_i]\!]$ is a share of the replicated SSS on $\mathbb{Z}_2$, parties can perform the bit-decomposition operation $[\![s]\!] \to ([\![s_{\ell-1}]\!], \ldots, [\![s_0]\!])$ by their local conversion.

2.3 Distributed Oblivious RAM

A RAM consists of a memory of N cells and each cell is of size w bits and it allows for "clients" to perform read and write operations of the form $(\mathsf{op}, \mathsf{addr}, \mathsf{d})$, where $\mathsf{op} \in \{\mathsf{read}, \mathsf{write}\}$, $\mathsf{addr} \in [N]$ and $\mathsf{d} \in \{0,1\}^w \cup \{\perp\}$. If $\mathsf{op} = \mathsf{read}$, then $\mathsf{d} = \perp$ and the returned value is the content of the block located in logical address addr in the memory. If $\mathsf{op} = \mathsf{write}$, then the memory data in logical address addr is updated to d. We can think of this as an ideal (reactive) functionality $\mathcal{F}_{\text{RAM}}$ that supports the following operation:

$\underline{\mathcal{F}_{\text{RAM}}}$:

- $v \leftarrow \text{ACCESS}(\mathsf{op}, \mathsf{addr}, \mathsf{d})$: The input is an operation $\mathsf{op} \in \{\mathsf{read}, \mathsf{write}\}$, a key $\mathsf{addr} \in [N]$, and a value $\mathsf{d} \in \{0,1\}^w \cup \{\perp\}$. An internal size N array $\mathbf{X}$, initialized to all 0s, is maintained. The procedure does:

  1. If $\mathsf{op} = \mathsf{read}$, then set $\mathsf{d}^* = \mathbf{X}[\mathsf{addr}]$.

2. If $\mathsf{op} = \mathsf{write}$, then set $\mathbf{X}[\mathsf{addr}] = \mathsf{d}$ and $\mathsf{d}^* = \mathsf{d}$.
3. Output d^*.

2.3.1 Distributed oblivious simulation

Our goal is to simulate a RAM correctly while guaranteeing the standard security notion of secure multi-party computation. Towards this goal, we have m servers that can communicate between themselves over a fully connected network in synchronous rounds of communication. Each server can further perform arbitrary local computation between rounds. We model each server machine as a RAM. The view of each machine includes the contents of its own memory and the contents of the incoming messages, where the latter include addresses of memory cells to access. Such a secure system is called *Distributed Oblivious RAM* (DORAM). The security guarantee stipulates that the view of a colluding subset of dishonest servers cannot learn anything about the computation being performed, except what is absolutely necessary (e.g., the length of the computation). We shall consider a passive (semi-honest) or active (malicious) adversary who controls up to $t < m$ servers.

We shall define distributed oblivious simulation with respect to an arbitrary (possibly reactive or stateful) functionality. The definition for the RAM functionality will be implied as a special case. For concreteness, it is convenient (though not necessary) to imagine that the input and RAM state are secret-shared between the servers, that is, each party holds one out of m shares of the RAM, and operations from a client are also written to each server in a secret shared fashion. We follow the real-ideal paradigm by defining two "worlds" and requiring that they are indistinguishable (following, e.g., Canetti [10]).

Definition 2 (View)**.** *The view of party i consists of its auxiliary input and randomness followed by the honest input and all the messages sent and received by the party during the computation. Since we model parties as RAMs, the incoming and outgoing messages contain physical memory locations.*

In what follows, we suppress mentioning the auxiliary information to simplify notation and presentation. All of the definitions and results readily extend to the setting where auxiliary input is present.

2.3.2 Non-reactive functionalities

Let $\mathcal{F}$ be a non-reactive functionality. Let Π be a distributed protocol implementing $\mathcal{F}$, $C \subseteq [m]$ be the set of $\leq t$ corrupted servers, and Sim be a PPT simulator. Denote $\bar{C} = [m] \setminus C$. We introduce the following experiments to define active (malicious) security and remark the necessary changes for passive (semi-honest) security.

- $\mathsf{Real}^{\mathsf{nr}}_{\Pi,C,\mathcal{A}}(\lambda, \{x_i\}_{i\in[m]})$: Run the protocol with security parameter λ, where honest parties (ones not in C) run the protocol Π honestly with their private input $x_i^* = x_i$, whereas corrupt parties (ones in C) get the corresponding x_i's but can deviate from the prescribed protocol arbitrarily, according to $\mathcal{A}$'s strategy. Let V_i be the view of server $i \in C$ throughout the execution and let y_i be the output of some honest party $i \in \bar{C}$. Output $(\{V_i\}_{i\in C}, \{y_i\}_{i\in\bar{C}})$.

  In the passive (semi-honest) setting, the experiment is the same except that corrupt parties use $x_i^* = x_i$ and they follow the specification of the protocol (i.e., $\mathcal{A}$ is passive).

- $\mathsf{Ideal}^{\mathsf{nr}}_{\mathcal{F},\mathsf{Sim},C,\mathcal{A}}(\lambda, \{x_i\}_{i\in[m]})$: First, the adversary sees the inputs of corrupted parties $\{x_i\}_{i\in C}$ and outputs $\{x_i^*\}_{i\in C}$ that may depend on them. Denote $x_i^* = x_i$ for each $i \in \bar{C}$. Then, we

invoke the functionality $y_1, \ldots, y_m \leftarrow \mathcal{F}(x_1^*, \ldots, x_m^*)$. Finally, the simulator is executed and the following pair is outputted $(\mathsf{Sim}^{\mathcal{A}}(\lambda, C, \{x_i^*\}_{i\in C}), \{y_i\}_{i\in\bar{C}})$.

In the passive (semi-honest) setting, the experiment is the same except that the adversary is passive and uses $x_i^* = x_i$.

A distributed protocol obliviously simulates a functionality $\mathcal{F}$ against active (resp. passive) adversaries if the corrupted servers in the real world have views that are indistinguishable from their views in the ideal world.

Definition 3 (Distributed oblivious simulation of non-reactive functionalities)**.** *An m-server protocol Π (t, m)-obliviously simulates $\mathcal{F}$ if for any attacker there exists a PPT simulator* Sim *such that, for every subset of t passive/active corrupt parties C, any non-uniform PPT adversary $\mathcal{A}$, and all inputs $x_0, \ldots, x_{m-1}$, the distributions* $\mathsf{Real}^{\mathsf{nr}}_{\Pi,C,\mathcal{A}}(\lambda, \{x_i\}_{i\in[m]})$ *and* $\mathsf{Ideal}^{\mathsf{nr}}_{\mathcal{F},\mathsf{Sim},C,\mathcal{A}}(\lambda, \{x_i\}_{i\in[m]})$ *are computationally indistinguishable.*

2.3.3 Reactive functionalities

A reactive functionality is one that can be repeatedly invoked and it may keep an internal secret state between invocations (a RAM is, in particular, a reactive functionality). The adversary $\mathcal{A}$ chooses the next operation $(\mathsf{op}, \{x_i\}_{i\in[m]})$ adaptively in each stage. In the real execution, the corrupt parties may deviate arbitrarily from the prescribed protocol and the goal is to ensure that they do not learn anything beyond what is absolutely necessary. That is, we execute the protocol in the presence of the malicious adversary. In the ideal execution, the adversary obtains inputs $\{x_i\}_{i\in C}$ and may choose new inputs $\{x_i^*\}_{i\in C}$. The new inputs are fed (together with the inputs of the honest parties) into the functionality $\mathcal{F}$ which outputs an output $\{y_i\}_{i\in\bar{C}}$. At this point, using the output of malicious parties, a simulator must simulate the view of the malicious parties, including their internal state and the obtained messages (and access pattern) from other servers. The adversary can then choose the next command, as well as the next input, in an adaptive manner, based on everything it has seen so far.

Definition 4 (Distributed oblivious simulation of a reactive functionality)**.** *We say that a stateful protocol Π is a (t, m)-distributed oblivious implementation of the reactive functionality $\mathcal{F}$ if there exists a stateful PPT simulator* Sim*, such that for any non-uniform PPT (stateful) adversary $\mathcal{A}$, the view of the adversary $\mathcal{A}$ in the following two experiments* $\mathsf{Real}_{\Pi,C,\mathcal{A}}(\lambda, \{x_i\}_{i\in[m]})$ *and* $\mathsf{Ideal}_{\mathcal{F},\mathsf{Sim},C,\mathcal{A}}(\lambda, \{x_i\}_{i\in[m]})$ *is computationally indistinguishable:*

$\mathsf{Real}_{\Pi,C,\mathcal{A}}(\lambda, \{x_i\}_{i\in[m]})$*:*
Let $(\mathsf{op}, \{x_i\}_{i\in[m]}) \leftarrow \mathcal{A}\left(1^\lambda\right)$.
Loop while $\mathsf{op} \neq \perp$*:*
  Let $x_i' = (\mathsf{op}, x_i)$ *for each* $i \in [m]$.
  $\{V_i\}_{i\in C}, \{y_i\}_{i\in\bar{C}} \leftarrow \mathsf{Real}^{\mathsf{nr}}_{\Pi,C,\mathcal{A}}\left(\lambda, \{x_i'\}_{i\in[m]}\right)$.
  $(\mathsf{op}, \{x_i\}_{i\in[m]}) \leftarrow \mathcal{A}\left(1^\lambda, \{V_i\}_{i\in C}, \{y_i\}_{i\in\bar{C}}\right)$.

$\mathsf{Ideal}_{\mathcal{F},\mathsf{Sim},C,\mathcal{A}}(\lambda, \{x_i\}_{i\in[m]})$*:*
Let $(\mathsf{op}, \{x_i\}_{i\in[m]}) \leftarrow \mathcal{A}\left(1^\lambda\right)$.
Loop while $\mathsf{op} \neq \perp$*:*
  Let $x_i' = (\mathsf{op}, x_i)$ *for each* $i \in [m]$.
  $\{V_i\}_{i\in C}, \{y_i\}_{i\in\bar{C}} \leftarrow \mathsf{Ideal}^{\mathsf{nr}}_{\mathcal{F},\mathsf{Sim},C,\mathcal{A}}(\lambda, \{x_i'\}_{i\in[m]})$.
  $(\mathsf{op}, \{x_i\}_{i\in[m]}) \leftarrow \mathcal{A}\left(1^\lambda, \{V_i\}_{i\in C}, \{y_i\}_{i\in\bar{C}}\right)$.

2.4 Oblivious Priority Queue

A priority queue (PQ) is a data structure that allows to set a *priority* to each data and to extract the data in ascending order of the priorities. We assume an ideal (reactive) functionality $\mathcal{F}_{\mathsf{PQ}}$ that supports the following opertion:

$\underline{\mathcal{F}_{\mathsf{PQ}}}$, holding a set $\mathbf{Q}$ as its state.

- $\mathsf{ref} \leftarrow \mathcal{F}_{\mathsf{PQ}}(\mathsf{Insert}, k, v)$: Receiving an Insert request with inputs key k and value v, $\mathcal{F}_{\mathsf{PQ}}$ returns a reference ref that uniquely corresponds to (k, v) and records it as $\mathbf{Q} \leftarrow \mathbf{Q} \cup \{(k, v, \mathsf{ref})\}$.
- $(k_{\min}, v_{\min}) \leftarrow \mathcal{F}_{\mathsf{PQ}}(\mathsf{ExtractMin})$: $\mathcal{F}_{\mathsf{PQ}}$ removes $(k_{\min}, v_{\min}, \mathsf{ref}_{\min}) \in \mathbf{Q}$ of the minimum key from $\mathbf{Q}$ and returns $(k_{\min}, v_{\min})$. If there exists two or more tuples of the minimum k, according to FIFO, $\mathcal{F}_{\mathsf{PQ}}$ outputs one inserted at first of them.
- $\mathcal{F}_{\mathsf{PQ}}(\mathsf{Delete}, \mathsf{ref})$: $\mathcal{F}_{\mathsf{PQ}}$ removes (k, v, ref) from $\mathbf{Q}$ and returns it. If there are no tuples corresponding to ref, (k, v) is set as $(\perp, \perp)$.
- $\mathcal{F}_{\mathsf{PQ}}(\mathsf{ModKey}, k', \mathsf{ref})$: $\mathcal{F}_{\mathsf{PQ}}$ removes (k, v, ref) from $\mathbf{Q}$ and sets $\mathbf{Q} \leftarrow \mathbf{Q} \cup \{(k', v, \mathsf{ref})\}$. If there are no tuples corresponding to ref, (k, v) is set as $(\perp, \perp)$.

Note that, similar to Shi [60], we define above to suit for the standard interface of a priority queue, i.e., we assume that, to call Delete and ModKey, the reference ref is required as input to handle the item being deleted or modified. We in addition assume that the reference ref is simply the *time*, e.g., the number of operations requested so far, at which the element was inserted.

2.5 Oblivious One-Time Memory

An Oblivious One-Time Memory (OTM), originally proposed by Chan et al. [13, 15] as a building block of their perfectly secure ORAM, is an ODS that completely hides access patterns up to one access for each block. We assume an ideal functionality of OTM, $\mathcal{F}_{\mathsf{OTM}}$, as below.

$\underline{\mathcal{F}_{\mathsf{OTM}}}$:

- $\mathbf{U} \leftarrow \textsc{Build}(\mathbf{I})$: Receiving a set $\mathbf{I}$ containing N elements $\mathsf{d}_i; i = 1, \ldots, N$, it returns a list $\mathbf{U}$ containing N positions pos_i uniquely corresponding to each d_i. In addition, $\mathcal{F}_{\mathsf{OTM}}$ stores all N pairs $(\mathsf{d}_i, \mathsf{pos}_i)$ and states $\mathbf{P}$ in its memory.
- $\mathsf{d} \leftarrow \textsc{Lookup}(\mathsf{pos})$: The input pos is either real ($\in \mathbf{U}$) or dummy ($=\perp$). If pos is real, the output d is a real element s.t. $\mathcal{F}_{\mathsf{OTM}}$ holds a pair $(\mathsf{d}, \mathsf{pos})$, otherwise d is dummy. When $\mathsf{pos} \in \mathbf{P}$, $\mathcal{F}_{\mathsf{OTM}}$ halts without any output. Otherwise, it updates $\mathbf{P} \leftarrow \mathbf{P} \cup \{\mathsf{pos}\}$ then returns d.
- $\widetilde{\mathbf{I}} \leftarrow \textsc{Extract}()$: $\mathcal{F}_{\mathsf{OTM}}$ returns a set $\widetilde{\mathbf{I}}$ that contains all real d_i s.t. $(\mathsf{d}_i, \mathsf{pos}_i)$ is in its memory and $\mathsf{pos}_i \notin \mathbf{P}$. This $\widetilde{\mathbf{I}}$ is padded with dummy elements and has a size N.

Slightly different from the definition described in [13, 15], in the above functionality definition, we omit a key k_i originally assigned to each element d_i (and also to each position pos_i) since it is enough for our OPQ where the key k_i cannot uniquely identify an exact element d_i. We in our scheme need to refer an element only by its position (treated as part of its reference), not by its key.

A perfectly secure oblivious simulation of $\mathcal{F}_{\mathsf{OTM}}$ can be obtained by slightly modifying the general OTM construction [13, 15]. More specifically, the algorithm is obtained by eliminating k_i from $\mathbf{U}$ and by specifying the parameters so that it holds up to N elements, accept up to N lookup requests, and no parallel query is allowed. Here we briefly describe its construction focusing on the features used in our scheme.

OTM:

- $\mathbf{U} \leftarrow \textsc{Build}(\mathbf{I})$: Receiving the input set $\mathbf{I} = (\mathsf{d}_1, \ldots, \mathsf{d}_N)$, the algorithm selects an injection $f : \{1, \ldots, N\} \to \{1, \ldots, 2N\}$ uniform randomly and returns a list $\mathbf{U} = (\mathsf{pos}_1, \ldots, \mathsf{pos}_N)$ s.t. $\mathsf{pos}_i = f(i)$. In OTM, a table $\mathbf{T}$ of size $2N$ is maintained, in which all d_i is in $\mathbf{T}[\mathsf{pos}_i]$ and all other elements are dummy $\mathsf{d}_\perp$.

- $\mathsf{d} \leftarrow$ LOOKUP(pos): The input pos is real ($\in \mathbf{U}$) or dummy ($=\perp$). If pos is real, the algorithm fetches $\mathsf{d} = \mathbf{T}[\mathsf{pos}]$, and update $\mathbf{T}$ as $\mathbf{T}[\mathsf{pos}] \leftarrow \mathsf{d}_\perp$. If $\mathsf{pos} = \perp$, it fetches $\mathbf{T}[\mathsf{pos}_\perp] = \mathsf{d}_\perp$ and returns $\mathsf{d} = \mathsf{d}_\perp$. For obliviousness, the position $\mathsf{pos}_\perp$ is chosen to be unique over the lifetime of OTM[1]. In our scheme, since we need to know which position pos' ($\in \{\mathsf{pos}, \mathsf{pos}_\perp\}$) has been actually accessed, we include this pos' in the output of OTM.LOOKUP hereafter.
- $\widetilde{\mathbf{I}} \leftarrow$ EXTRACT(): The output is a set $\widetilde{\mathbf{I}}$ that contains all real elements remaining in $\mathbf{T}$ and dummies, to be a size N.

Fact 5. *The above* $\mathsf{OTM} = (\text{BUILD}, \text{LOOKUP}, \text{EXTRACT})$ *satisfies that: there exists a stateful simulator* $\mathsf{Sim}_{\mathsf{OTM}}$ *s.t. for any query sequence* $(\mathbf{I}, q_1, \ldots, q_m)$*, the simulation sequence* $\big(\mathsf{Sim}_{\mathsf{OTM}}(\text{BUILD}, N), \mathsf{Sim}_{\mathsf{OTM}}(\text{LOOKUP}, N), \ldots, \mathsf{Sim}_{\mathsf{OTM}}(\text{LOOKUP}, N), \mathsf{Sim}_{\mathsf{OTM}}(\text{EXTRACT}, N)\big)$ *is identical to the real access pattern* $\big(\mathsf{OTM}.\text{BUILD}(\mathbf{I}), \mathsf{OTM}.\text{LOOKUP}(\mathit{pos}_{q_1}), \ldots, \mathsf{OTM}.\text{LOOKUP}(\mathit{pos}_{q_m}), \mathsf{OTM}.\text{EXTRACT}()\big)$ *if* $N = |\mathbf{I}|$ *and*

- $m \le N$,
- $q_i \in \{1, \ldots, N, \perp\}$ *(where we assume* $\mathit{pos}_\perp = \perp$*) for all* $i \le m$*, and*
- $q_i \neq q_j$ *for all* $i, j \le m$ *and* $q_i, q_j \neq \perp$.

Moreover, for any input set $\mathbf{I}$ *of size* N*, the total work of* OTM.BUILD *and* OTM.EXTRACT *are* $O(N \log N)$*, and that of* OTM.LOOKUP *is* $O(1)$.

2.6 Secure Computation Building Blocks

Our schemes rely on various existing building blocks from the secure computation literature. To encapsulate these building blocks, we assume the existence of an Arithmetic Black Box (ABB) functionality, $\mathcal{F}_{\text{ABB}}$, which is a (reactive) multi-party functionality. $\mathcal{F}_{\text{ABB}}$ should consist of functions MULT, RND, RESHARE, EQ, IFELSE, BITEXT, TRUNC, and PRF, each listed below.

To simplify the notation, we denote *"calling a function p of* $\mathcal{F}_{\text{ABB}}$*"* as *"calling* $\mathcal{F}_p$*"*, e.g., *"parties call* $\mathcal{F}_{\text{MULT}}$*"* represents that the parties invoke $\mathcal{F}_{\text{ABB}}$ to call its function MULT with their inputs.

Assuming that all secrets are in $\mathbb{Z}_{2^b}$, all implementations we introduce below consume $O(b)$-bit communication and $O(b)$ local CPU computation steps except OPRFs that consume $O(\lambda + b)$-bits and steps.

Multiplication: Let $\mathcal{F}_{\text{MULT}}$ be a secure multiplication functionality that receives $[\![a]\!]$ and $[\![b]\!]$ and returns $[\![ab]\!]$. For the $(1,3)$-replicated SSS on the extension field $\mathbb{Z}_{2^\ell}$, we can use the implementation of Araki et al. [3] or Chida et al. [18]. For ease of notation, we occasionally denote $\mathcal{F}_{\text{MULT}}$ as $[\![a]\!] \times [\![b]\!]$.

Generating random shares: Let $\mathcal{F}_{\text{RND}}$ be a functionality that requires no inputs but returns a share $[\![r]\!]$ of a secret random value r. In the $(1,3)$-replicated SSS, since the form of shares is $[\![a]\!]_{i \mod 3} = (a_i, a_{i+1})$; $a = a_0 + a_1 + a_2$, $\mathcal{F}_{\text{RND}}$ is simply implemented in information-theoretical security as that: Each party P_i locally generate a random r_i, send it to P_{i-1} and set $[\![r]\!]_i$ as (r_i, r_{i+1}). It is also known that, trading off the information-theoretical security, a pseudorandom r can be shared without any communication except a pre-computation. This is called Pseudorandom Secret Sharing (PRSS) [19].

Resharing: Let $\mathcal{F}_{\text{RESHARE}}$ be a functionality that receives $\langle a \rangle_0$ and $\langle a \rangle_1$, and returns $[\![a]\!]$. It can be simply implemented as that the parties P_{i_0} and P_{i_1}, who have $\langle a \rangle_0$ and $\langle a \rangle_1$ respectively, secret-shares their shares as $[\![\langle a \rangle_i]\!]$ for all parties and then they observe $[\![a]\!] = [\![\langle a \rangle_0 + \langle a \rangle_1]\!]$. Under the use of PRSS, the slightly efficient implementation is known [17].

[1] In practice, as same as [13, 15], this *one-time dummy* lookup is realized by a linked list of dummy positions provided in BUILD. To simplify our algorithm descriptions, we omit these details here.

Equality test: Let $\mathcal{F}_{\text{EQ}}$ be a functionality that receives $[\![a]\!]$ and $[\![b]\!]$ then return $[\![c]\!]$ where $c \in \{0,1\}$ is equal to $(a =_? b)$. Though there are numerous implementations of these functionalities, for concreteness, we expect to use the one of Catrina and de Hoogh [11].

Selection: Let $\mathcal{F}_{\text{IfElse}}$ be a functionality that receives $[\![c]\!], [\![t]\!]$ and $[\![f]\!]$ such that $c \in \{0,1\}$, and returns $[\![t]\!]$ if $c = 1$, or $[\![f]\!]$ otherwise. Observe that $\mathcal{F}_{\text{IfElse}}$ is equal to $f + c(t - f)$.

Bit operations: Let $\mathcal{F}_{\text{BitExt}}, \mathcal{F}_{\text{Trunc}}$, and $\mathcal{F}_{\text{R\_Shift}}$ each be a functionality that receives shares $[\![a]\!]$, whose bit-representation is $a = a_\ell \dots a_1$, and an integer $i; 1 \leq i \leq \ell$, then returns the following output:

- $\mathcal{F}_{\text{BitExt}}([\![a]\!], i) \to [\![a_i]\!]$ s.t. $a_i (\in \{0,1\})$ is the i-th least significant bit of a.
- $\mathcal{F}_{\text{Trunc}}([\![a]\!], i) \to [\![a']\!]$ s.t. $a' = a_\ell \dots a_{i+1} \parallel 0^i$.
- $\mathcal{F}_{\text{R\_Shift}}([\![a]\!], i) \to [\![a']\!]$ s.t. $a' = 0^i \parallel a_\ell \dots a_{i+1}$.

Using the local bit-decomposition described in Section 2.2, $\mathcal{F}_{\text{BitExt}}$ can be implemented straightforwardly as follows: For $[\![a]\!] = \sum_{i=0}^{\ell-1} [\![a_i]\!] 2^i$, parties extract the target bit $[\![a_i]\!]$, generate $[\![\vec{0}]\!] = \sum_{i=0}^{\ell-1} [\![0]\!] 2^i$, and compute $[\![\vec{0}]\!] + ([\![a_i]\!] 2^0)$. $\mathcal{F}_{\text{Trunc}}$ and $\mathcal{F}_{\text{R\_Shift}}$ can be realized in a similar manner.

Oblivious PRF (OPRF): Let $\mathcal{F}_{\text{PRF}}$ be a functionality that receives shares $[\![\mathsf{sk}]\!]$ and $[\![x]\!]$ from all parties and sends them $[\![y]\!]$ where y is given by a PRF $\mathsf{F}_{\mathsf{sk}}(x)$. A combination of known oblivious block ciphers [2, 18, 20, 47] and $\mathcal{F}_{\text{R\_Shift}}$ implements $\mathcal{F}_{\text{PRF}}$.

Chapter 3

Efficient Passively Secure Distributed Oblivious Hashing

In this chapter, we propose a simple $(1,3)$-distributed oblivious hashing scheme that is inspired by Lu et al. [49]. Since Lu's DORAM requires too many (at least $100 \log N$) OPRF calls for distributed blocks, the practical computation cost becomes expensive even if the asymptotic overhead is down to $O(\log N)$. To reduce the practical computation complexity without increasing the asymptotic overhead, we construct a concretely efficient distributed oblivious hashing from a simple new permutation protocol for 3-party computation.

We present two different oblivious distributed hash table constructions. One hash table will be very efficient but will only work (i.e., be secure) if $n \in \Omega(\log^2 N)$. The other construction is much simpler and will work for smaller tables, which is a standard technique from prior works (e.g. [32]). At a very high level, both hash table constructions work as follows (assuming a permutation protocol):

1. Starting with $(1,3)$-shares of input blocks, the parties securely compute (pseudorandom) addresses for the blocks to be placed.

2. The parties divide their roles into one *permuter* and two *storages*, and then only the permuter reveals the addresses of the blocks.

3. The permuter computes a permutation for the blocks, which is a *sorting permutation* depending on the addresses.

4. The parties obliviously apply the permutation (that is secret for the storages) and the storages receive the $(1,2)$-shares of the permuted blocks.

5. Now, the permuted blocks can take the form of some hash table (if the permutation is valid), and only the storages can access the table.

   (a) If the input array is long enough (say $\Omega(\log^2 N)$) we can directly apply a distributed version of Cuckoo hashing. This results with linear time build and constant time lookup.

   (b) Otherwise, if the input array is too short (say $O(\log^2 N)$), we use a much simpler and standard hash table construction (also used by [32, 49] in a similar context). Set $\ell = 3 \log N / \log \log N$. We split the input into N/ℓ bins each of size ℓ by sending element with key k to bin $\mathsf{PRF}(k)$. Overflowing elements are routed to a σ-size stash (we will use $\sigma = \log N$). Lookup then costs $O(\ell + \sigma)$. Looking forward, we will use this hash

table construction for arrays of size $\log N, 2\log N, 4\log N, \ldots, \log^2 N$ and combine all of their stashes. Since there are $O(\log\log N)$ such tables and the lookup cost in each one is $O(\log N/\log\log N)$, the total cost will be $O(\log N + \sigma)$, as we want.

3.1 Reactive Functionality $\mathcal{F}_{\mathsf{HT}}$

A (static) hash table is a data structure supporting three operations BUILD, LOOKUP, and EXTRACT, that realizes the following reactive functionality. The BUILD procedure creates an in-memory data structure from an input array $\mathbf{I}$ containing real and dummy elements where each element is a (key, value) pair. Dummy elements have their key be $\perp$. It is assumed that all real elements in $\mathbf{I}$ have distinct keys. The LOOKUP procedure allows a requestor to look up the value of a key. A $\perp$ symbol is returned if the key is not found or if $\perp$ is the requested key. We say a (key, value) pair is visited if the key was searched for and found before. Finally, EXTRACT is the destructor and it returns a list containing unvisited elements padded with dummies to the same length as the input array $\mathbf{I}$.

The description of this functionality, denoted $\mathcal{F}_{\mathsf{HT}}$, is described next:

$\underline{\mathcal{F}_{\mathsf{HT}}}$:

- BUILD($\mathbf{I}$): The input is an array $\mathbf{I} = (a_1, \ldots, a_n)$ containing n elements, where each a_i is either dummy or a (key, value) pair denoted $a_i = (k_i, v_i)$. It is assumed that keys and values fit into $O(1)$ memory words and that all real keys are distinct. The procedure does:
  1. Initialize the state $\mathsf{H} = (\mathbf{I}, \mathbf{P})$ where $\mathbf{P} = \emptyset$.
- LOOKUP(k): The input is a key k (that might be $\perp$, i.e., dummy). The procedure does:
  1. If $k \in \mathbf{P}$ (i.e., k is a recurring lookup), then halt and return $\perp$.
  2. If $k = \perp$ or $k \notin \mathbf{I}$, set $v^* = \perp$.
  3. Otherwise, set $v^* = v$, where v is the value corresponding to $k(\in \mathbf{I})$.
  4. Update $\mathbf{P} = \mathbf{P} \cup \{k\}$.
  5. Output v^*.
- EXTRACT(): The procedure does:
  1. Define $\mathbf{I}' = \{a'_1, \ldots, a'_n\}$ such that: For $i \in [n]$, set $a'_i = a_i$ if $a_i = (k, v)$ and $k \notin \mathbf{P}$. Otherwise, set $a'_i = \perp$.
  2. Output $\mathbf{I}'$.

3.2 Distributed Oblivious Permutation

As a building block for our distributed oblivious hash table, we first construct an efficient 3-party secure permutation protocol. The precise functionality that we implement is described below. This section is devoted to the implementation of these functionalities via a 3-party protocol where at most one may be corrupted.

$\underline{\mathcal{F}_{\text{PERM}}}$:

1. Receive a permutation π from the *permuter* P, and receive $(1,3)$-shares of an array $[\![\mathbf{I}]\!]$ from all parties.
2. Obtain $\mathbf{I}$, compute $\pi \cdot \mathbf{I}$, and choose a random string $\mathbf{R}$ of the same size as $\mathbf{I}$.

3. Send $\langle \mathbf{I}' \rangle_0 = \mathbf{R}$ to the first *storage* S_0 and $\langle \mathbf{I}' \rangle_1 = \pi \cdot \mathbf{I} - \mathbf{R}$ to the second S_1.

$\mathcal{F}_{\text{UNPERM}}$:

1. Receive a permutation π from the *permuter* P and 2-out-of-2 shares of an array $\langle \mathbf{I} \rangle_0, \langle \mathbf{I} \rangle_1$ from the *storages* S_0, S_1.
2. Reconstruct $\mathbf{I}$ and compute $\pi^{-1} \cdot \mathbf{I}$.
3. Return $[\![\mathbf{I}']\!] \leftarrow \mathcal{F}_{\text{SHARE}}(\pi^{-1} \cdot \mathbf{I})$ for all parties.

Lemma 6 (Realization of $\mathcal{F}_{\text{PERM}}$)**.** *There is a (1,3)-distributed oblivious implementation, described as Algorithm 1, of $\mathcal{F}_{\text{PERM}}$ in the presence of a passive adversary that controls one party. The protocol consumes $4nb + 2n\lceil \log n \rceil$ bits of communication and $12nb + 2n\lceil \log n \rceil$ local CPU computation steps where n is the number of blocks in the input array and b is the bit-length of each block. It also consumes 2 communication rounds.*

Lemma 7 (Realization of $\mathcal{F}_{\text{UNPERM}}$)**.** *There is a distributed 3-party protocol, described as Algorithm 2, that securely realizes $\mathcal{F}_{\text{UNPERM}}$ in the presence of a passive adversary that controls one party and in the $\mathcal{F}_{\text{RESHARE}}$-hybrid model. By composition and using the implementation of $\mathcal{F}_{\text{RESHARE}}$ described in Section 2.6, the protocol consumes $8nb + 2n\lceil \log n \rceil$ bits of communication cost and $19nb + 2n\lceil \log n \rceil$ local CPU computation steps where n is the number of blocks in the input array and b is the bit-length of each block. It also consumes 3 communication rounds.*

Algorithm 1 $(\cdot, \langle \mathbf{I}' \rangle_0, \langle \mathbf{I}' \rangle_1) \leftarrow \Pi_{\text{PERM}}((\pi, [\![\mathbf{I}]\!]_0), [\![\mathbf{I}]\!]_1, [\![\mathbf{I}]\!]_2)$

Notation: P is the "permuter" and S_0, S_1 are two "storages."
Require: P has a permutation π and each party has shares of an array $[\![\mathbf{I}]\!]$.
Ensure: $\mathbf{I}' = \pi \cdot \mathbf{I}$.

1: P and S_0 convert their $(1,3)$-shares $[\![\mathbf{I}]\!]_0, [\![\mathbf{I}]\!]_1$ to $(1,2)$-shares $\langle \mathbf{I} \rangle_0, \langle \mathbf{I} \rangle_1$, respectively.
2: P chooses random strings $\mathbf{U}, \mathbf{V}$ of the same size as $\langle \mathbf{I} \rangle_0$, and random permutations π_0, π_1 s.t. $\pi_1 \circ \pi_0 = \pi$.
3: P sends $\pi_0, \mathbf{U}, \mathbf{V}$ to S_0 and $\pi_1, \widetilde{\mathbf{I}}_0 := \pi \cdot \langle \mathbf{I} \rangle_0 - \pi_1 \cdot \mathbf{U} - \mathbf{V}$ to S_1.
4: S_0 sends $\widetilde{\mathbf{I}}_1 := \pi_0 \cdot \langle \mathbf{I} \rangle_1 + \mathbf{U}$ to S_1.
5: S_0 outputs $\langle \mathbf{I}' \rangle_0 := \mathbf{V}$, and S_1 outputs $\langle \mathbf{I}' \rangle_1 := \widetilde{\mathbf{I}}_0 + \pi_1 \cdot \widetilde{\mathbf{I}}_1$.

Algorithm 2 $[\![\mathbf{I}']\!]_0, [\![\mathbf{I}']\!]_1, [\![\mathbf{I}']\!]_2 \leftarrow \Pi_{\text{UNPERM}}(\pi, \langle \mathbf{I} \rangle_0, \langle \mathbf{I} \rangle_1)$

Notation: P is the "permuter" and S_0, S_1 are two "storages."
Require: P has a permutation π and S_0, S_1 have a shares $\langle \mathbf{I} \rangle_0, \langle \mathbf{I} \rangle_1$, respectively.
Ensure: $\mathbf{I}' = \pi^{-1} \cdot \mathbf{I}$.

1: S_1 chooses a random strings $\mathbf{U}, \mathbf{V}$ of the same size as $\langle \mathbf{I} \rangle_0$.
2: P chooses random permutations π_0, π_1 s.t. $\pi_1 \circ \pi_0 = \pi$.
3: S_1 sends $\mathbf{U}$ to P and $\mathbf{V}$ to S_0. P sends π_0 to S_0 and π_1 to S_1.
4: S_0 sends $\widetilde{\mathbf{I}}_0 := \langle \mathbf{I} \rangle_0 + \mathbf{V}$ to P. S_1 sends $\widetilde{\mathbf{I}}_1 := \pi_1^{-1} \cdot (\langle \mathbf{I} \rangle_1 - \mathbf{V}) - \mathbf{U}$ to S_0.
5: P computes $\langle \mathbf{I}' \rangle_0 := \pi^{-1} \cdot \widetilde{\mathbf{I}}_0 + \pi_0^{-1} \cdot \mathbf{U}$, and S_0 computes $\langle \mathbf{I}' \rangle_1 := \pi_0^{-1} \cdot \widetilde{\mathbf{I}}_1$.
6: Parties call $\mathcal{F}_{\text{RESHARE}}$ to convert $\langle \mathbf{I}' \rangle_0, \langle \mathbf{I}' \rangle_1$ to $[\![\mathbf{I}']\!]_0, [\![\mathbf{I}']\!]_1, [\![\mathbf{I}']\!]_2$.

Proof of Lemma 6. The output of S_0 is $\mathbf{V}$ and the output of S_1 is

$$\begin{aligned} \widetilde{\mathbf{I}}_0 + \pi_1 \cdot \widetilde{\mathbf{I}}_1 &= \pi \cdot \langle \mathbf{I} \rangle_0 - \pi_1 \cdot \mathbf{U} - \mathbf{V} + \pi_1 \circ \pi_0 \cdot \langle \mathbf{I} \rangle_1 + \pi_1 \cdot \mathbf{U} \\ &= \pi \cdot \mathbf{I} - \mathbf{V}. \end{aligned}$$

Together, they form a $(1,2)$-share of $\pi \cdot \mathbf{I}$, as needed for correctness. The claimed efficiency follows by direct inspection. The strings $\mathbf{U}$ and $\mathbf{V}$ are each nb bits long, similarly to $\widetilde{\mathbf{I}}_0$ and $\widetilde{\mathbf{I}}_0$. The bit-length of π_0 and π_1 is $n\lceil \log n \rceil$, each.

For security, observe that the permuter P never receives any message and does not have any output, and therefore it is trivial to simulate its view. Similarly, the first storage server S_0 only gets one message $\mathbf{U}, \mathbf{V}, \pi_0$ (from P), all of which are uniformly random and independent of the inputs of all parties. The output of S_0 contains $\mathbf{V}$ and so overall it is immediate to simulate its view. The only case remaining is when S_1 is corrupted. Its view in the protocol consists of $\pi_1, \widetilde{\mathbf{I}}_0, \widetilde{\mathbf{I}}_1$ which can be simulated by 3 uniformly random strings of appropriate length. Indeed, $\widetilde{\mathbf{I}}_0$ is masked by $\mathbf{V}$ and then $\widetilde{\mathbf{I}}_1$ is masked by $\mathbf{U}$, all of which are not known to S_1. □

Proof of Lemma 7. Since we are in the $\mathcal{F}_{\text{RESHARE}}$-hybrid model, for correctness we need to show that $\langle \mathbf{I}' \rangle_0 + \langle \mathbf{I}' \rangle_1 = \pi^{-1} \cdot \mathbf{I}$. Indeed,

$$\begin{aligned}\langle \mathbf{I}' \rangle_0 + \langle \mathbf{I}' \rangle_1 &= \pi^{-1} \cdot (\langle \mathbf{I} \rangle_0 + \mathbf{V}) + \pi_0^{-1} \cdot \mathbf{U} + \pi_0^{-1} \cdot (\pi_1^{-1} \cdot (\langle \mathbf{I} \rangle_1 - \mathbf{V}) - \mathbf{U}) \\ &= \pi^{-1} \cdot \mathbf{I}.\end{aligned}$$

The claimed efficiency follows by direct inspection. The strings $\mathbf{U}$ and $\mathbf{V}$ are each nb bits long, similarly to $\widetilde{\mathbf{I}}_0$ and $\widetilde{\mathbf{I}}_0$. The bit-length of π_0 and π_1 is $n\lceil \log n \rceil$, each.

For security, if S_1 is corrupted, we can easily simulate its view as it does not receive any message except π_1 which is uniformly random and independent of the other inputs. If P is corrupted, then it again immediately simulates its view since it only receives $\mathbf{U}$ and $\widetilde{\mathbf{I}}_0 := \langle \mathbf{I} \rangle_0 + \mathbf{V}$ throughout the execution, both of which are uniformly distributed (in P's view). Lastly, assume that S_0 is corrupted. Its view consists of $\mathbf{V}, \pi_0$ and $\widetilde{\mathbf{I}}_1 := \pi_1^{-1} \cdot (\langle \mathbf{I} \rangle_1 - \mathbf{V}) - \mathbf{U}$. Since it does not know $\mathbf{U}$, the term $\widetilde{\mathbf{I}}_1$ looks completely uniform and therefore the whole view can be simulated by 3 uniformly random strings of the appropriate length. □

3.3 Distributed Oblivious Hashing for Short Inputs

Equipped with the above permutation protocols, we present a distributed oblivious hash table for short input. Specifically, the hash table that we give out here works best when n, the input array size, is $O(\log^2 N)$. Similar hash was employed in [32, 49] in a similar context.

Consider a balls-into-bins hash table T of size τ with bin size β and a stash S of size σ. Below, we construct a distributed hash table $\langle \mathsf{T} \rangle$ and stash $[\![\mathsf{S}]\!]$ from n elements (Algorithm 3), support lookup in the main table (Algorithm 4) and in the stash (Algorithm 5), and finally deconstruct the structure (Algorithm 6). We rely on our role-splitting permutation protocols and an OPRFs.

Lemma 8. *Assume that the input array consists of $n \le \log^2 N$ blocks and each block is b bits. Also assume that the table size $\tau = n$, the bin size $\beta = \lceil 3 \log N / \log \log N \rceil$, and the stash size $\sigma = \lceil \log N \rceil$. There exists a distributed 3-party protocol, described as Algorithms 3, 4, 5, and 6, that securely realizes $\mathcal{F}_{\mathsf{HT}}$ in the $(\mathcal{F}_{\text{SHARE}}, \mathcal{F}^{\mathcal{P}}_{\text{REVEAL}}, \mathcal{F}_{\text{ABB}}, \mathcal{F}_{\text{PERM}}, \mathcal{F}_{\text{UNPERM}})$-hybrid model and in the presence of a passive adversary that controls one party.*

- *The* BUILD *procedure consumes $O(n(\lambda + b + \log n))$ local computation steps.*
- *The* LOOKUP *procedure consumes $O((\lambda + b)\frac{\log N}{\log \log N})$ local computation steps.*
- *The* STASHSEARCH *procedure consumes $O(b \log \lambda)$ local computation steps.*
- *The* EXTRACT *procedure consumes $O(nb)$ local computation steps.*

Furthermore, all of the above procedures consume $O(1)$ communication rounds.

Algorithm 3 $\pi, \langle \mathsf{T} \rangle_{0,1}, [\![\mathsf{S}]\!], [\![s]\!], \mathsf{ctr}, \mathbf{P} \leftarrow \Pi_{\text{BUILD}}([\![\mathsf{D}]\!]_0, [\![\mathsf{D}]\!]_1, [\![\mathsf{D}]\!]_2)$

Notation: Let P be the permuter and S_0, S_1 be the storages. Let $\tau := |\mathsf{T}|$ and $\sigma := |\mathsf{S}|$ be the size of expected hash table and stash (T, S) storing n items, and let $n' = \tau + \sigma$.

Require: Parties have $(1,3)$-shares of a dataset $[\![\mathsf{D}]\!]_{i \in \{0,1,2\}} = ([\![\mathsf{d}_0]\!]_i, \ldots, [\![\mathsf{d}_{n-1}]\!]_i)$.

Ensure: S_0 and S_1 obtain $(1,2)$-shares of a balls-into-bins hash table, $\langle \mathsf{T} \rangle_0$ and $\langle \mathsf{T} \rangle_1$, respectively. P obtains a permutation π, and all parties hold $(1,3)$-shares of a stash $[\![\mathsf{S}]\!]$ and a PRF key $[\![s]\!]$. Parties also hold a query counter ctr, and S_0, S_1 hold a set $\mathbf{P}$, as their states.

(Computing pseudorandom addresses for the input data.)

1: Parties call $\mathcal{F}_{\text{RND}}$ to generate a random PRF key $[\![s]\!]$.
2: **for all** $i \in [n]$ **do**
3: Parties call $\mathcal{F}_{\text{EQ}}$ to obtain $[\![\mathsf{isDummy}_i]\!] := [\![k_i =_? \perp]\!]$.
4: Parties call $\mathcal{F}_{\text{IfElse}}$ to simulate:
If $\mathsf{isDummy}_i = 1$*, then* $[\![\widetilde{k_i}]\!] := [\![\perp + i]\!]$*; otherwise* $[\![\widetilde{k_i}]\!] := [\![k_i]\!]$.
5: Parties call $\mathcal{F}_{\text{PRF}}$ to obtain virtual addresses $[\![\mathsf{addr}_i]\!]$ from $([\![s]\!], [\![\widetilde{k_i}]\!])$.

(Building a balls-into-bins hash table via permutation.)

6: Parties call $\mathcal{F}_{\text{SHARE}}$ to generate an array $[\![\mathsf{E}]\!]$ consisting of $n' - n$ dummy blocks.
7: Let $[\![\widetilde{\mathsf{D}}]\!] := [\![\mathsf{D}]\!] \parallel [\![\mathsf{E}]\!]$ be a concatenated dataset.
8: Parties call $\mathcal{F}_{\text{REVEAL}}^{\{P\}}$ to reveal to P all $\mathsf{addr}_{i,0}, \mathsf{addr}_{i,1}$ for $i \in [n]$.
9: P computes a permutation $\pi \colon [n'] \to [n']$ that indicates the bin-placements of elements. That is, π says where to place $\widetilde{\mathsf{D}}$'s elements into T (or S) as indicated by addr_i.
10: Parties call $\mathcal{F}_{\text{PERM}}$ with π and $[\![\widetilde{\mathsf{D}}]\!]$ to make S_0 and S_1 obtain $\langle \widetilde{\mathsf{D}'} \rangle_0$ and $\langle \widetilde{\mathsf{D}'} \rangle_1$ respectively, where $\widetilde{\mathsf{D}'} = \pi \cdot \widetilde{\mathsf{D}}$.
11: Each S_i for $i \in \{0,1\}$ organizes the array $\langle \widetilde{\mathsf{D}'} \rangle_i$ into a hash table $\langle \mathsf{T} \rangle_i$ and stash $\langle \mathsf{S} \rangle_i$, by separating $\langle \widetilde{\mathsf{D}'} \rangle_i$ into the first τ and the last σ elements.
12: Parties call $\mathcal{F}_{\text{RESHARE}}$ to convert $\langle \mathsf{S} \rangle_{0,1}$ to $[\![\mathsf{S}]\!]$.
13: Parties set their state $\mathsf{ctr} = 0$, and S_0, S_1 allocate an empty set $\mathbf{P} = \emptyset$.
14: **Return** $\pi, \langle \mathsf{T} \rangle_{0,1}, [\![\mathsf{S}]\!], [\![s]\!], \mathsf{ctr}, \mathbf{P}$

Proof sketch. We sketch complexity and why the construction is secure for completeness because this hash table has been used many times in the ORAM literature. For instance, in our range of parameters, it was directly used in [49, Section 3.5 of the full version]. Recall that our input is already randomly shuffled and so which element goes to which bin is completely hidden from the adversary. Also the bins are padded to their maximum capacity. (Notice that the permuter knows the PRF key and so can pad appropriately but never sees lookup queries.) Also, by the analysis of [49, Section 3.5 of the full version], except with negligible probability in N, a σ-size stash suffices to store all overflowing elements and so the construction is successful. □

3.4 Distributed Oblivious Hashing for Long Inputs

The idea is simple to describe at a high level: given a set of elements we first obliviously permute them and then we index the permuted set using a non-oblivious efficient hashing scheme. For the latter, we use *Cuckoo hashing* [55], a hashing paradigm that resolves collisions in a table by using two hash functions and two tables, cleverly assigning each element to one of the two tables, and enabling lookup using only two queries. The standard version of Cuckoo hashing suffers from inverse polynomial probability of build failure which does not suffice for our application (since we aim for negligible error). To this end, we use a variant of Cuckoo hashing where items that cannot be stored in one of the two tables are stored in a (typically small) "stash". According to Noble [52], for any number of elements $n = \omega(\log N)$, there exists Cuckoo hashing that has the table of size $\tau = (1+\epsilon)n$ and the stash of size $\sigma = \Theta(\log N)$ with negligible failure probability. Henceforth, we

Algorithm 4 $[\![\mathsf{d}]\!], [\![\mathsf{found}]\!] \leftarrow \Pi_{\text{LOOKUP}}([\![k]\!], \langle \mathsf{T} \rangle_{0,1}, [\![s]\!], \mathsf{ctr}, \mathbf{P})$

Notation: Let P be the permuter and let S_0, S_1 be the storages. Let β and γ be the bin size and number of bins, respectively, i.e., $|\mathsf{T}| = \tau = \beta\gamma$.

Require: $[\![k]\!]$ is a $(1,3)$-share of an input key to be searched for. S_i have the distributed hash table $\langle \mathsf{T} \rangle_i$ and a set $\mathbf{P}$. Parties have $(1,3)$-shares of a PRF key $[\![s]\!]$ and a query counter ctr.

Ensure: $\mathsf{d} = (k, v), \mathsf{found} = 1$ if T contains (k, v). Otherwise, $\mathsf{d} = (0, 0), \mathsf{found} = 0$.

(Computing pseudorandom addresses to be fetched.)

1: Parties call $\mathcal{F}_{\text{EQ}}$ to obtain $[\![\mathsf{isDummy}]\!] := [\![k =_? \perp]\!]$.

2: Parties call $\mathcal{F}_{\text{IFELSE}}$ to simulate:
If $\mathsf{isDummy} = 1$, *then set* $[\![\widetilde{k}]\!] := [\![\perp + \mathsf{ctr}]\!]$; *otherwise* $[\![\widetilde{k}]\!] := [\![k]\!]$.

3: Parties call $\mathcal{F}_{\text{PRF}}$ to obtain $[\![\mathsf{addr}]\!]$ from $([\![s]\!], [\![\widetilde{k}]\!])$.

4: Parties call $\mathcal{F}_{\text{REVEAL}}^{\{S_0,S_1\}}$ to recover addr to both S_0 and S_1. If $\mathsf{addr} \in \mathbf{P}$, S_0 and S_1 halt. Otherwise, they update $\mathbf{P} \leftarrow \mathbf{P} \cup \{\mathsf{addr}\}$.

(Searching for the table T.)

5: **for all** $i \in [\beta]$ **do**

6: Parties call $\mathcal{F}_{\text{RESHARE}}$ to convert $\langle \mathsf{T}[\gamma \cdot \mathsf{addr} + i] \rangle$ to $([\![k_i]\!], [\![v_i]\!])$, i.e., fetch the i-th element of the addr-th bin.

7: Parties call $\mathcal{F}_{\text{EQ}}$ to obtain $[\![\mathsf{isQueried}_i]\!] = [\![k_i =_? k]\!]$.

8: Parties call $\mathcal{F}_{\text{IFELSE}}$ and $\mathcal{F}_{\text{MULT}}$ to simulate:
If $\mathsf{isDummy} = 0$ *and* $\mathsf{isQueried}_i = 1$,
then $\mathsf{found} = 1$, $[\![\mathsf{d}]\!] = ([\![k_i]\!], [\![v_i]\!])$, *and* $\langle \mathsf{T}[\gamma \cdot \mathsf{addr} + i] \rangle = \langle \mathsf{d}^{\mathsf{dummy}} \rangle$.

9: Parties increment ctr.

10: **Return** $[\![\mathsf{d}]\!], [\![\mathsf{found}]\!]$

Algorithm 5 $[\![\mathsf{d}]\!], [\![\mathsf{found}]\!] \leftarrow \Pi_{\text{STASHSEARCH}}([\![k]\!], [\![\mathsf{S}]\!])$

Require: $[\![k]\!]$ is a $(1,3)$-share of an input key to be searched for, and $[\![\mathsf{S}]\!]$ is an array consisting of σ shares of key-value pairs.

Ensure: $\mathsf{d} = (k, v), \mathsf{found} = 1$ if S contains (k, v). Otherwise, $\mathsf{d} = (0, 0), \mathsf{found} = 0$.

1: **for all** $([\![k_u]\!], [\![v_u]\!])_{u \in [\sigma]}$ in $[\![\mathsf{S}]\!]$ **do**

2: Parties call $\mathcal{F}_{\text{EQ}}$ to obtain $[\![\mathsf{isQueried}_u]\!] = [\![k_u =_? k]\!]$.

3: Parties call $\mathcal{F}_{\text{IFELSE}}$ and $\mathcal{F}_{\text{MULT}}$ to simulate:
If $\mathsf{isQueried}_u = 1$, *then* $\mathsf{found} = 1$, $[\![\mathsf{d}]\!] = ([\![k_u]\!], [\![v_u]\!])$, *and* $[\![\mathsf{S}[u]]\!] = [\![\mathsf{d}^{\mathsf{dummy}}]\!]$.

4: **Return** $[\![\mathsf{d}]\!], [\![\mathsf{found}]\!]$

specify $\tau = 2n$ and $\sigma = \lceil \log N \rceil$ for concrete efficiency analysis.

To ease presentation of our implementations, we use the following notations to represent shares of real/dummy blocks.

- Let $[\![\mathsf{d}]\!] = ([\![k]\!], [\![v]\!])$ be a share of a data block that contains shares of a key k and value v.
- Let $[\![\mathsf{d}^{\mathsf{dummy}}]\!] = ([\![\perp]\!], [\![\perp]\!])$ be a share of a dummy block. We assume that $\perp$ is a number greater than any real k.
- Let $[\![\mathsf{D}]\!] = ([\![\mathsf{d}_0]\!], \ldots, [\![\mathsf{d}_{n-1}]\!])$ be a share of a dataset of size n.

Algorithm 7 constructs 2-out-of-2 shares of a Cuckoo hash table, $(\langle \mathsf{T} \rangle, \langle \mathsf{S} \rangle)$, from 2-out-of-3 shares of a dataset $[\![\mathsf{D}]\!]$. In this protocol, the party assigned the role of *permuter* obtains all addresses the data should be placed and then computes a permutation π that moves the data to (one of) the corresponding addresses. Now, parties can efficiently convert D into the table (T, S) via $\mathcal{F}_{\text{PERM}}$.

Algorithm 6 $[\![\mathsf{D}]\!]_0, [\![\mathsf{D}]\!]_1, [\![\mathsf{D}]\!]_2 \leftarrow \Pi_{\text{Extract}}(\pi, \langle \mathsf{T} \rangle_{0,1}, [\![\mathsf{S}]\!])$

Notation: Let P be the permuter and let S_0, S_1 be the storages.

Require: S_0 and S_1 have the distributed hash table $\langle \mathsf{T} \rangle_{0,1}$, respectively. P has the permutation π and all parties have the distributed stash $[\![\mathsf{S}]\!]$.

Ensure: A dataset D contains all real elements in T and S.

1: Each S_i converts $[\![\mathsf{S}]\!]$ to $\langle \mathsf{S} \rangle_i$ and reorganizes an array $\langle \widetilde{\mathsf{D}} \rangle_i$ as $\widetilde{\mathsf{D}} = \mathsf{T} \| \mathsf{S}$.
2: Parties call $\mathcal{F}_{\text{Unperm}}$ with $(\pi, \langle \widetilde{\mathsf{D}} \rangle_0, \langle \widetilde{\mathsf{D}} \rangle_1)$ to obtain $[\![\widetilde{\mathsf{D}'}]\!]_{j \in \{0,1,2\}}$ where $\widetilde{\mathsf{D}'} = \pi^{-1} \cdot \widetilde{\mathsf{D}}$.
3: Let $[\![\mathsf{D}]\!]_j$ be the first n elements of $[\![\widetilde{\mathsf{D}'}]\!]_i$.
4: **Return** $[\![\mathsf{D}]\!]_0, [\![\mathsf{D}]\!]_1, [\![\mathsf{D}]\!]_2$

Algorithm 7 $\pi, \langle \mathsf{T} \rangle_{0,1}, [\![\mathsf{S}]\!], [\![s_0]\!], [\![s_1]\!], \mathsf{ctr}, \mathbf{P} \leftarrow \Pi_{\text{Build}}([\![\mathsf{D}]\!]_0, [\![\mathsf{D}]\!]_1, [\![\mathsf{D}]\!]_2)$

Notation: Let P be the permuter and S_0, S_1 be the storages. Let $\tau := |\mathsf{T}|$ and $\sigma := |\mathsf{S}|$ be the size of expected hash table and stash (T, S) storing n items, and let $n' = \tau + \sigma$.

Require: Parties have $(1,3)$-shares of a dataset $[\![\mathsf{D}]\!]_{i \in \{0,1,2\}} = ([\![\mathsf{d}_0]\!]_i, \ldots, [\![\mathsf{d}_{n-1}]\!]_i)$.

Ensure: S_0 and S_1 obtain $(1,2)$-shares of Cuckoo hash table, $\langle \mathsf{T} \rangle_{0,1}$, respectively. P obtains a permutation π, and all parties hold $(1,3)$-shares of a stash $[\![\mathsf{S}]\!]$ and PRF keys $[\![s_0]\!]$ and $[\![s_1]\!]$. Parties also hold a query counter ctr, and S_0, S_1 hold a set $\mathbf{P}$, as their states.

(Computing pseudorandom addresses for the input data.)

1: Parties call $\mathcal{F}_{\text{Rnd}}$ to generate random PRF keys $[\![s_0]\!]$ and $[\![s_1]\!]$.
2: **for all** $i \in [n]$ **do**
3: Parties call $\mathcal{F}_{\text{Eq}}$ to obtain $[\![\mathsf{isDummy}_i]\!] := [\![k_i =_? \perp]\!]$.
4: Parties call $\mathcal{F}_{\text{IfElse}}$ to simulate:
If $\mathsf{isDummy}_i = 1$, *then* $[\![\widetilde{k_i}]\!] := [\![\perp + i]\!]$; *otherwise* $[\![\widetilde{k_i}]\!] := [\![k_i]\!]$.
5: Parties call $\mathcal{F}_{\text{PRF}}$ to obtain virtual addresses $[\![\mathsf{addr}_{i,0}]\!]$ and $[\![\mathsf{addr}_{i,1}]\!]$ from $([\![s_0]\!], [\![\widetilde{k_i}]\!])$ and $([\![s_1]\!], [\![\widetilde{k_i}]\!])$, respectively.

(Building a Cuckoo hash table via permutation.)

6: Parties call $\mathcal{F}_{\text{Share}}$ to generate an array $[\![\mathsf{E}]\!]$ consisting of $n' - n$ dummy blocks.
7: Let $[\![\widetilde{\mathsf{D}}]\!] := [\![\mathsf{D}]\!] \,\|\, [\![\mathsf{E}]\!]$ be a concatenated dataset.
8: Parties call $\mathcal{F}_{\text{Reveal}}^{\{P\}}$ to reveal to P all $\mathsf{addr}_{i,0}, \mathsf{addr}_{i,1}$ for $i \in [n]$.
9: P computes a permutation $\pi \colon [n'] \to [n']$ that indicates the bin-placements of Cuckoo hashing. That is, π says where to place $\widetilde{\mathsf{D}}$'s elements as indicated by either $\mathsf{addr}_{i,0}$ or $\mathsf{addr}_{i,1}$.
10: Parties call $\mathcal{F}_{\text{Perm}}$ with π and $[\![\widetilde{\mathsf{D}}]\!]$ to make S_0 and S_1 obtain $\langle \widetilde{\mathsf{D}'} \rangle_0$ and $\langle \widetilde{\mathsf{D}'} \rangle_1$ respectively, where $\widetilde{\mathsf{D}'} = \pi \cdot \widetilde{\mathsf{D}}$.
11: Each S_i for $i \in \{0,1\}$ organizes the array $\langle \widetilde{\mathsf{D}'} \rangle_i$ into a hash table $\langle \mathsf{T} \rangle_i$ and stash $\langle \mathsf{S} \rangle_i$, by separating $\langle \widetilde{\mathsf{D}'} \rangle_i$ into the first τ and the last σ elements.
12: Parties call $\mathcal{F}_{\text{Reshare}}$ to convert $\langle \mathsf{S} \rangle_{0,1}$ to $[\![\mathsf{S}]\!]$.
13: Parties set their state $\mathsf{ctr} = 0$, and S_0, S_1 allocate an empty set $\mathbf{P} = \emptyset$.
14: **Return** $\pi, \langle \mathsf{T} \rangle_{0,1}, [\![\mathsf{S}]\!], [\![s_0]\!], [\![s_1]\!], \mathsf{ctr}, \mathbf{P}$

Algorithm 8 fetches a queried item from the Cuckoo hash table. The main part of this protocol is that the parties assigned the role of *storage* obtain two addresses of the queried item, access T of the location indicated by them, and select one out of the two items of T. When the parties receive a dummy query, random locations of T are fetched.

Algorithm 9 is a deconstruction procedure that applies the inverted permutation π^{-1} to the hash table. This π^{-1} sorts all real blocks in the hash table into the order in which they were input to Π_{Build}. The stash lookup procedure is the same as Algorithms 5 and so we reuse the code and avoid repetition.

Lemma 9. *Assume that the input array consists of $n > \log^2 N$ blocks and each block is b bits. There exists a distributed 3-party protocol, described as Algorithms 5, 7, 8 and 9, that securely realizes $\mathcal{F}_{\mathsf{HT}}$ in the $(\mathcal{F}_{\text{Share}}, \mathcal{F}_{\text{Reveal}}^{\mathcal{P}}, \mathcal{F}_{\text{ABB}}, \mathcal{F}_{\text{Perm}}, \mathcal{F}_{\text{Unperm}})$-hybrid model and in the presence of a*

Algorithm 8 $[\![\mathsf{d}]\!], [\![\mathsf{found}]\!] \leftarrow \Pi_{\text{LOOKUP}}([\![k]\!], \langle \mathsf{T} \rangle_{0,1}, [\![s_0]\!], [\![s_1]\!], \mathsf{ctr}, \mathbf{P})$

Notation: Let P be the permuter and let S_0, S_1 be the storages.

Require: $[\![k]\!]$ is a $(1,3)$-share of an input key to be searched for. S_0, S_1 have the distributed hash table $\langle \mathsf{T} \rangle_{0,1}$ and a set $\mathbf{P}$. Parties have $(1,3)$-shares of PRF keys $[\![s_0]\!], [\![s_1]\!]$ and a query counter ctr.

Ensure: $\mathsf{d} = (k, v), \mathsf{found} = 1$ if T contains (k, v). Otherwise, $\mathsf{d} = (0, 0), \mathsf{found} = 0$.

(Computing pseudorandom addresses to be fetched.)

1: Parties call $\mathcal{F}_{\text{EQ}}$ to obtain $[\![\mathsf{isDummy}]\!] := [\![k =_? \perp]\!]$.

2: Parties call $\mathcal{F}_{\text{IFELSE}}$ to simulate:
If $\mathsf{isDummy} = 1$, *then set* $[\![\widetilde{k}]\!] := [\![\perp + \mathsf{ctr}]\!]$; *otherwise* $[\![\widetilde{k}]\!] := [\![k]\!]$.

3: Parties call $\mathcal{F}_{\text{PRF}}$ to obtain $[\![\mathsf{addr}_0]\!]$ and $[\![\mathsf{addr}_1]\!]$ from $([\![s_0]\!], [\![\widetilde{k}]\!])$ and $([\![s_1]\!], [\![\widetilde{k}]\!])$, respectively.

4: Parties call $\mathcal{F}_{\text{REVEAL}}^{\{S_0,S_1\}}$ to recover $\mathsf{addr}_0, \mathsf{addr}_1$ to both S_0 and S_1. If $(\mathsf{addr}_0, \mathsf{addr}_1) \in \mathbf{P}$, S_0 and S_1 halt. Otherwise, they update $\mathbf{P} \leftarrow \mathbf{P} \cup \{(\mathsf{addr}_0, \mathsf{addr}_1)\}$.

(Searching for the table T.)

5: **for all** $i = 0, 1$ **do**

6: Parties call $\mathcal{F}_{\text{RESHARE}}$ to convert $\langle \mathsf{T}[\mathsf{addr}_i] \rangle$ to $([\![k_i]\!], [\![v_i]\!])$.

7: Parties call $\mathcal{F}_{\text{EQ}}$ to obtain $[\![\mathsf{isQueried}_i]\!] = [\![k_i =_? k]\!]$.

8: Parties call $\mathcal{F}_{\text{IFELSE}}$ and $\mathcal{F}_{\text{MULT}}$ to simulate:
If $\mathsf{isDummy} = 0$ *and* $\mathsf{isQueried}_i = 1$,
then $\mathsf{found} = 1$, $[\![\mathsf{d}]\!] = ([\![k_i]\!], [\![v_i]\!])$, *and* $\langle \mathsf{T}[\mathsf{addr}_i] \rangle = \langle \mathsf{d}^{\mathsf{dummy}} \rangle$.

9: Parties increment ctr.

10: **Return** $[\![\mathsf{d}]\!], [\![\mathsf{found}]\!]$

Algorithm 9 $[\![\mathsf{D}]\!]_0, [\![\mathsf{D}]\!]_1, [\![\mathsf{D}]\!]_2 \leftarrow \Pi_{\text{EXTRACT}}(\pi, \langle \mathsf{T} \rangle_{0,1}, [\![\mathsf{S}]\!])$

Notation: Let P be the permuter and let S_0, S_1 be the storages.

Require: S_0 and S_1 have the distributed hash table $\langle \mathsf{T} \rangle_{0,1}$, respectively. P has the permutation π and all parties have the distributed stash $[\![\mathsf{S}]\!]$.

Ensure: A dataset D contains all real elements in T and S.

1: Each S_i converts $[\![\mathsf{S}]\!]$ to $\langle \mathsf{S} \rangle_i$ and reorganizes an array $\langle \widetilde{\mathsf{D}} \rangle_i$ as $\widetilde{\mathsf{D}} = \mathsf{T} \| \mathsf{S}$.

2: Parties call $\mathcal{F}_{\text{UNPERM}}$ with $(\pi, \langle \widetilde{\mathsf{D}} \rangle_0, \langle \widetilde{\mathsf{D}} \rangle_1)$ to obtain $[\![\widetilde{\mathsf{D}'}]\!]_{j \in \{0,1,2\}}$ where $\widetilde{\mathsf{D}'} = \pi^{-1} \cdot \widetilde{\mathsf{D}}$.

3: Let $[\![\mathsf{D}]\!]_j$ be the first n elements of $[\![\widetilde{\mathsf{D}'}]\!]_i$.

4: **Return** $[\![\mathsf{D}]\!]_0, [\![\mathsf{D}]\!]_1, [\![\mathsf{D}]\!]_2$

passive adversary that controls one party.

- *The* BUILD *procedure consumes* $O(n(\lambda + b + \log n))$ *local computation steps.*
- *The* LOOKUP *procedure consumes* $O(\lambda + b)$ *local computation steps.*
- *The* STASHSEARCH *procedure consumes* $O(b \log N)$ *local computation steps.*
- *The* EXTRACT *procedure consumes* $O(nb)$ *local computation steps.*

Furthermore, all those procedures consumes $O(1)$ *communication rounds.*

Proof. The proof of security and correctness is given in Appendix C.1. We focus on the efficiency analysis below.

Algorithm 7 consists of $O(n)$ calls of $\mathcal{F}_{\text{EQ}}, \mathcal{F}_{\text{IFELSE}}, \mathcal{F}_{\text{PRF}}, \mathcal{F}_{\text{SHARE}}$, and $\mathcal{F}_{\text{REVEAL}}$, and an invocation of $\mathcal{F}_{\text{PERM}}$. In Algorithm 8, the parties need to call $\mathcal{F}_{\text{MULT}}, \mathcal{F}_{\text{RESHARE}}$, $\mathcal{F}_{\text{EQ}}, \mathcal{F}_{\text{IFELSE}}, \mathcal{F}_{\text{REVEAL}}$, and $\mathcal{F}_{\text{PRF}}$ $O(1)$ times for the table lookup. Algorithm 9 requires $\mathcal{F}_{\text{UNPERM}}$ at once. In addition, all iterative operations in Algorithm 7, 8, and 9 can be performed in parallel. Lastly, the cost of STASHSEARCH was analyzed in Lemma 8. □

Chapter 4

Optimal DORAM Against Passive Adversaries

In this chapter, we give our optimal 3-party DORAM that is secure against a passive adversary who colludes with one of the three servers.

Our DORAM is on the known hierarchical paradigm, i.e. the data structure is built via a hierarchy of $L := \lceil \log N - \log\log N \rceil$ distributed hash tables and one top-level array. All levels $i = 1, \ldots, L$ in the hierarchy are implemented using our distributed oblivious hash table $\mathcal{F}_{\mathsf{HT}}$ from Sections 3.3 and 3.4. The size of the stash in our distributed oblivious hash table is set to $\sigma = \lceil \log N \rceil$. The top-level array, $[\![\mathsf{S}]\!]$, can store up to $c = 2\sigma$ data blocks. The capacity of level $i \in [L]$ is $c2^{i-1}$. Each data block may be associated with metadata that is used to keep track of the location of an element. Specifically, an "augmented data block" $[\![\widetilde{\mathsf{d}}_i]\!] := ([\![\mathsf{d}_i]\!], [\![\mathsf{lv}_i]\!])$ consists of the main data block $[\![\mathsf{d}_i]\!] = ([\![k_i]\!], [\![v_i]\!])$ and additional information $[\![\mathsf{lv}_i]\!]$; $\mathsf{lv}_i \in \{0,1\}^{L-1}$ that indicates the levels to which $[\![\mathsf{d}_i]\!]$ is associated.[1]

The smaller tables of level $i = 1$ to $2\log\log N$, i.e., tables that hold up to $\log^2 N$ elements, are implemented with our hash table for short inputs (Section 3.3). For them, each instance consists of a single table, split into bins and a stash. Recall that we merge all stashes of all levels together into one logarithmic-size stash. Lookup goes into one of the bins and scans it. For the tables holding longer arrays, from $i = 2\log\log N + 1$ to L, the hash table is implemented with our hash table for long inputs (Section 3.4). Each such hash table consists of two parts: the main table and a stash. Again, the stashes from all levels are combined into the top-level array, as commonly done (e.g., in [49]).

Theorem 10. *There is a 3-party protocol, described as Algorithm 10 and 11, that securely realizes $\mathcal{F}_{\mathrm{RAM}}$ in the $(\mathcal{F}_{\mathrm{ABB}}, \mathcal{F}_{\mathsf{HT}})$-hybrid model and in the presence of a passive adversary that controls one party. The construction costs $O(\log N)$ amortized computational overhead and $O(\log N)$ communication rounds with $\lambda + b$ block size, where $b = \Omega(\log N)$.*

Proof. The proof of security is given in Appendix C.2.Correctness is clear from the algorithms since we are in the $\mathcal{F}_{\mathsf{HT}}$-hybrid model,. Indeed, a lookup is performed through the whole hierarchy and when an element is found, it is re-inserted into the hierarchy. Hence, we focus on efficiency analysis next.

Algorthm 10 requires one call of $\Pi_{\textsc{StashSearch}}$ with input size c, $O(L)$ calls of $\mathcal{F}_{\textsc{Bitext}}, \mathcal{F}_{\textsc{IfElse}}$, and $\mathcal{F}_{\textsc{Mult}}$, $O(\log\log N)$ calls to of $\mathsf{H}.\textsc{Lookup}$ of the balls-and-bins-based scheme, calls $O(L)$ calls of

[1] The metadata associated with each data blocks is used to avoid the stash-resampling attack of [25], same as was done in [4, 5, 25].

[2] Note that if $p \leq \log\log N$, the obtained PRF key is single, $[\![s_p]\!]$.

Algorithm 10 $[\![\mathsf{d}]\!] \leftarrow \Pi_{\text{Access}}([\![\mathsf{op}]\!], [\![k]\!], [\![v']\!])$

Require: The input contains shares of an operation $[\![\mathsf{op}]\!]$; $\mathsf{op} \in \{\mathsf{read}, \mathsf{write}\}$, key $[\![k]\!]$, and value $[\![v']\!]$.
Ensure: $\mathsf{d} = (k, v)$ if the ORAM holds (k, v), or $\mathsf{d} = (k, v')$ if $\mathsf{op} = \mathsf{write}$. Otherwise, $\mathsf{d} = (\perp, \perp)$.
(Searching for the top-level array.)
1: Parties run $\Pi_{\text{StashSearch}}([\![k]\!], [\![\mathsf{S}]\!])$ for the top-level array $[\![\mathsf{S}]\!]$ to obtain $[\![\widetilde{\mathsf{d}}]\!], [\![\mathsf{found}]\!]$.
(Searching for hash tables in the hierarchy.)
2: **for** $\ell = 1$ to L **do**
3: Parties call $\mathcal{F}_{\text{BitExt}}$ to obtain $[\![\mathsf{lv}_\ell]\!]$ where lv_ℓ is the ℓ-th bit of lv of $\widetilde{\mathsf{d}}$.
4: Parties call $\mathcal{F}_{\text{IfElse}}$ and $\mathcal{F}_{\text{Mult}}$ to simulate:
If $\mathsf{found} = 1$ *and* $\mathsf{lv}_\ell = 0$ *then* $[\![\widetilde{k}]\!] := [\![\perp]\!]$, *otherwise* $[\![\widetilde{k}]\!] := [\![k]\!]$.
5: Parties call HT_i.Lookup$([\![\widetilde{k}]\!])$ (only in the main tables, ignoring the stash)
to obtain $[\![\widetilde{\mathsf{d}}_i]\!], [\![\mathsf{found}_i]\!]$ with its state
$(\langle \mathsf{T}_i \rangle_0, \langle \mathsf{T}_i \rangle_1, [\![s_i]\!], \mathsf{ctr}_i, \mathbf{P}_i)$ when $1 \le i \le \log\log N$, or
$(\langle \mathsf{T}_i \rangle_0, \langle \mathsf{T}_i \rangle_1, [\![s_{i,0}]\!], [\![s_{i,1}]\!], \mathsf{ctr}_i, \mathbf{P}_i)$ when $\log\log N < i$.
6: Set $[\![\widetilde{\mathsf{d}}]\!] = [\![\widetilde{\mathsf{d}}]\!] + [\![\widetilde{\mathsf{d}}_i]\!]$ and $[\![\mathsf{found}]\!] = [\![\mathsf{found}]\!] + [\![\mathsf{found}_i]\!]$.
(Rewriting (if needed) and re-storing the retrieved data.)
7: Parties call $\mathcal{F}_{\text{IfElse}}$ to simulate: *If* $\mathsf{found} = 0$ *then set* $[\![\mathsf{d}]\!] = [\![\mathsf{d}^{\mathsf{dummy}}]\!]$.
8: Parties call $\mathcal{F}_{\text{IfElse}}$ to simulate: *If* $\mathsf{op} = \mathsf{write}$ *then set* $[\![\mathsf{d}]\!] = ([\![k]\!], [\![v']\!])$.
9: Parties set $[\![\widetilde{\mathsf{d}}]\!] = ([\![\mathsf{d}]\!], [\![0]\!])$ and concatenate it into the end of the top-level array.
10: If the size of the top-level array is c, parties run $\Pi_{\text{Reshuffle}}()$ to refresh the hierarchy.
11: **Return** $[\![\mathsf{d}]\!]$

H.Lookup (without stash) for the Cuckoo-hash based scheme, and an invocation of $\Pi_{\text{Reshuffle}}$, for $c = L = O(\log N)$. For any block size $b = \Omega(\log N)$ bits, this procedure requires $O((\lambda+b)\log N)+\mathcal{C}$ computational steps where $\mathcal{C}$ is the amortized cost of $\Pi_{\text{Reshuffle}}$. Algorithm 11 costs H_i.Extract for all $i = 1, \ldots, p$, H_p.Build at once, and $O(c2^{p-1})$ calls of $\mathcal{F}_{\text{Trunc}}$, per $c2^{p-1}$ access. Hence, its amortized cost can be estimated as $\mathcal{C}_{\text{Reshuffle}} = \sum_{p=1}^{L} \frac{O(c2^p(\lambda+b))}{c2^{p-1}}$. □

Concrete Efficiency. As we mentioned in Section 1.1, the overhead claimed above depends on a small hidden constant. Specifically, our Π_{Access} consists of at most $4\log N$ (amortized) calls of $\mathcal{F}_{\text{PRF}}$ per access, which is 25 times smaller than the known optimal DORAM [49]. For a more detailed analysis, see Appendix A.

Algorithm 11 $\Pi_{\text{Reshuffle}}()$

Require: p is the level s.t. all $\mathsf{HT}_{i<p}$ in the hierarchy is full and HT_p is not.
Ensure: All $\mathsf{HT}_{i<p}$ in the hierarchy become empty, and HT_p becomes full.
(Extracting all the data that needs to be reshuffled.)
1: Allocate an array $[\![\mathbf{A}]\!]$ of sufficient capacity and insert all elements of $[\![\mathsf{S}]\!]$ to $[\![\mathbf{A}]\!]$.
2: For all $i = 1, \ldots, p-1$, parties call HT_i.Extract() with its state $(\pi_i, \langle \mathsf{T}_i \rangle_0, \langle \mathsf{T}_i \rangle_1)$ to extract all real elements as $[\![\mathsf{D}_i]\!]$ and combine them to $[\![\mathbf{A}]\!]$ as $\mathbf{A} = \mathbf{A} \parallel \mathsf{D}_i$.
3: For all elements $[\![\widetilde{\mathsf{d}}_j]\!] = ([\![\mathsf{d}_j]\!], [\![\mathsf{lv}_j]\!])$ of $[\![\mathbf{A}]\!]$, parties call $\mathcal{F}_{\text{Trunc}}$ to set the p first indices of lv_j to 0.
(Building a new hash table.)
4: Parties call HT_p.Build$([\![\mathbf{A}]\!])$ to construct a distributed hash table $\big(\pi_p, \langle \mathsf{T}_p \rangle_{0,1}, [\![\mathsf{S}_p]\!], [\![s_{p,0}]\!], [\![s_{p,1}]\!], \mathsf{ctr}, \mathbf{P}\big)$.[2]
5: Parties set $[\![\mathsf{S}]\!]$ as a new top-level array, and for all $[\![\widetilde{\mathsf{d}}_u]\!] = ([\![\mathsf{d}_u]\!], [\![\mathsf{lv}_u]\!])$ of $[\![\mathsf{S}]\!]$, call $\mathcal{F}_{\text{EQ}}$ and $\mathcal{F}_{\text{IfElse}}$ to simulate:
If $k_i \neq \perp$, *set* $[\![\mathsf{lv}_u]\!] = [\![\mathsf{lv}_u]\!] + [\![1]\!]2^{p-1}$.

Chapter 5

Security Extension Against Active Adversaries

We extend our oblivious hashing and DORAM to be secure against an adversary that can deviate from the prescribed protocols. Though it is *almost* feasible by a generic framework of actively secure MPC, we require a new permutation (Section 5.1) and *verifying permutations* protocol (Section 5.2) in addition.

By known frameworks of actively secure MPC with abort [16, 36, 41], we assume that our protocols are in the flow of the following three phases:

- **Randomization phase.** For any share $[\![a]\!]$, parties compute $[\![ra]\!]$ with random secret r and store $([\![a]\!], [\![ra]\!])$ as the share of *a with a MAC*. This r serves as a blinding factor and is unknown to any party.

- **Computation phase.** Parties compute a target function F on input $([\![a]\!], [\![ra]\!])$ while recording checksums. For example, let $F = f_0 \circ f_1$ and assume Π_{f_0}, Π_{f_1} that are protocols used to compute $([\![f_{\{0,1\}}(a)]\!], [\![rf_{\{0,1\}}(a)]\!])$ from $([\![a]\!], [\![ra]\!])$ and are futher secure up to additive attacks [29, 28]. Now, the parties allocate a set of checksums $\mathcal{S} = \emptyset$ and perform Π_{f_2} and Π_{f_1} in sequence to obtain $[\![F(a)]\!]$ while storing all inputs and outputs of the protocols, i.e., $([\![a]\!], [\![ra]\!])$, $([\![f_2(a)]\!], [\![rf_2(a)]\!])$ and $([\![F(a)]\!], [\![rF(a)]\!])$, into $\mathcal{S}$.

- **Proof phase.** To detect cheating, parties evaluate the shares and their MACs recorded as checksums. Following the above example, the parties generate new random shares $[\![\rho_0]\!], [\![\rho_1]\!]$ and $[\![\rho_2]\!]$, and compute inner products
$$[\![\phi]\!] = ([\![\rho_0]\!] \times [\![a]\!] + [\![\rho_1]\!] \times [\![f_2(a)]\!] + [\![\rho_2]\!] \times [\![F(a)]\!]) \text{ and}$$
$$[\![r\phi]\!] = ([\![\rho_0]\!] \times [\![ra]\!] + [\![\rho_1]\!] \times [\![rf_2(a)]\!] + [\![\rho_2]\!] \times [\![rF(a)]\!]).$$
The parties recover $[\![\eta]\!] = [\![r]\!] \times [\![\phi]\!] - [\![r\phi]\!]$, and if $\eta \neq 0$ then they abort.

For our DORAM, we should be more concerned about the form of shares than in the passive security model. Since part of building blocks in Section 2.6, e.g., $\mathcal{F}_{\text{PRF}}$, requires bit-wise operations, we should assign MACs to bits $a_{b-1}, \ldots, a_0 \in \mathbb{Z}_2$, instead of the whole $a \in \mathbb{Z}_{2^b}$. According to Kikuchi et al. [41], we can provide a MAC for a bit $u \in \{0, 1\}$ as $[\![ru]\!] := ([\![r_{\kappa-1}u]\!], \ldots, [\![r_0u]\!])$ where each r_j is in $\mathbb{Z}_2$. To detect cheating with overwhelming probability $1 - \mathsf{negl}(N)$, this κ should be $\omega(\log N)$. Hence, by encoding $([\![a]\!], [\![ra]\!])$ as $\big(([\![a_{b-1}]\!], \ldots, [\![a_0]\!]), ([\![ra_{b-1}]\!], \ldots, [\![ra_{b-1}]\!])\big)$, the $\mathcal{F}_{\text{ABB}}$ functionality described in Section 2.6 is also available in active security model.

To simplify the notation, we denote $([\![a]\!], [\![ra]\!])$ as $[\![a]\!]^{\mathsf{m}}$ in the following.

5.1 Secure Oblivious Permutation Up To Additive Attacks

We start with constructing a secure permutation protocol up to additive attacks. In contrast to Section 3.2, we assume that a permutation π provided by one party has been already separated into π_0 and π_1 s.t. $\pi = \pi_1 \circ \pi_0$ and shared between parties. We discuss verification for the permutation π in Section 5.2, and here we focus on cheating on an array that should be permuted.

Our permutation protocol described in Algorithm 1 relies on (semi-)honest parties and is difficult to convert to be secure up to additive attacks. Instead, we construct a permutation protocol using a reshare-based shuffling as in Ikarashi et al. [36]. Let $\mathcal{F}_{\textsc{Shuffle}}$ be functionality for secure shuffling up to an additive attack s.t. it receives shares of an array $[\![\mathbf{I}]\!]$ and a permutation π from honest parties and an array Δ of the same size as $\mathbf{I}$ from an adversary, then it returns $[\![\pi \cdot \mathbf{I} + \Delta]\!]$. Now, the following protocol, originally proposed by Laur et al. [48], is the implementation of $\mathcal{F}_{\textsc{Shuffle}}$ that costs n calls of $\mathcal{F}_{\textsc{Reshare}}$.

$[\![\mathbf{I}']\!]_0, [\![\mathbf{I}']\!]_1, [\![\mathbf{I}']\!]_2 \leftarrow \Pi_{\textsc{Shuffle}}((\pi, [\![\mathbf{I}]\!]_0), (\pi, [\![\mathbf{I}]\!]_1), [\![\mathbf{I}]\!]_2)$:

1. P_0 and P_1 convert their $[\![\mathbf{I}]\!]_{i\in\{0,1\}}$ to $\langle\mathbf{I}\rangle_i$ and compute $\langle\mathbf{I}'\rangle_i = \pi \cdot \langle\mathbf{I}\rangle_i$ each.
2. Parties call $\mathcal{F}_{\textsc{Reshare}}$ to obtain $[\![\mathbf{I}']\!]_{\{0,1,2\}}$ from $\langle\mathbf{I}'\rangle_{\{0,1\}}$.
3. Output $[\![\mathbf{I}']\!]_{\{0,1,2\}}$.

Algorithm 12 $[\![\mathbf{I}']\!]^{\mathsf{m}}_0, [\![\mathbf{I}']\!]^{\mathsf{m}}_1, [\![\mathbf{I}']\!]^{\mathsf{m}}_2 \leftarrow \Pi^{\text{active}}_{\textsc{Perm}}((\pi_0, \pi_1), [\![\mathbf{I}]\!]^{\mathsf{m}}_0, [\![\mathbf{I}]\!]^{\mathsf{m}}_1, [\![\mathbf{I}]\!]^{\mathsf{m}}_2)$

Notation: Let P be the permuter and let S_0, S_1 be the storages. Let $\mathcal{S}$ be a set of checksums.
Require: P has both π_0, π_1 and $[\![\mathbf{I}]\!]^{\mathsf{m}}_0$. Each $S_{i=0,1}$ has π_i and $[\![\mathbf{I}]\!]^{\mathsf{m}}_{i+1}$.
Ensure: $\mathbf{I}' = \pi \cdot \mathbf{I}$ and $r\mathbf{I}' = \pi \cdot (r\mathbf{I})$ where $\pi = \pi_1 \circ \pi_0$.

1: Parties record their input shares into $\mathcal{S}$ as $\mathcal{S} = \mathcal{S} \cup \{[\![\mathbf{I}]\!]^{\mathsf{m}}\}$, and let $[\![\mathbf{I}'_0]\!]^{\mathsf{m}} = [\![\mathbf{I}]\!]^{\mathsf{m}}$.
2: **for** $i = 0, 1$ **do**
3: Parties call $\mathcal{F}_{\textsc{Shuffle}}$ with inputs π_i and $[\![\mathbf{I}'_i]\!]^{\mathsf{m}}$ to obtain $[\![\mathbf{I}'_{i+1}]\!]^{\mathsf{m}} = [\![\pi_i \cdot \mathbf{I}'_i]\!]^{\mathsf{m}}$.
4: Parties record their shares into $\mathcal{S}$ as $\mathcal{S} \leftarrow \mathcal{S} \cup \{[\![\mathbf{I}'_{i+1}]\!]^{\mathsf{m}}\}$.
5: Parties store $[\![\mathbf{I}'_2]\!]^{\mathsf{m}}_0, [\![\mathbf{I}'_2]\!]^{\mathsf{m}}_1, [\![\mathbf{I}'_2]\!]^{\mathsf{m}}_2$ as $[\![\mathbf{I}']\!]^{\mathsf{m}}_0, [\![\mathbf{I}']\!]^{\mathsf{m}}_1, [\![\mathbf{I}']\!]^{\mathsf{m}}_2$, respectively.
6: **Return** $[\![\mathbf{I}']\!]^{\mathsf{m}}_{\{0,1,2\}}$

In Algorithm 12, we describe an actively secure permutation protocol for our distributed oblivious hashing in the $\mathcal{F}_{\textsc{Shuffle}}$-hybrid model. An actively secure unpermutation protocol, $\Pi^{\text{active}}_{\textsc{Unperm}}$, is achieved from $\Pi^{\text{active}}_{\textsc{Perm}}$ straightforwardly, i.e., parties first compute $[\![\mathbf{I}']\!]^{\mathsf{m}} = [\![\pi_1^{-1} \cdot \mathbf{I}]\!]^{\mathsf{m}}$, then obtain $[\![\mathbf{I}'']\!]^{\mathsf{m}} = [\![\pi_0^{-1} \cdot \mathbf{I}']\!]^{\mathsf{m}}$.

5.2 Distributed Oblivious Hashing Against Active Adversaries

Even though an actively secure oblivious hashing can be obtained *almost* completely straightforwardly by replacing all functionalities used in our distributed oblivious hashing (Section 3.3 and 3.4) by actively secure ones, it is still *not secure against a corrupted permuter* that can input an invalid permutation. We thus focus on solving this problem. We can say the permutation π is *valid* if all keys are always found in their place, *whatever π actually is*. This means that the pseudorandom addresses can work as *witnesses* for the correctness of the hash table, i.e., we can verify π by checking whether

(*one of two pseudorandom addresses computed from a non-dummy key*)
$-$ (*the actual address where the element resides*)

is equal to 0 for all real keys (in Cuckoo hashing). This verification can be done in the BUILD procedure as follows: For all real blocks $[\![\mathsf{d}_i]\!]$ located in $\mathsf{T}[\mathsf{addr}^{\mathrm{r}}_i]$ and assigned to the pseudorandom addresses $([\![\mathsf{addr}^{\mathrm{p}}_{i,0}]\!], [\![\mathsf{addr}^{\mathrm{p}}_{i,1}]\!])$, parties check the following equation with uniformly random ρ_i.

$$0 =_? \sum_i [\![\rho_i]\!] \times ([\![\mathsf{addr}^{\mathrm{p}}_{i,0}]\!] - \mathsf{addr}^{\mathrm{r}}_i) \times ([\![\mathsf{addr}^{\mathrm{p}}_{i,1}]\!] - \mathsf{addr}^{\mathrm{r}}_i). \tag{5.1}$$

In addition, for efficiency, we combine the above verification with the MAC verification. Remember that, in the Proof phase for MACs, fresh random shares $[\![\rho_i]\!]$ are given for each checksum $([\![a]\!], [\![ra]\!])$. Noticing the similarity in the use of the random ρ_i, for any r, we can transform Equation (5.1) as below:

$$[\![r]\!] \times \sum_i [\![\rho_i]\!] \times ([\![\mathsf{addr}^{\mathrm{p}}_{i,0}]\!] \times [\![\mathsf{addr}^{\mathrm{p}}_{i,1}]\!] + (\mathsf{addr}^{\mathrm{r}}_i)^2) =_?$$
$$\sum_i [\![\rho_i]\!] \times ([\![r \times \mathsf{addr}^{\mathrm{p}}_{i,0}]\!] + [\![r \times \mathsf{addr}^{\mathrm{p}}_{i,1}]\!]) \times \mathsf{addr}^{\mathrm{r}}_i. \tag{5.2}$$

Now, to make parties evaluate the above equation in the Proof phase, we propose the following new protocol that checks the consistency between the virtual address and the actual address of each non-dummy block.

$\underline{\Pi_{\mathrm{VERPERM2}}([\![\mathsf{addr}^{\mathrm{p}}_{i,0}]\!]^{\mathsf{m}}, [\![\mathsf{addr}^{\mathrm{p}}_{i,1}]\!]^{\mathsf{m}}, \mathsf{addr}^{\mathrm{r}}_i)}$:

1. At first, parties call the actively secure $\mathcal{F}_{\mathrm{MULT}}$ to obtain
$$[\![\mathsf{addr}'_i]\!]^{\mathsf{m}} = ([\![\mathsf{addr}'_i]\!], [\![r \times \mathsf{addr}'_i]\!]) = [\![\mathsf{addr}^{\mathrm{p}}_{i,0}]\!]^{\mathsf{m}} \times [\![\mathsf{addr}^{\mathrm{p}}_{i,1}]\!]^{\mathsf{m}}.$$
$[\![\mathsf{addr}^{\mathrm{p}}_{i,0}]\!]^{\mathsf{m}}, [\![\mathsf{addr}^{\mathrm{p}}_{i,1}]\!]^{\mathsf{m}}$ and $[\![\mathsf{addr}'_i]\!]^{\mathsf{m}}$ are recorded in $\mathcal{S}$ as checksums.

2. Parties locally compute below and record $[\![\mathsf{ver}^{(1)}_i]\!]^{\mathsf{m}}$ and $[\![\mathsf{ver}^{(2)}_i]\!]^{\mathsf{m}}$ to $\mathcal{S}$.
$$[\![\mathsf{ver}^{(1)}_i]\!]^{\mathsf{m}} := ([\![\mathsf{addr}'_i]\!] + (\mathsf{addr}^{\mathrm{r}}_i)^2, ([\![r \times \mathsf{addr}^{\mathrm{p}}_{i,0}]\!] + [\![r \times \mathsf{addr}^{\mathrm{p}}_{i,1}]\!]) \times \mathsf{addr}^{\mathrm{r}}_i), \text{ and}$$
$$[\![\mathsf{ver}^{(2)}_i]\!]^{\mathsf{m}} := (([\![\mathsf{addr}^{\mathrm{p}}_{i,0}]\!] + [\![\mathsf{addr}^{\mathrm{p}}_{i,1}]\!]) \times \mathsf{addr}^{\mathrm{r}}_i, [\![r \times \mathsf{addr}'_i]\!] + [\![r]\!] \times (\mathsf{addr}^{\mathrm{r}}_i)^2).$$

Since whether the permuter has provided a valid permutation is equivalent to whether $\mathsf{addr}^{(1)}_i = \mathsf{addr}^{(2)}_i$, the verification for *checksums* $[\![\mathsf{ver}^{(1)}_i]\!]^{\mathsf{m}}$ and $[\![\mathsf{ver}^{(2)}_i]\!]^{\mathsf{m}}$ includes Equation (5.2) and its transformation. Furthermore, since the input and output of $\mathcal{F}_{\mathrm{MULT}}$, $[\![\mathsf{addr}^{\mathrm{p}}_{i,0}]\!]^{\mathsf{m}}, [\![\mathsf{addr}^{\mathrm{p}}_{i,1}]\!]^{\mathsf{m}}$ and $[\![\mathsf{addr}'_i]\!]^{\mathsf{m}}$, are recorded as checksums, we can attribute the attack providing an invalid permutation to an additive attack that modifies $\mathsf{addr}^{\mathrm{r}}_i$ to $\mathsf{addr}^{\mathrm{r}}_i + \delta$ for some δ.

In addition, we can also achieve a simpler permutation verification protocol for our balls-into-bins hashing as follows:

$\underline{\Pi_{\mathrm{VERPERM1}}([\![\mathsf{addr}^{\mathrm{p}}_i]\!]^{\mathsf{m}}, \mathsf{addr}^{\mathrm{r}}_i)}$:

1. Parties locally compute below and record $[\![\mathsf{ver}_i]\!]^{\mathsf{m}}$ to $\mathcal{S}$.
$$[\![\mathsf{ver}_i]\!]^{\mathsf{m}} := ([\![\mathsf{addr}^{\mathrm{p}}_i]\!] - \mathsf{addr}^{\mathrm{r}}_i, [\![r \times \mathsf{addr}^{\mathrm{p}}_i]\!] - [\![r]\!] \times \mathsf{addr}^{\mathrm{r}}_i).$$

Now, we can obtain our actively secure BUILD procedure as Algorithm 13 and 14. Note that new LOOKUP, STASHSEARCH, and EXTRACT procedures can be achieved by natural conversion from the passive MPC to active, thus we skip detailing their pseudocode here.

Lemma 11. *There exists a 3-party distributed balls-into-bins hashing of which the* BUILD *procedure is described in Algorithm 13 and the other procedures are achieved from the actively secure conversion of Algorithm 4, 5, and 6 that securely realizes $\mathcal{F}_{\mathsf{HT}}$ in the $(\mathcal{F}_{\mathrm{SHARE}}, \mathcal{F}^{\mathcal{P}}_{\mathrm{REVEAL}}, \mathcal{F}_{\mathrm{ABB}})$-hybrid model in the presence of an active adversary that controls one party.*

Assume that the input size n and $b' = b \cdot \omega(\log N), \lambda' = \lambda \cdot \omega(\log N)$.

Algorithm 13 $\pi_0, \pi_1, \langle \mathsf{T} \rangle^{\mathsf{m}}_{0,1}, [\![\mathsf{S}]\!]^{\mathsf{m}}, [\![s]\!]^{\mathsf{m}}, \mathsf{ctr}, \mathbf{P} \leftarrow \Pi^{\text{active}}_{\text{BUILD}}([\![\mathsf{D}]\!]^{\mathsf{m}})$

(Constructing a balls-into-bins hash table.)

1: Parties do the same as Algorithm 3 with actively secure components to obtain $\pi_0, \pi_1, \langle \mathsf{T} \rangle^{\mathsf{m}}_{0,1}, [\![\mathsf{S}]\!]^{\mathsf{m}}, [\![s]\!]^{\mathsf{m}}, \mathsf{ctr}, \mathbf{P}$. Note that, in the actively secure variant, once the original construction permutation π is split into π_0, π_1; $\pi = \pi_1 \circ \pi_0$, each π_i should be held by the storage S_i until EXTRACT for the consistency of the permutation.

(Verifying π.)

2: For the virtual addresses $[\![\mathsf{A}]\!]^{\mathsf{m}} = ([\![\mathsf{addr}_0]\!]^{\mathsf{m}} \ldots, [\![\mathsf{addr}_{n-1}]\!]^{\mathsf{m}})$, which are obtained in Line 5 of Algorithm 3, parties set $[\![\widetilde{\mathsf{A}}]\!]^{\mathsf{m}} = [\![\mathsf{A} \parallel \mathsf{E}]\!]^{\mathsf{m}}$ of size n' in the same way as Line 6 and 7 of Algorithm 3. Then, parties run $[\![\widetilde{\mathsf{A}}']\!]^{\mathsf{m}} \leftarrow \Pi^{\text{active}}_{\text{PERM}}(\pi_0, \pi_1, [\![\widetilde{\mathsf{A}}]\!]^{\mathsf{m}})$.

3: **for all** $i \in [\tau]$ **do**

4: Let $(\langle k_i \rangle^{\mathsf{m}}, \langle v_i \rangle^{\mathsf{m}})$ be the i-th element of $\langle \mathsf{T} \rangle^{\mathsf{m}}$.

5: Let $[\![\widetilde{\mathsf{addr}_i}']\!]^{\mathsf{m}}$ be the i-th element of $[\![\widetilde{\mathsf{A}}']\!]^{\mathsf{m}}$.

6: Let $\mathsf{addr}^{\mathrm{r}}_i := \lfloor i/\beta \rfloor$ where β is bin size of T.

7: Parties call $\mathcal{F}_{\text{RESHARE}}, \mathcal{F}_{\text{EQ}}$ and $\mathcal{F}_{\text{IFELSE}}$ to simulate:
If $k_i = \perp$ *then* $[\![\mathsf{addr}^{\mathrm{p}}_i]\!]^{\mathsf{m}} = [\![\mathsf{addr}^{\mathrm{r}}_i]\!]^{\mathsf{m}}$, *otherwise* $[\![\mathsf{addr}^{\mathrm{p}}_i]\!]^{\mathsf{m}} = [\![\widetilde{\mathsf{addr}_i}']\!]^{\mathsf{m}}$.

8: Parties perform $\Pi_{\text{VERPERM1}}([\![\mathsf{addr}^{\mathrm{p}}_i]\!]^{\mathsf{m}}, \mathsf{addr}^{\mathrm{r}}_i)$.

9: **Return** $\pi_0, \pi_1, \langle \mathsf{T} \rangle^{\mathsf{m}}_{0,1}, [\![\mathsf{S}]\!]^{\mathsf{m}}, [\![s]\!]^{\mathsf{m}}, \mathsf{ctr}, \mathbf{P}$

- *The* BUILD *procedure consumes* $O(n(b' + \lambda' + \log n))$ *local computation steps.*
- *The* LOOKUP *procedure consumes* $O((b' + \lambda')\frac{\log N}{\log \log N})$ *local computation steps.*
- *The* STASHSEARCH *procedure consumes* $O(b' \log N)$ *local computation steps.*
- *The* EXTRACT *procedure consumes* $O(nb')$ *local computation steps.*

Lemma 12. *There exists a 3-party distributed Cuckoo hashing of which the* BUILD *procedure is described in Algorithm 14 and the other procedures are achieved from the actively secure conversion of Algorithm 5, 8, and 9 that securely realizes* $\mathcal{F}_{\mathsf{HT}}$ *in the* $(\mathcal{F}_{\text{SHARE}}, \mathcal{F}^{\mathcal{P}}_{\text{REVEAL}}, \mathcal{F}_{\text{ABB}})$*-hybrid model in the presence of an active adversary that controls one party.*

Assume that the input size n *and* $b' = b \cdot \omega(\log N), \lambda' = \lambda \cdot \omega(\log N)$.

- *The* BUILD *procedure consumes* $O(n(b' + \lambda' + \log n))$ *local computation steps.*
- *The* LOOKUP *procedure consumes* $O(b' + \lambda')$ *local computation steps.*
- *The* STASHSEARCH *procedure consumes* $O(b' \log N)$ *local computation steps.*
- *The* EXTRACT *procedure consumes* $O(nb')$ *local computation steps.*

Proof of Lemma 11 and 12. Since each output of Algorithm 13 and 14 is obtained from a general actively secure conversion of Algorithm 3 and 7 respectively, the correctness of them are also given by Lemma 8 and 9.

In addition, the security of the algorithms is also guaranteed from a known actively secure MPC and our additional evaluations $\Pi_{\text{VERPERM1}}, \Pi_{\text{VERPERM2}}$. Though known actively secure MPC frameworks cannot prevent a malicious permuter from tampering with a permutation, Π_{VERPERM1} and Π_{VERPERM2} can detect blocks that are located in invalid places of a hash table since the *checksums* provided in $\Pi_{\text{VERPERM1}}, \Pi_{\text{VERPERM2}}$ cannot be a valid pair of MAC with overwhelming probability when the adversary that does not know the MAC key r tampers with the real addresses of the blocks (remember that the virtual addresses, $\mathsf{addr}^{\mathrm{p}}_i$, are derived from actively secure OPRFs, and the real addresses, $\mathsf{addr}^{\mathrm{r}}_i$, are the result of the permutation).

For the efficiency analysis, we show the following claim.

Algorithm 14 $\pi_0, \pi_1, \langle \mathsf{T} \rangle^{\mathsf{m}}_{0,1}, [\![\mathsf{S}]\!]^{\mathsf{m}}, [\![s_0]\!]^{\mathsf{m}}, [\![s_1]\!]^{\mathsf{m}}, \mathsf{ctr}, \mathbf{P} \leftarrow \Pi^{\text{active}}_{\text{BUILD}}([\![\mathsf{D}]\!]^{\mathsf{m}})$

(Constructing a cuckoo hash table.)

1: Parties do the same as Algorithm 7 with actively secure components to obtain $\pi_0, \pi_1, \langle \mathsf{T} \rangle^{\mathsf{m}}_{0,1}, [\![\mathsf{S}]\!]^{\mathsf{m}}, [\![s_0]\!]^{\mathsf{m}}, [\![s_1]\!]^{\mathsf{m}}, \mathsf{ctr}, \mathbf{P}$.

(Verifying π.)

2: For the virtual addresses $[\![\mathsf{A}]\!]^{\mathsf{m}} = \left(([\![\mathsf{addr}_{i,0}]\!]^{\mathsf{m}}, [\![\mathsf{addr}_{i,1}]\!]^{\mathsf{m}})\right)_{i \in [n]}$, which are obtained in Line 5 of Algorithm 7, parties set $[\![\widetilde{\mathsf{A}}]\!]^{\mathsf{m}} = [\![\mathsf{A} \parallel \mathsf{E}]\!]^{\mathsf{m}}$ of size n' in the same way as Line 6 and 7 of Algorithm 7. Then, parties run $[\![\widetilde{\mathsf{A}}']\!]^{\mathsf{m}} \leftarrow \Pi^{\text{active}}_{\text{PERM}}(\pi_0, \pi_1, [\![\widetilde{\mathsf{A}}]\!]^{\mathsf{m}})$.

3: **for all** $i \in [\tau]$ **do**

4: Let $(\langle k_i \rangle^{\mathsf{m}}, \langle v_i \rangle^{\mathsf{m}})$ be the i-th element of $\langle \mathsf{T} \rangle^{\mathsf{m}}$.

5: Let $([\![\widetilde{\mathsf{addr}}'_{i,0}]\!]^{\mathsf{m}}, [\![\widetilde{\mathsf{addr}}'_{i,1}]\!]^{\mathsf{m}})$ be the i-th element of $[\![\widetilde{\mathsf{A}}']\!]^{\mathsf{m}}$.

6: Let $\mathsf{addr}^{\mathrm{r}}_i := i$.

7: Parties call $\mathcal{F}_{\text{RESHARE}}, \mathcal{F}_{\text{EQ}}$ and $\mathcal{F}_{\text{IFELSE}}$ to simulate:
If $k_i = \perp$ *then* $([\![\mathsf{addr}^{\mathrm{p}}_{i,0}]\!]^{\mathsf{m}}, [\![\mathsf{addr}^{\mathrm{p}}_{i,1}]\!]^{\mathsf{m}}) = ([\![\mathsf{addr}^{\mathrm{r}}_i]\!]^{\mathsf{m}}, [\![\mathsf{addr}^{\mathrm{r}}_i]\!]^{\mathsf{m}})$,
otherwise $([\![\mathsf{addr}^{\mathrm{p}}_{i,0}]\!]^{\mathsf{m}}, [\![\mathsf{addr}^{\mathrm{p}}_{i,1}]\!]^{\mathsf{m}}) = ([\![\widetilde{\mathsf{addr}}'_{i,0}]\!]^{\mathsf{m}}, [\![\widetilde{\mathsf{addr}}'_{i,1}]\!]^{\mathsf{m}})$.

8: Parties perform $\Pi_{\text{VERPERM2}}([\![\mathsf{addr}^{\mathrm{p}}_{i,0}]\!]^{\mathsf{m}}, [\![\mathsf{addr}^{\mathrm{p}}_{i,1}]\!]^{\mathsf{m}}, \mathsf{addr}^{\mathrm{r}}_i)$.

9: **Return** $\pi_0, \pi_1, \langle \mathsf{T} \rangle^{\mathsf{m}}_{0,1}, [\![\mathsf{S}]\!]^{\mathsf{m}}, [\![s_0]\!]^{\mathsf{m}}, [\![s_1]\!]^{\mathsf{m}}, \mathsf{ctr}, \mathbf{P}$

Claim 13. Π_{VERPERM1} *and* Π_{VERPERM2} *each consumes* $O(b')$ *local computation steps.*

This claim is clear from their algorithms, and hence it shows that the additional $O(n)$ calls of Π_{VERPERM1} or Π_{VERPERM2} do not affect to the asymptotic cost of Algorithm 13 or 14, each.

□

5.3 Distributed ORAM Against Active Adversaries

Given our actively secure distributed oblivious hashing, we achieve an actively secure DORAM.

Theorem 14. *There exists a 3-party protocol that securely realizes* $\mathcal{F}_{\text{RAM}}$ *in the presence of an active adversary that controls one party. In addition, this implementation consumes* $O(\log N)$ *amortized overhead and* $O(\log N)$ *communication rounds with* $\omega((\lambda + b) \log N)$ *block size where* $b = \Omega(\log N)$.

Proof. By straightforward composition, we can replace all functionalities related to $\mathcal{F}_{\text{ABB}}$ and $\mathcal{F}_{\mathsf{HT}}$ in Algorithms 10 and 11 with their concrete implementations and thereby get a concrete actively secure DORAM. Security can be showen by considering the same simulator as of Appendix C.2, except that it uses an actively secure distributed oblivious hashing simulator.Moreover, since the efficiency of each $\mathcal{F}_{\text{ABB}}$ and $\mathcal{F}_{\mathsf{HT}}$ is asymptotically the same as in passive security except for the increased block size, the actively secure DORAM achieves $O(\log N)$ overhead using a slightly larger block size $\omega((\lambda + b) \log N)$. □

Chapter 6

Perfectly Secure Oblivious Priority Queue

In this chapter, we propose a novel perfectly-secure OPQ that supports four operations, insertion, extraction of the top-priority element, deletion, and key-modification, and operation hiding.

We remark that, regardless of our interest of *distributed* ORAMs, the OPQ we describe below is available in standard model and aim to not computational but perfect indistinguishability of it's oblivious simulation. Hence, here we show a definition of perfectly oblivious simulation in the standard model, that is slight modification of Definition 4, by using the real world $\mathsf{Real}_{\Pi,\mathcal{A}}(x)$ and the ideal world $\mathsf{Ideal}_{\mathcal{F},\mathsf{Sim},\mathcal{A}}(x)$ where the input string is not distributed to m parties and the adversary's view does not limited by the set C.

Definition 15 (Perfectly oblivious simulation of a reactive functionality)**.** *We say that a stateful protocol Π is a perfectly oblivious implementation of the reactive functionality $\mathcal{F}$ if there exists a stateful simulator Sim, such that for any non-uniform (stateful) adversary $\mathcal{A}$, the view of the adversary $\mathcal{A}$ in the following two experiments $\mathsf{Real}_{\Pi,\mathcal{A}}(x)$ and $\mathsf{Ideal}_{\mathcal{F},\mathsf{Sim},\mathcal{A}}(x)$ is identical:*

$\mathsf{Real}_{\Pi,\mathcal{A}}(x)$*:*
Let $(\mathsf{op}, x) \leftarrow \mathcal{A}()$.
Loop while $\mathsf{op} \neq \perp$*:*
  Let $x' = (\mathsf{op}, x)$.
  $V, y \leftarrow \mathsf{Real}^{\mathsf{nr}}_{\Pi,\mathcal{A}}(x')$.
  $(\mathsf{op}, x) \leftarrow \mathcal{A}(V, y)$.

$\mathsf{Ideal}_{\mathcal{F},\mathsf{Sim},\mathcal{A}}(x)$*:*
Let $(\mathsf{op}, x) \leftarrow \mathcal{A}())$.
Loop while $\mathsf{op} \neq \perp$*:*
  Let $x' = (\mathsf{op}, x)$.
  $V, y \leftarrow \mathsf{Ideal}^{\mathsf{nr}}_{\mathcal{F},\mathsf{Sim},\mathcal{A}}(x')$.
  $(\mathsf{op}, x) \leftarrow \mathcal{A}(V, y)$.

6.1 Our Data Structure: A Hierarchical Tree

Before describing the details of our OPQ, we introduce the components of our data structure below.

- For each value v stored to the OPQ, we call a tuple (k, τ, v) a *block*, where k is a key that indicates the priority of v, and τ denotes the time in which v is inserted. This τ is used to uniquely identify the first inserted one from the tuples of the same key.
- A dummy block is denoted as $(\perp, \perp, \perp)$.
- Let N be the maximum number of blocks stored to the OPQ. Then, our data structure consists of a L-level hierarchy of OTMs, where $L = \lceil \log N \rceil + 1$. For each level $i = 1, \ldots, L$,

we assume the instance OTM_i of capacity 2^{i-1}, i.e., there is a table $\mathbf{T}_i$ of size 2^i in each OTM_i. In our scheme, each $\mathbf{T}_i$ is updated only via the three operations of OTM_i, but a block in a cell $\mathbf{T}_i[j]$ is directly referred by any algorithms specifying i and j. In addition, we assign a binary state filled or emptyto each OTM_i. We say that OTM_i is filled if OTM_i.Extract has not been performed after the last OTM_i.Build, otherwise say that OTM_i is empty.

- For each (non-dummy) block (k, τ, v), let $\mathsf{ref} = (k, \tau, \mathsf{pos}, \mathsf{lv})$ be the *reference* of v, which uniquely identifies that the value v of key k, inserted in time τ, is in $\mathbf{T}_{\mathsf{lv}}[\mathsf{pos}]$. Also, let $\mathsf{ref}_\perp = (\perp, \perp, \perp, \perp)$ be a dummy reference.
- Our data structure also contains a complete binary tree of height L, called MinTree, that manages the reference to the top-priority block. In this tree, each node $\mathsf{node}_{i,j}; j = 1, \ldots, 2^i$ of height $i = 1, \ldots, L$ (except the root) corresponds to a cell $\mathbf{T}_i[j]$, and holds a reference ref that satisfies Claim 16 described below.

We call the data structure described above a *Hierarchical Tree*, and also show its abstract in Figure 6.1.

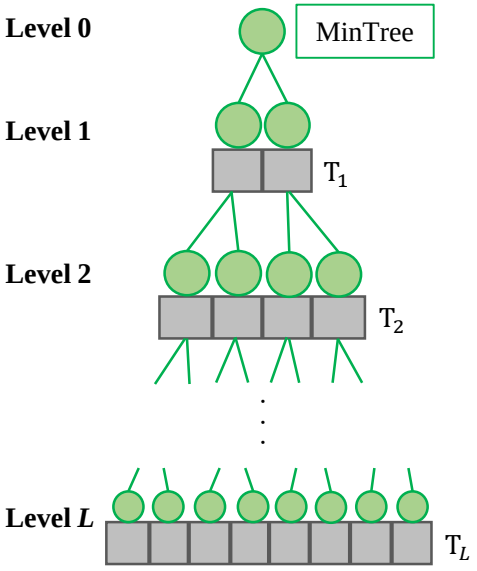

Figure 6.1: A Hierarchical Tree: Each block (k, τ, v) in the Hierarchical Tree is placed to one of the cells of $\mathbf{T}_i$, the gray squares. The green circles denote the nodes of MinTree, each of which holds a reference ref.

Claim 16 (A state of MinTree)**.** *Each node in* MinTree *satisfies the following conditions.*

- *For level $i = 0, \ldots, L$, each node $\mathsf{node}_{i,j} (j = 1, \ldots, 2^i)$ holds the top-priority reference $\mathsf{ref}_{i,j} = (k_{i,j}^{\min}, \tau_{i,j}^{\min}, pos_{i,j}^{\min}, \mathsf{lv}_{i,j}^{\min})$ s.t. $k_{i,j}^{\min}$ is the minimum key of all cells each corresponding to a node of* MinTree*'s subtree where the root is $\mathsf{node}_{i,j}$, and $\tau_{i,j}^{\min}$ indicates the first time a block with $k_{i,j}^{\min}$ is inserted. $(pos_{i,j}^{\min}, \mathsf{lv}_{i,j}^{\min})$ denotes the position where the block that has $(k_{i,j}^{\min}, \tau_{i,j}^{\min})$ is.*
- *In particular, for level L, each leaf $\mathsf{node}_{L,j} (j = 1, \ldots, 2^L)$ holds the reference $\mathsf{ref}_{L,j} = (k_{L,j}, \tau_{L,j}, j, L)$ of its corresponding cell $\mathbf{T}_L[j]$.*

If there exist two or more references of the minimum key, let the top-priority one is that has the minimum τ. Also, if there are no real keys, let $\mathsf{ref}_\perp$ be the top-priority reference.

In Lemma 22, we show that Claim 16 always holds in our scheme.

6.2 Priority-Queue Operations for the Hierarchical Tree

Before we describe the details of our entire OPQ, we introduce the separate implementation of each PQ operation. The implementations of Insert and Delete are shown in Section 6.2.2 and 6.2.3, respectively. Each of ExtractMin and ModKey is in practice realized by using Insert and Delete (see Section 6.2.4 and 6.2.5). A set of these implementations is almost[1] an oblivious simulation of $\mathcal{F}_{\mathsf{PQ}}$, but clearly leaks its running operation.

6.2.1 Subroutine: SelectMin

We first introduce a subroutine SelectMin, described in Algorithm 15, to manage references in MinTree. This SelectMin receives node, a node of MinTree, and it updates the input node with the top-priority reference among one that refers the cell corresponding to node and ones held by its children. It requires $O(1)$ work and has access patterns determined by node.

Algorithm 15 SelectMin(node)

1: Obtain the reference ref of the cell corresponding to node, i.e., if node is the pos-th node in level lv, load the corresponding cell $\mathbf{T}_{\mathsf{lv}}[\mathsf{pos}] = (k, \tau, v)$ and set $\mathsf{ref} = (k, \tau, \mathsf{pos}, \mathsf{lv})$.
If $\mathbf{T}_{\mathsf{lv}}[\mathsf{pos}] = (\perp, \perp, \perp)$ or $\mathsf{OTM}_{\mathsf{lv}}$ is empty, then set $\mathsf{ref} = \mathsf{ref}_\perp$.
Also, if node is the root of MinTree, then set $\mathsf{ref} = \mathsf{ref}_\perp$.
2: Load the references $\mathsf{ref}_L = (k_L, \tau_L, \mathsf{pos}_L, \mathsf{lv}_L)$ and $\mathsf{ref}_R = (k_R, \tau_R, \mathsf{pos}_R, \mathsf{lv}_R)$ from the two children of node.
If node is a leaf of MinTree, set $\mathsf{ref}_L = \mathsf{ref}_R = \mathsf{ref}_\perp$.
3: Select the top-priority one $\mathsf{ref}_{\min}$ from ref, ref_L, and ref_R.
4: Store $\mathsf{ref}_{\min}$ to node.

Lemma 17. *Assume that $\mathbf{T}_{\mathsf{lv}}[pos]$ is modified in operating the Hierarchical Tree, and* MinTree *satisfies Claim 16 before this modification. Let $\mathsf{node}_{\mathsf{lv}}$ be a node corresponding to the cell $\mathbf{T}_{\mathsf{lv}}[pos]$, and let $\mathsf{node}_{\mathsf{lv}-1}, \ldots, \mathsf{node}_0$ be nodes on the path from $\mathsf{node}_{\mathsf{lv}}$ to the root (= node_0). Then, the claim holds even after the modification if $\mathsf{SelectMin}(\mathsf{node}_i)$ is executed in the order from $i = \mathsf{lv}$ to 0.*

The proof of Lemma 17 is given in Appendix C.3

6.2.2 The Insert Algorithm

In Algorithm 16, we describe the details of our implementation of Insert that consists of two phases: Inserting the input to the OTM hierarchy and Updating MinTree.

In the first phase, the input block is inserted to the lowest-level OTM that is empty, and also all blocks in upper filled OTMs are moved to that level. Since the capacity of OTM_i is 2^{i-1}, in line 3 of Algorithm 16, the size of $\mathbf{I}_\ell$ finally becomes to $1 + \sum_{i=1}^{\ell-1} 2^{i-1} = 2^{\ell-1}$, and hence it satisfies

[1] As we mention in Remark 2 of Section 6.2.3, consecutive multiple runs of Delete violate obliviousness.

[2] This process is realized by using an *oblivious sorting* relying on such as AKS sorting network [1], i.e., obliviously sort all blocks of $\mathbf{I}_\ell$ by ascending order of keys and then set $\mathbf{I}_L$ as the first N elements of $\mathbf{I}_\ell$ (that contains $2N - 1$ blocks).

Algorithm 16 $\overrightarrow{\mathsf{ref}} \leftarrow \mathsf{Insert}(k, \tau, v)$

(Inserting a block (k, τ, v) to the OTM hierarchy.)

1: Let ℓ be the lowest level where OTM_ℓ is empty. If all L OTMs are filled, then set ℓ as $L+1$.
2: Initialize a set $\mathbf{I}_\ell = \{(k, \tau, v)\}$.
3: For $i = 1, \ldots, \ell - 1$, perform $\mathbf{I}_i \leftarrow \mathsf{OTM}_i.\textsc{Extract}()$, then set $\mathbf{I}_\ell \leftarrow \mathbf{I}_\ell \cup \widetilde{\mathbf{I}}_i$ and OTM_i as empty.
4: If $\ell \leq L$, run $\mathbf{U}_\ell \leftarrow \mathsf{OTM}_\ell.\textsc{Build}(\mathbf{I}_\ell)$ and set OTM_ℓ as filled.
Otherwise, compute a set $\mathbf{I}_L$ of size N, which contains all real blocks of $\mathbf{I}_\ell$[2], then run $\mathbf{U}_L \leftarrow \mathsf{OTM}_L.\textsc{Build}(\mathbf{I}_L)$ and set OTM_L as filled.
5: For all $(k_j, \tau_j, v_j) \in \mathbf{I}_\ell$ and $\mathsf{pos}_j \in \mathbf{U}_\ell$, set $\mathsf{ref}_j = (k_j, \tau_j, \mathsf{pos}_j, \ell)$.
If $\ell = L + 1$, consider the above $\mathbf{I}_\ell$ and $\mathbf{U}_\ell$ as $\mathbf{I}_L$ and $\mathbf{I}_L$, respectively.

(Updating $\mathsf{MinTree}$.)

6: For each level i from ℓ to 0, for all $\mathsf{node}_{i,j}; j = 1, \ldots, 2^\ell$, perform $\mathsf{SelectMin}(\mathsf{node}_{i,j})$.
7: **Return** $\overrightarrow{\mathsf{ref}} = (\mathsf{ref}_1, \ldots, \mathsf{ref}_{2^{\ell-1}})$.

the capacity of OTM_ℓ when $\ell \leq L$. Moreover, even when $\ell = L + 1$, since the total capacity of the Hierarchical Tree is N, there are at most N real elements in $\mathbf{I}_{L+1}$. Hence, all real elements in the Hierarchical Tree are correctly stored to OTM_L in line 4.

Finally, since $\mathbf{T}_\ell$ is rebuilt and lower $\mathsf{OTM}_i; i > \ell$ are not changed, Insert updates all nodes in level ℓ or higher.

Lemma 18 (Efficiency of Insert)**.** Insert *described in Algorithm 16 requires (amortized)* $O(\log^2 N)$ *work.*

Lemma 19 (Correctness of Insert)**.** *After an execution of* Insert *described in Algorithm 16, the input block* (k, τ, v) *is correctly stored to the Hierarchical Tree, and any block previously held in the Hierarchical tree remains in it, with probability* 1*. Moreover, if Claim 16 has been satisfied before an execution of* Insert*, it is satisfied thereafter.*

The proofs of Lemma 18 and 19 are given in Appendices C.4 and C.5, respectively.

Remark 1. *Though* Insert *described in Algorithm 16 can work almost as an implementation of* $\mathcal{F}_{\mathsf{PQ}}(\mathsf{Insert}, k, v)$*, for our goal to hide the running operations, we consider it only as a building block of our entire scheme. Hence, here we define* Insert *to take* τ *as input, unlike* $\mathcal{F}_{\mathsf{PQ}}$*.*

6.2.3 The Delete Algorithm

We show the details of our implementation of Delete in Algorithm 17. Similar to Insert, this Delete also has two phases to remove the queried block and then update $\mathsf{MinTree}$. In this algorithm, since L cells in the hierarchy are accessed to remove one block to hide which cell is actually queried, we should update all nodes corresponding the L cells as well as their all ancestors.

Lemma 20 (Efficiency of Delete)**.** Delete *described in Algorithm 17 requires* $O(log^2 N)$ *work.*

Lemma 21 (Correctness of Delete)**.** *After an execution of* Delete *described in Algorithm 17, the exact element referred by the input is extracted from the Hierarchical Tree, and all other elements remain in the Hierarchical Tree, with probability* 1*. Moreover, if Claim 16 has been satisfied before, it still holds after an execution of* Delete*.*

Proof of Lemma 20 and 21 are given in Appendices C.6 and C.7, respectively.

Algorithm 17 $(k, \tau, v) \leftarrow \mathsf{Delete}(\mathsf{ref})$

(Removing (k, τ, v) from the OTM hierarchy.)

1: **for all** level $i = 1, \ldots, L$ where OTM_i is filled **do**
2: For the input $\mathsf{ref} = (k, \tau, \mathsf{pos}, \mathsf{lv})$, set $\mathsf{pos}_i = \mathsf{pos}$ if $i = \mathsf{lv}$, otherwise set $\mathsf{pos}_i = \perp$.
3: Perform $(k_i, \tau_i, v_i), \mathsf{pos}'_i \leftarrow \mathsf{OTM}_i.\textsc{Lookup}(\mathsf{pos}_i)$.
4: Set $(k, \tau, v) = (k_i, \tau_i, v_i)$ for some i s.t. (k_i, τ, v_i) is a non-dummy block. If all (k_i, τ_i, v_i) is dummy, then set $(k, \tau, v) = (\perp, \perp, \perp)$.

(Updating MinTree.)

5: **for** level $i = L$ to 0 **do**
6: Run $\mathsf{SelectMin}(\mathsf{node}_{i,\mathsf{pos}'_i})$.
7: Run $\mathsf{SelectMin}(\mathsf{node}_{i,j})$ for all $\mathsf{node}_{i,j}$ s.t. at least one of the nodes $\{\mathsf{node}_{i+1,\mathsf{pos}'_{i+1}}, \ldots, \mathsf{node}_{L,\mathsf{pos}'_L}\}$ is a descendant of $\mathsf{node}_{i,j}$.
▷ The number of $\mathsf{node}_{i,j}$ is at most $L - i$.
8: **Return** (k, τ, v).

Remark 2. *Note that this* Delete *does not work as* $\mathcal{F}_{\mathsf{PQ}}(\mathsf{Delete}, *)$ *by itself. Specifically, for example, operation sequences such as executing* Delete *twice in a row violate obliviousness of* OTM_1 *that allows only one* Lookup *request in its lifetime. We solve this problem in the construction of our entire OPQ.*

6.2.4 The ExtractMin Algorithm

Algorithm 18 $(k_{\min}, \tau_{\min}, v_{\min}) \leftarrow \mathsf{ExtractMin}()$

1: Load the reference $\mathsf{ref}_{\min}$ in the root of MinTree.
2: Perform $(k_{\min}, \tau_{\min}, v_{\min}) \leftarrow \mathsf{Delete}(\mathsf{ref}_{\min})$.
3: **Return** $(k_{\min}, \tau_{\min}, v_{\min})$.

Algorithm 18 denotes an implementation of ExtractMin to obtain the top-priority block. Since the root of MinTree always has the reference to the top-priority block when Claim 16 holds, this algorithm correctly extracts the block relying on correctness of Delete.

6.2.5 The ModKey Algorithm

Algorithm 19 $\mathsf{ModKey}(k', \mathsf{ref})$

1: Perform $(k, \tau, v) \leftarrow \mathsf{Delete}(\mathsf{ref})$.
2: Perform $\overrightarrow{\mathsf{ref}} \leftarrow \mathsf{Insert}(k', \tau, v)$.

As shown in Algorithm 19, we can straightforwardly achieve ModKey as that; Delete a queried block, modify the key of the block, and then re-insert it. If Delete correctly removes the target block from the Hierarchical Tree, and if Insert correctly stores the input block to that, it is clear that Algorithm 19 is an implementation of $\mathcal{F}_{\mathsf{PQ}}$.ModKey.

Algorithm 20 $\overrightarrow{\mathsf{ref}}, (\tilde{k}, \tilde{v}) \leftarrow \mathsf{PQ}(\mathsf{op}, k, v, \mathsf{ref})$

Require: $\mathsf{op} \in \{\mathsf{Insert}, \mathsf{Delete}, \mathsf{ExtractMin}, \mathsf{ModKey}\}$.

(Extracting a block requested by op.)

1: Let ctr be a counter that indicates the number of PQ performed so far.
2: Load the reference $\mathsf{ref}_{\min}$ in the root of MinTree.
3: Set $\mathsf{ref}' = \mathsf{ref}$ if $\mathsf{op} \in \{\mathsf{Delete}, \mathsf{ModKey}\}$,
or set $\mathsf{ref}' = \mathsf{ref}_{\min}$ if $\mathsf{op} = \mathsf{ExtractMin}$.
Otherwise, set $\mathsf{ref}' = \mathsf{ref}_{\perp}$.
4: Perform $(\tilde{k}, \tilde{\tau}, \tilde{v}) \leftarrow \mathsf{Delete}(\mathsf{ref}')$.

(Inserting a block requested by op.)

5: Set $(k', \tau', v') = (k, \mathsf{ctr}, v)$ if $\mathsf{op} = \mathsf{Insert}$,
or set $(k', \tau', v') = (k, \tilde{\tau}, \tilde{v})$ if $\mathsf{op} = \mathsf{ModKey}$.
Otherwise, set $(k', \tau', v') = (\perp, \perp, \perp)$.
6: Perform $\overrightarrow{\mathsf{ref}} \leftarrow \mathsf{Insert}(k', \tau', v')$.
7: Set $\mathsf{ctr} \leftarrow \mathsf{ctr} + 1$.
8: **Return** $\overrightarrow{\mathsf{ref}}, (\tilde{k}, \tilde{v})$.

6.3 Our Perfectly Secure OPQ, PQ

Now, we can construct our perfectly secure OPQ that supports all four operations described above, while hiding *which operation is in progress*. I.e., the OPQ simulates all Insert, Delete, ExtractMin, and ModKey with access patterns independent from the running operation.

Remarking that our ExtractMin and ModKey are realized on an execution of Delete (or Delete-then-Insert sequence), and also that our Insert can refresh OTMs reached their limit of lookup requests via Delete[3], we can obtain a *deterministic sequence* that realizes OPQ. We show the details of our scheme in Algorithm 20.

Lemma 22. *Assume the initial state of the Hierarchical Tree where all* $\mathsf{OTM}_i; i = 1, \ldots, L$ *are* empty*and all nodes in* MinTree *holds* $\mathsf{ref}_{\perp}$*. Then, Claim 16 holds after any* PQ *execution.*

Proof. In the initial state, it is clear that MinTree satisfies Claim 16. Therefore, from Lemma 19 and 21, the claim holds after any execution of PQ. □

Theorem 23. PQ*, described in Algorithm 20, is a perfectly oblivious implementation of* $\mathcal{F}_{\mathsf{PQ}}$ *that consumes amortized* $O(\log^2 N)$ *total work per operation with* $\Omega(\log N)$ *block size.*

Proof. Proof of security is given in Appendix 23, and we show efficiency and correctness of our PQ here. The efficiency is derived by Lemma 18 and 20 as that: Delete requires $O(\log^2 N)$ work and Insert requires amortized $O(\log^2 N)$ work.

To prove correctness of PQ, we show PQ behaviors at each input op.

- If $\mathsf{op} = \mathsf{Insert}$, PQ runs $\mathsf{Delete}(\mathsf{ref}_{\perp})$ then $\mathsf{Insert}(k, \tau, v)$ where τ is a query counter.
- If $\mathsf{op} = \mathsf{Delete}$, PQ runs $\mathsf{Delete}(\mathsf{ref})$ then $\mathsf{Insert}(\perp, \perp, \perp)$.
- If $\mathsf{op} = \mathsf{ExtractMin}$, PQ runs the same algorithm as $\mathsf{ExtractMin}()$ then $\mathsf{Insert}(\perp, \perp, \perp)$.
- If $\mathsf{op} = \mathsf{ModKey}$, PQ runs the same algorithm as $\mathsf{ModKey}(k, \mathsf{ref})$.

[3]The similar *access-then-re-randomize* sequence is commonly used in hierarchical ORAMs [31, 4, 13, 15].

Now, by Lemma 19 and 21, the interface of PQ with each $\mathsf{op} \in \{\mathsf{Insert}, \mathsf{Delete}, \mathsf{ModKey}\}$ is equivalent to that of $\mathcal{F}_{\mathsf{PQ}}.\mathsf{Insert}$, $\mathcal{F}_{\mathsf{PQ}}.\mathsf{Delete}$, and $\mathcal{F}_{\mathsf{PQ}}.\mathsf{ModKey}$, respectively. Moreover, with $\mathsf{op} = \mathsf{ExtractMin}$, the interface of PQ is equivalent to that of $\mathcal{F}_{\mathsf{PQ}}.\mathsf{ExtractMin}$ by Lemma 21 and 22. □

Chapter 7

Conclusion

We proposed an optimal DORAM of $O(\log N)$ overhead with small hidden constant, that relies on only $4 \log N$ OPRF calls. The key building block is a novel 3-party permutation protocol, in which parties split their roles into one permuter that knows the whole permutation and two storages that hold a permuted table. Since these roles do not overlap, the permuter never observes access patterns to the permuted table even though it knows the structure of the table.

In addition, we extended the above (passively secure) DORAM to an actively secure one. Since our passively secure construction depends on the permutation protocol in which one party has full control of a permutation, we additionally construct a novel protocol to verify the permutation provided by the (possibly dishonest) party. Then, we achieved an actively secure DORAM of $O(\log N)$ overhead with slightly larger $\omega((\lambda + b) \log N)$-bit block size.

We also proposed a novel perfectly secure oblivious priority queue that supports four operations, Insert, Delete, ExtractMin, and ModKey, while hiding not only its access patterns but also queried operations. This property is the same as the known best OPQ of Shi [60] except our stronger security. Moreover, for any positive integer N, our OPQ of capacity N can be accessed in amortized $O(\log^2 N)$ total work, which is the same cost as the known perfectly secure OPQ of Toft [63].

Acknowledgements

I would like to thank all involved in this study for their discussion, contribution, guidance, and encouragement. I would like to thank my supervisor, Professor Wakaha Ogata for her numerous mentoring and encouragement throughout this study and this dissertation. I would like to thank my co-authors, Ilan Komargodski, Koki Hamada, Ryo Kikuchi, and Dai Ikarashi, for their helpful advice, discussions, and contributions to the DORAM results of this study. I would like to thank my immediate superiors, Toshiyuki Miyazawa and Koji Chida (who has been an associate professor at Gunma University), for their enormous support in balancing work and doctoral studies. I would like to thank Professor Tomohiko Uematsu, Professor Isao Yamada, Professor Keisuke Tanaka, Professor Ryutaroh Matsumoto, and Associate Professor Kenji Yasunaga, for reviewing this dissertation and having a vigorous discussion on this study. I am also grateful to the members of Ogata Laboratory for their interesting discussions. Finally, I appreciate my wife Saori, my parents Koya and Hiro, and my younger brother Ryoga, for their dedicated support in my personal life.

Author's Publications Relevant to This Study

Refereed Papers

1. Ichikawa, A., Ogata, W.: Perfectly secure oblivious priority queue. In: IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences. 106(3): 272–280 (2023).

2. Ichikawa, A., Komargodski, I., Hamada, K., Kikuchi, R., Ikarashi, D.: 3-party secure computation for rams: Optimal and concretely efficient. In: TCC. pp. 471–502 (2023).

Non-refereed Papers

1. Ichikawa, A., Hamada, K., Kikuchi, R., Ikarashi, D.: Optimal oblivious hashing and sublogarithmic oram for 3pc. In: SCIS (2020).

2. Ichikawa, A.: Improvement for perfectly secure oblivious priority queues based on multiparty computation. In: CSS (2021).

Author's Other Publications

Refereed Papers

1. Ichikawa, A., Ogata, W., Hamada, K., Kikuchi, R.: Efficient secure multi-party protocols for decision tree classification. In: ACISP. pp. 362–380 (2019).

2. Eriguchi, R., Ichikawa, A., Kunihiro, N., Nuida, K.: Efficient noise generation to achieve differential privacy with applications to secure multiparty computation. In: FC. pp. 271–290 (2021).

3. Eriguchi, R., Ichikawa, A., Kunihiro, N., Nuida, K.: Efficient noise generation protocols for differentially private multiparty computation. In: IEEE Transactions on Dependable and Secure Computing. 20(6): 4486–4501 (2023).

4. Chida, K., Hamada, K., Ichikawa, A., Kii, M., Tomida, J.: Communication-efficient inner product private join and compute with cardinality. In: AsiaCCS. pp. 678–688 (2023).

Non-refereed Papers

1. Ichikawa, A., Ogata, W.: Cheating detectable multi-use password protected secret sharing secure against active adversary. In: IEICE Tech. Rep., vol. 115, no. 28, ISEC2015-4, pp. 23–30 (2015).

2. Ichikawa, A., Kikuchi, R., Ogata, W.: Efficiency comparison of secure decision tree computation and its improvement by offline computation. In: SCIS (2016).

3. Ichikawa, A., Ogata, W.: Efficient multiparty computation for decision tree using oblivious ram. In: SCIS (2017).

4. Ichikawa, A., Hamada, K., Tanaka, S.: Shopping with anonymous! Secure e-commerce system, and record analysis on it. In: SCIS (2018).

5. Ichikawa, A., Hamada, K.: Implementation and evaluation of oblivious decision tree for secure computation. In: CSS (2018).

6. Ichikawa, A., Hamada, K.: Auto privact-preserving query answering without libraries. In: SCIS (2019).

7. Ichikawa, A., Sudo, H., Takenouchi, D., Ikarashi, D., Mishina, I., Hamada, K., Kikuchi, R.: Practical secure computation for medical statistics on big data. In: CSS (2019).

8. Ichikawa, Hamada, K.: Oblivious ram for conjunctive keyword. In: CSS (2019).

9. Ichikawa, A., Sudo, H., Mishina, I., Takenouchi, D., Kikuchi, R., Hamada, K., Ikarashi, D.: Fast Secure elementary function algorithms make secure machine-learning and advanced statistical analyses practical. In: SCIS (2020).

10. Ichikawa, A., Hamada, K.: Efficient perfectly secure distributed oram without client operation. In: SCIS (2021).

11. Ichikawa, A.: Succinct random-index oram. In: CSS (2022).

12. Ichikawa, A.: Round-efficient hierarchical multi-server oram. In: SCIS (2023).

Bibliography

[1] Ajtai, M., Komlós, J., Szemerédi, E.: An 0(n log n) sorting network. In: Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing. p. 1–9. STOC '83, Association for Computing Machinery, New York, NY, USA (1983). https://doi.org/10.1145/800061.808726

[2] Albrecht, M.R., Rechberger, C., Schneider, T., Tiessen, T., Zohner, M.: Ciphers for mpc and fhe. In: EUROCRYPT. pp. 430–454 (2015)

[3] Araki, T., Furukawa, J., Lindell, Y., Nof, A., Ohara, K.: High-throughput semi-honest secure three-party computation with an honest majority. In: CCS. pp. 805–817 (2016)

[4] Asharov, G., Komargodski, I., Lin, W., Nayak, K., Peserico, E., Shi, E.: Optorama: Optimal oblivious RAM. J. ACM **70**(1), 4:1–4:70 (2023)

[5] Asharov, G., Komargodski, I., Lin, W., Peserico, E., Shi, E.: Optimal oblivious parallel RAM. In: ACM-SIAM Symposium on Discrete Algorithms, SODA. pp. 2459–2521 (2022)

[6] Asharov, G., Komargodski, I., Lin, W., Shi, E.: Oblivious RAM with worst-case logarithmic overhead. In: CRYPTO. pp. 610–640 (2021)

[7] Blass, E.O., Mayberry, T., Noubir, G.: Multi-client oblivious ram secure against malicious servers. In: ACNS. pp. 686–707 (2017)

[8] Boyle, E., Naor, M.: Is there an oblivious RAM lower bound? In: ITCS. pp. 357–368 (2016)

[9] Bunn, P., Katz, J., Kushilevitz, E., Ostrovsky, R.: Efficient 3-party distributed oram. Cryptology ePrint Archive (2018)

[10] Canetti, R.: Security and composition of multiparty cryptographic protocols. J. Cryptology **13**(1), 143–202 (2000)

[11] Catrina, O., de Hoogh, S.: Improved primitives for secure multiparty integer computation. In: Garay, J.A., De Prisco, R. (eds.) SCN. pp. 182–199 (2010)

[12] Chan, T.H.H., Guo, Y., Lin, W.K., Shi, E.: Oblivious hashing revisited, and applications to asymptotically efficient oram and opram. In: ASIACRYPT. pp. 660–690 (2017)

[13] Chan, T.H.H., Nayak, K., Shi, E.: Perfectly secure oblivious parallel ram. In: Theory of Cryptography. pp. 636–668. Springer International Publishing, Cham (2018)

[14] Chan, T.H., Shi, E.: Circuit OPRAM: unifying statistically and computationally secure orams and oprams. In: TCC. pp. 72–107 (2017)

[15] Chan, T.H.H., Shi, E., Lin, W.K., Nayak, K.: Perfectly oblivious (parallel) ram revisited, and improved constructions. Cryptology ePrint Archive, Report 2020/604 (2020), https://ia.cr/2020/604

[16] Chida, K., Genkin, D., Hamada, K., Ikarashi, D., Kikuchi, R., Lindell, Y., Nof, A.: Fast large-scale honest-majority mpc for malicious adversaries. In: CRYPTO. pp. 34–64 (2018)

[17] Chida, K., Hamada, K., Ikarashi, D., Kikuchi, R., Kiribuchi, N., Pinkas, B.: An efficient secure three-party sorting protocol with an honest majority. Cryptology ePrint Archive (2019)

[18] Chida, K., Hamada, K., Ikarashi, D., Kikuchi, R., Pinkas, B.: High-throughput secure aes computation. In: WAHC. p. 13–24 (2018)

[19] Cramer, R., Damgård, I., Ishai, Y.: Share conversion, pseudorandom secret-sharing and applications to secure computation. In: TCC. pp. 342–362 (2005)

[20] Damgård, I., Keller, M.: Secure multiparty aes. In: FC. pp. 367–374 (2010)

[21] Devadas, S., van Dijk, M., Fletcher, C.W., Ren, L., Shi, E., Wichs, D.: Onion oram: A constant bandwidth blowup oblivious ram. In: TCC. pp. 145–174 (2016)

[22] Dittmer, S., Ostrovsky, R.: Oblivious tight compaction in o(n) time with smaller constant. In: SCN. pp. 253–274 (2020)

[23] Faber, S., Jarecki, S., Kentros, S., Wei, B.: Three-party ORAM for secure computation. In: ASIACRYPT. pp. 360–385 (2015)

[24] Falk, B., Noble, D., Ostrovsky, R., Shtepel, M., Zhang, J.: Doram revisited: Maliciously secure ram-mpc with logarithmic overhead. IACR Cryptol. ePrint Arch. p. 578 (2023)

[25] Falk, B.H., Noble, D., Ostrovsky, R.: Alibi: A flaw in cuckoo-hashing based hierarchical ORAM schemes and a solution. In: Advances in Cryptology - EUROCRYPT. pp. 338–369 (2021)

[26] Falk, B.H., Noble, D., Ostrovsky, R.: 3-party distributed ORAM from oblivious set membership. In: SCN. pp. 437–461 (2022)

[27] Furukawa, J., Lindell, Y., Nof, A., Weinstein, O.: High-throughput secure three-party computation for malicious adversaries and an honest majority. In: EUROCRYPT. pp. 225–255 (2017)

[28] Genkin, D., Ishai, Y., Polychroniadou, A.: Efficient multi-party computation: From passive to active security via secure simd circuits. In: CRYPTO. pp. 721–741 (2015)

[29] Genkin, D., Ishai, Y., Prabhakaran, M.M., Sahai, A., Tromer, E.: Circuits resilient to additive attacks with applications to secure computation. In: STOC. pp. 495–504 (2014)

[30] Goldreich, O.: Towards a theory of software protection and simulation by oblivious rams. In: STOC. pp. 182–194 (1987)

[31] Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious rams. J. ACM **43**(3), 431–473 (1996)

[32] Goodrich, M.T., Mitzenmacher, M.: Privacy-preserving access of outsourced data via oblivious ram simulation. In: ICALP. pp. 576–587 (2011)

[33] Hoang, T., Guajardo, J., Yavuz, A.A.: MACAO: A maliciously-secure and client-efficient active ORAM framework. In: NDSS (2020)

[34] Hoang, T., Ozkaptan, C.D., Yavuz, A.A., Guajardo, J., Nguyen, T.: S3oram: A computation-efficient and constant client bandwidth blowup oram with shamir secret sharing. In: CCS. pp. 491–505 (2017)

[35] Håstad, J., Impagliazzo, R., Levin, L.A., Luby, M.: A pseudorandom generator from any one-way function. J. Computing **28**(4), 1364–1396 (1999)

[36] Ikarashi, D., Kikuchi, R., Hamada, K., Chida, K.: Actively private and correct mpc scheme in $t < n/2$ from passively secure schemes with small overhead. Cryptology ePrint Archive (2014)

[37] Ito, M., Saito, A., Nishizeki, T.: Secret sharing scheme realizing general access structure. GLOBECOM pp. 99–102 (1987)

[38] Jacob, R., Larsen, K.G., Nielsen, J.B.: Lower bounds for oblivious data structures. In: SODA. pp. 2439–2447 (2019)

[39] Jafargholi, Z., Larsen, K.G., Simkin, M.: Optimal Oblivious Priority Queues, p. 2366–2383. Society for Industrial and Applied Mathematics, USA (2021)

[40] Keller, M., Scholl, P.: Efficient, oblivious data structures for mpc. In: ASIACRYPT. pp. 506–525 (2014)

[41] Kikuchi, R., Attrapadung, N., Hamada, K., Ikarashi, D., Ishida, A., Matsuda, T., Sakai, Y., Schuldt, J.C.N.: Field extension in secret-shared form and its applications to efficient secure computation. In: ACISP. pp. 343–361 (2019)

[42] Komargodski, I., Lin, W.: A logarithmic lower bound for oblivious RAM (for all parameters). In: CRYPTO. pp. 579–609 (2021)

[43] Kushilevitz, E., Lu, S., Ostrovsky, R.: On the (in)security of hash-based oblivious ram and a new balancing scheme. In: SODA. pp. 143–156 (2012)

[44] Kushilevitz, E., Mour, T.: Sub-logarithmic distributed oblivious ram with small block size. In: PKC. pp. 3–33 (2019)

[45] Larsen, K.G., Nielsen, J.B.: Yes, there is an oblivious RAM lower bound! In: CRYPTO. pp. 523–542 (2018)

[46] Larsen, K.G., Simkin, M., Yeo, K.: Lower bounds for multi-server oblivious rams. In: TCC. pp. 486–503 (2020)

[47] Laur, S., Talviste, R., Willemson, J.: From oblivious aes to efficient and secure database join in the multiparty setting. In: ACNS. pp. 84–101 (2013)

[48] Laur, S., Willemson, J., Zhang, B.: Round-efficient oblivious database manipulation. In: ISC. pp. 262–277 (2011)

[49] Lu, S., Ostrovsky, R.: Distributed oblivious ram for secure two-party computation. In: TCC. pp. 377–396 (2013), full version: https://eprint.iacr.org/2011/384

[50] Maffei, M., Malavolta, G., Reinert, M., Schröder, D.: Maliciously secure multi-client oram. Cryptology ePrint Archive (2017)

[51] Naor, M.: Bit commitment using pseudo-randomness (extended abstract). In: CRYPTO. p. 128–136 (1989)

[52] Noble, D.: Explicit, closed-form, general bounds for cuckoo hashing with a stash. Cryptology ePrint Archive (2021)

[53] Ostrovsky, R.: Efficient computation on oblivious rams. In: STOC. pp. 514–523 (1990)

[54] Ostrovsky, R., Shoup, V.: Private information storage (extended abstract). In: STOC. pp. 294–303 (1997)

[55] Pagh, R., Rodler, F.F.: Cuckoo hashing. J. Algorithms **51**(2), 122–144 (2004)

[56] Patel, S., Persiano, G., Raykova, M., Yeo, K.: PanORAMa: Oblivious RAM with logarithmic overhead. In: FOCS. pp. 871–882 (2018)

[57] Pippenger, N., Fischer, M.J.: Relations among complexity measures. J. ACM **26**(2), 361–381 (1979)

[58] Ren, L., Fletcher, C.W., Yu, X., van Dijk, M., Devadas, S.: Integrity verification for path oblivious-ram. In: HPEC. pp. 1–6 (2013)

[59] Shamir, A.: How to share a secret. Commun. ACM **22**(11), 612–613 (1979)

[60] Shi, E.: Path oblivious heap: Optimal and practical oblivious priority queue. In: 2020 IEEE Symposium on Security and Privacy (SP). pp. 842–858. IEEE Computer Society, Los Alamitos, CA, USA (may 2020). https://doi.org/10.1109/SP40000.2020.00037

[61] Shi, E., Chan, T.H., Stefanov, E., Li, M.: Oblivious RAM with o((logn)3) worst-case cost. In: ASIACRYPT. pp. 197–214 (2011)

[62] Stefanov, E., van Dijk, M., Shi, E., Fletcher, C., Ren, L., Yu, X., Devadas, S.: Path oram: An extremely simple oblivious ram protocol. In: CCS. pp. 299–310 (2013)

[63] Toft, T.: Secure data structures based on multi-party computation. In: Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing. p. 291–292. PODC '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/1993806.1993859

[64] Wang, X., Chan, T.H., Shi, E.: Circuit ORAM: on tightness of the goldreich-ostrovsky lower bound. In: CCS. pp. 850–861 (2015)

[65] Wang, X.S., Nayak, K., Liu, C., Chan, T.H.H., Shi, E., Stefanov, E., Huang, Y.: Oblivious data structures. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. p. 215–226. CCS '14, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2660267.2660314

Appendix A

Comparing Concrete Efficiency of Our Passively Secure DORAM with [49] and [26]

Functionality	# of calls
$\mathcal{F}_{\text{Share}}$	$3L+1$
$\mathcal{F}_{\text{Reveal}}$	$4L$
$\mathcal{F}_{\text{Mult}}$	$4L+c$
$\mathcal{F}_{\text{Rnd}}$	2
$\mathcal{F}_{\text{Reshare}}$	$3L+1$
$\mathcal{F}_{\text{Eq}}$	$5L+c+1$
$\mathcal{F}_{\text{IfElse}}$	$6L+c+3$
$\mathcal{F}_{\text{Bitext}}$	L
$\mathcal{F}_{\text{Trunc}}$	L
$\mathcal{F}_{\text{Prf}}$	$4L$
$\mathcal{F}_{\text{Perm}}$	$p-1$, with input of size $c2^i+\frac{c}{2}$ for each $i=1,\ldots,p-1$ per $c2^{p-1}$ accesses.
$\mathcal{F}_{\text{Unperm}}$	1, with input of size $c2^p+\frac{c}{2}$ per $c2^{p-1}$ accesses.

Table A.1: The amortized number of (at most[2]) calls of each functionality in Algorithm 10, where $L=\lceil \log N - \log\log N \rceil$ and $c=2\lceil \log N \rceil$.

We evaluate the concrete computational complexity of our passively secure DORAM. We treat each functionality as a black box here and consider the (amortized) number of calls per access, for ease of estimation[3]. Table A.1 describes the concrete number of calls of the building blocks required in Algorithm 10.

[1]We omit the *minus* $\log\log N$ *factors* here to make the comparison with the prior works easier, thus the actual number of calls is slightly smaller than that.

[2]We omit the *minus* $\log\log N$ *factors* here to make the comparison with the prior works easier, thus the actual number of calls is slightly smaller than that.

[3]See Table 2 in [26], for example, for more specific costs of each functionality.

Comparison with [49] and [26]. Following the same setting as Falk et al. [26], we can estimate the efficiency of our DORAM more concretely. From our Table A.1 and Lemmas 6, 7, and Table 2 of [26], we conclude that Algorithm 10 requires $(181L + 9c + 40)B$ bits of communication for block size $B = \lambda + \Omega(\log N)$ bits. Considering $L \sim \log N$ and $c \sim 2 \log N$, the cost of our DORAM becomes $(199 \log N + 40)B$ bits per access.

According to [26], the DORAM of Lu and Ostrovsky [49] requires at least $100 \log N$ OPRF calls and hence costs at least $2100 \log N \cdot B$ bits of communication per access[4] — this is significantly higher than ours. On the other hand, the DORAM of Falk et al. [26] requires $(153.5 \log N + 6)B$ bits per access. I.e., even though the number of our calls to the most expensive building block is double, it does not affect the practical bandwidth much. Moreover, considering that the DORAM of Falk et al. requires additional computational cost for $\log^2 N$ non-oblivious hash functions per access, our optimal DORAM appears to be faster in the practical size of N.

[4] In practice, the cost of other functionalities, besides OPRF, is added to this.

Appendix B

Comparison of Our Actively Secure DORAM with [24]

A concurrent and independent work of Falk, Noble, Ostrovsky, Shtepel, and Zhang [24] achieves very similar results to us (both passive and active DORAMs) albeit with somewhat different techniques. As opposed to us, they rely and extend [26]. In particular, they obtain the optimal $O(\log N)$ overhead by a new $\omega(\log N)$-size cache that can be accessed efficiently, thereby avoiding the usage of a hierarchy of Bloom filters for small inputs.

We note that the usage of asymptotically larger than logarithmic size blocks in the actively secure DORAM is common to both works. For us, it stems from the usage of MACs that have negligible probability (in N) of being forged. For them, it stems from the usage of an actively secure MPC by Furukawa et al. [27] which requires $\omega(\log N)$ bits of communication per AND gate computation, assuming $\mathsf{negl}(N)$ statistical security is required. (The authors of [27] mention an optimization to perform multiplication in constant time but guaranteeing security only 2^{-40}.) Overall, treating N (the input size) as the statistical security parameter, the complexity of [24]'s DORAM is $O((\lambda + b) \log N) \cdot \omega(\log N)$. Alternatively, $O(\log N)$ overhead in accesses with block size $\omega((\lambda + b) \log N)$, which is identical to our complexity.

Appendix C

Deferred Proofs

C.1 Proof of Lemma 9

For correctness of our scheme, by [52], we observe that the permuter P achieves a permutation π for hashing with overwhelming probability. Since we are in the $\mathcal{F}_{\text{PERM}}$-hybrid model, all input blocks are correctly placed in their location as $\pi \cdot \mathsf{D} = \mathsf{T}\|\mathsf{S}$ in Algorithm 7. Similarly, since we are in the $\mathcal{F}_{\text{UNPERM}}$-hybrid model, the blocks are correctly retrieved as $\mathsf{D} = \pi^{-1} \cdot (\mathsf{T}\|\mathsf{S})$ in Algorithm 9. Moreover, when the above permutation π exists, in Algorithm 8, the storages always fetch $\mathsf{T}[\mathsf{addr}_0], \mathsf{T}[\mathsf{addr}_1]$, and S, one block of which is the queried data.

For obliviousness, consider a simulator Sim that simulates the view of a passive adversary that corrupts one of three party in $\Pi_{\text{BUILD}}, \Pi_{\text{LOOKUP}}$ and Π_{EXTRACT}.

- **Simulating Π_{Build}.** Receiving an instruction to simulate Π_{BUILD} with the size of input n and a collusion target $C \in \{P, S_0, S_1\}$, Sim runs the real protocol Π_{BUILD} on input n dummy blocks. If $C = P$ then Sim outputs π and all $\mathsf{addr}_{i,j}$ provided from $\mathcal{F}_{\text{PRP}}(\llbracket s_j \rrbracket, \llbracket \perp + i \rrbracket)$ as *access patterns*. Otherwise, Sim outputs nothing. Sim holds the result of Π_{BUILD} simulation, $(\pi, (\langle \mathsf{T} \rangle_{\{0,1\}}, \langle \mathsf{S} \rangle_{\{0,1\}}), \llbracket s_{\{0,1\}} \rrbracket, \mathsf{ctr}, \mathbf{P})$, as its state.

- **Simulating Π_{Lookup}.** When the adversary requests to run Π_{LOOKUP} with a key k, Sim simulates the real protocol Π_{LOOKUP} with a key $\perp$ and its state $(\pi, (\langle \mathsf{T} \rangle_{\{0,1\}}, \langle \mathsf{S} \rangle_{\{0,1\}}), \llbracket s_{\{0,1\}} \rrbracket, \mathsf{ctr}, \mathbf{P})$. If C is either S_0 or S_1, Sim outputs $\mathsf{addr}_{\{0,1\}}$ provided from $\mathcal{F}_{\text{PRP}}(\llbracket s_{\{0,1\}} \rrbracket, \llbracket \perp + \mathsf{ctr} \rrbracket)$ as *access patterns*. Otherwise, Sim outputs nothing.

- **Simulating Π_{Extract}.** When the adversary requests to run Π_{EXTRACT}, Sim halts and outputs π as an *access pattern* if $C = P$, otherwise it outputs nothing.

Since we are in $\mathcal{F}_{\text{ABB}}$-hybrid model and Algorithm 7, 8 and 9 are all deterministic except for the addresses recovered to a party, we should consider only the addresses as the access patterns. Now, consider the following hybrid argument.

$\underline{\mathsf{Hyb}_0}$: The real execution of $\Pi_{\text{BUILD}}, \Pi_{\text{LOOKUP}}$ and Π_{EXTRACT} in the $(\mathcal{F}_{\text{SHARE}}, \mathcal{F}^{\mathcal{P}}_{\text{REVEAL}}, \mathcal{F}_{\text{ABB}}, \mathcal{F}_{\text{PERM}}, \mathcal{F}_{\text{UNPERM}})$-hybrid model.

$\underline{\mathsf{Hyb}_1}$: The same as Hyb_0 except that, in the execution of Π_{BUILD} (Algorithm 7), the $\mathcal{F}_{\text{IFELSE}}$ functionality of line 4 always set $\widetilde{k}_i$ to $\perp + i$.

$\underline{\mathsf{Hyb}_2}$: The same as Hyb_1, but parties always search for a dummy in Π_{LOOKUP}, i.e., the $\mathcal{F}_{\text{IFELSE}}$ functionality of Algorithm 8, line 2, always set $\widetilde{k}$ to $\perp + \mathsf{ctr}$.

$\underline{\mathsf{Hyb}_3}$: The ideal simulation of Sim.

Let $\mathsf{View}_C(\mathsf{Hyb}_u)$ be the view of the adversary that colludes with $C \in \{P, S_0, S_1\}$ in Hyb_u. By the security of the PRF, $\mathsf{View}_P(\mathsf{Hyb}_0)$ and $\mathsf{View}_P(\mathsf{Hyb}_1)$ are computationally indistinguishable as long as $n < \mathsf{poly}(\lambda)$. $\mathsf{View}_{S_i}(\mathsf{Hyb}_0)$ and $\mathsf{View}_{S_i}(\mathsf{Hyb}_1)$ are obviously identical for any $i \in \{0, 1\}$.

It is also obvious that $\mathsf{View}_P(\mathsf{Hyb}_1) \equiv \mathsf{View}_P(\mathsf{Hyb}_2)$. In addition, $\mathsf{View}_{S_i}(\mathsf{Hyb}_1)$ and $\mathsf{View}_{S_i}(\mathsf{Hyb}_2)$ are computationally indistinguishable for any $i \in \{0, 1\}$ by security of the PRF. Furthermore, $\mathsf{View}_C(\mathsf{Hyb}_3)$ is the same as $\mathsf{View}_C(\mathsf{Hyb}_2)$ for any $C \in \{P, S_0, S_1\}$, directly.

C.2 Proof of Theorem 10

Consider the following simulator Sim that simulates the view of a passive adversary that corrupts one of three party in $\Pi_{\textsc{Access}}$.

- Let $\mathsf{Sim}_{\mathsf{HT}}(n)$ be a simulator that simulates the view of a passive adversary in distributed oblivious hashing with input length n. As setup, Sim initializes $\mathsf{Sim}_{\mathsf{HT}}(c), \ldots, \mathsf{Sim}_{\mathsf{HT}}(c2^{L-1})$ where $c = 2\lceil \log N \rceil$ and $L = \lceil \log N - \log\log N \rceil$. In addition, Sim allocates an empty array of size c and initialize a query counter $\mathsf{ctr} = 0$
- Receiving a request to simulate $\Pi_{\textsc{Access}}$, Sim simulates the ctr-th execution of $\Pi_{\textsc{Access}}$ with dummy inputs and an dummy data structure. All invocations of $\mathsf{HT}_i.\textsc{Lookup}$ is replaced with the lookup simulation of $\mathsf{Sim}_{\mathsf{HT}}(c2^{i-1})$, and also all $\mathsf{HT}_i.\textsc{Extract}$ and $\mathsf{HT}_i.\textsc{Build}$ in the subroutine $\Pi_{\textsc{Reshuffle}}$ are replaced with the extract and build simulation of that, respectively. Sim finally increments ctr and returns all outputs of $\mathsf{Sim}_{\mathsf{HT}}$.

Now, consider the following hybrid argument.

$\underline{\mathsf{Hyb}_0}$: The real execution of $\Pi_{\textsc{Access}}$.

$\underline{\mathsf{Hyb}_1}$: The same as Hyb_0 except performing $\Pi_{\textsc{Access}}([\![\mathsf{read}]\!], [\![\bot]\!], [\![\bot]\!])$ for any input.

$\underline{\mathsf{Hyb}_2}$: The ideal simulation of Sim.

Since we are in the $(\mathcal{F}_{\mathrm{ABB}}, \mathcal{F}_{\mathsf{HT}})$-hybrid model and the parties do not reveal any secret in the algorithm $\Pi_{\textsc{Access}}$ and its subroutine $\Pi_{\textsc{Reshuffle}}$, the adversary's view of Hyb_1 is identical to that of Hyb_0 with overwhelming probability. Furthermore, in the $(\mathcal{F}_{\mathrm{ABB}}, \mathcal{F}_{\mathsf{HT}})$-hybrid model, Hyb_1 and Hyb_2 are identical. The view of the adversary in the real $\Pi_{\textsc{Access}}$ and Sim are therefore indistinguishable.

C.3 Proof of Lemma 17

If only $\mathbf{T}_{\mathsf{lv}}[\mathsf{pos}]$ is modified when all nodes in $\mathsf{MinTree}$ satisfies the condition of Claim 16, it is clear that all nodes except ancestors of $\mathsf{node}_{\mathsf{lv},\mathsf{pos}}$ corresponding to $\mathbf{T}_{\mathsf{lv}}[\mathsf{pos}]$ still satisfy the condition after the modification. Furthermore, for any node, if both of its children holds the condition, i.e., each child has the top-priority reference in all of its descendants, then it is clear that $\mathsf{SelectMin}(\mathsf{node})$ makes node to meet the condition. Therefore, running $\mathsf{SelectMin}(\mathsf{node}_i)$ in order from $i = \mathsf{lv}$ to 0 while tracing the path from $\mathsf{node}_{\mathsf{lv}}$ to the root node_0 ensures that all nodes in $\mathsf{MinTree}$ meet the condition.

C.4 Proof of Lemma 18

When $\ell \leq L$, since Algorithm 16 requires executions of OTM_i.Extract for $i = 1, \ldots, \ell - 1$ and OTM_ℓ.Build, it takes $\sum_{i=1}^{\ell} O(2^i \log 2^i) = O(2^\ell \ell)$ work. It also needs SelectMin for $2^{\ell+1} - 1$ nodes, thus it additionally costs $O(2^\ell)$ work. Hence, the Insert on level ℓ has $O(2^\ell \ell)$ total work.

We can observe that the above work occurs for every 2^ℓ insertions, for each $\ell = 1, \ldots, L$. Consider that $\mathsf{OTM}_1, \ldots, \mathsf{OTM}_{\ell-1}$ are emptyand OTM_ℓ is filled, i.e., OTM_ℓ is built in the last insertion. Since $\mathsf{OTM}_1, \ldots, \mathsf{OTM}_{\ell-1}$ can hold totally $2^{\ell-1} - 1$ blocks, OTM_ℓ is deconstructed in $2^{\ell-1}$-th execution of Insert, in which $\mathsf{OTM}_{\ell+1}$ (or lower) is built. OTM_ℓ is constructed again after $2^{\ell-1}$ additional insertions, thus the life-cycle of OTM_ℓ is 2^ℓ insertions.

On the other hand, when $\ell = L+1$, Algorithm 16 additionally requires an oblivious sorting on $\mathbf{I}_\ell$ of size 2^{L+1}. If we employ the oblivious sorting based on AKS sorting network [1], since it requires $O(N \log N)$ to sort N elements, Insert costs totally $O(2^L L)$ work in this case. Once OTM_L is built, the above work occurs for every 2^{L-1} insertions since the total capacity of $\mathsf{OTM}_1, \ldots, \mathsf{OTM}_{L-1}$ is $2^{L-1} - 1$.

Consequently, the amortized cost of Insert is $\sum_{\ell=1}^{L} O(2^\ell \ell)/2^\ell + O(2^L L)/2^{L-1} = O(\log^2 N)$.

C.5 Proof of Lemma 19

Since OTM_i.Extract realizes $\mathcal{F}_{\mathsf{OTM}}$.Extract with probability 1, all real blocks of $\mathbf{T}_1, \ldots, \mathbf{T}_{\ell-1}$ are combined to $\mathbf{I}_\ell$ in line 3 of Algorithm 16. In addition, since the input block is also in $\mathbf{I}_\ell$, the Hierarchical Tree holds the input and all (non-dummy) blocks previously held in level 1 to $\ell - 1$ are correctly stored to level ℓ, relying on correctness of OTM.Build.

In Algorithm 16, all cells of level ℓ and higher are modified. Hence, by Lemma 17, Claim 16 is satisfied after Insert since it updates all nodes from level ℓ to 0 iteratively in line 6.

C.6 Proof of Lemma 20

Algorithm 17 requires L executions of OTM_i.Lookup and at most $\sum_{i=0}^{L}(L - i)$ executions of SelectMin. Thus, it costs $O(L^2) = O(\log^2 N)$ work.

C.7 Proof of Lemma 21

In Algorithm 17, since $\mathsf{OTM}_{\mathsf{lv}}$.Lookup realizes $\mathcal{F}_{\mathsf{OTM}}$.Lookup with probability 1, we can find and remove the target block (k, τ, v) of $\mathbf{T}_{\mathsf{lv}}[\mathsf{pos}]$, referred by the input $\mathsf{ref} = (k, \tau, \mathsf{pos}, \mathsf{lv})$, correctly. On the other hand, since the algorithm runs OTM_i.Lookup($\perp$) for all $i \neq \mathsf{lv}$, all other (non-dummy) blocks in the Hierarchical Tree remain in their place.

Moreover, in line 5 to 7 of Algorithm 17, at least all nodes on the path from $\mathsf{node}_{\mathsf{lv},\mathsf{pos}}$ (corresponding to the deleted $\mathbf{T}_{\mathsf{lv}}[\mathsf{pos}]$) to the root of MinTree are updated, thus Claim 16 maintains after Delete by Lemma 17[1].

C.8 Proof of Theorem 23

To prove the access-pattern obliviousness of PQ, consider an ideal-world simulator Sim described below.

[1]Note that we can ignore any *deletion of dummies* since it never affects to the priority order in MinTree.

$\underline{\mathsf{Sim}}$: At the beginning of the experiment $\mathbf{Ideal}^{\mathcal{A}}_{\mathcal{F}_{\mathsf{PQ}}}(N)$, Sim generates a Hierarchical Tree of capacity N in its memory, where all OTM_i are $\mathsf{empty}$and all nodes of $\mathsf{MinTree}$ hold $\mathsf{ref}_\perp$. Then, for each request of $\mathcal{A}$, Sim runs $\mathsf{PQ}(\mathsf{Insert}, \perp, \perp, \mathsf{ref}_\perp)$ regardless of $\mathcal{A}$'s query and returns the access pattern of that simulation.

From the definition of the challenger $\mathcal{C}$ in the real world, $\mathcal{C}$'s behavior is described as follows.
$\underline{\mathcal{C}}$: At the beginning of the experiment $\mathbf{Real}^{\mathcal{A}}_{\mathsf{PQ}}(N)$, $\mathcal{C}$ allocates a Hierarchical Tree of capacity N in its memory to perform PQ. The Hierarchical Tree is initialized as all OTM_i are $\mathsf{empty}$and all nodes of $\mathsf{MinTree}$ hold $\mathsf{ref}_\perp$. In addition, $\mathcal{C}$ maintains a *dictionary* (initialized as empty). This dictionary is used to store real-world references $\mathsf{ref} = (k, \tau, \mathsf{pos}, \mathsf{lv})$, and also to search for ref by τ. Note that this τ, representing the time at which a block (k, τ, v) was inserted, is regarded as an ideal-world reference. After this initialization, receiving query consisting of parts op, k, v, and τ from $\mathcal{A}$, $\mathcal{C}$ works as a man-in-the-middle between $\mathcal{A}$ and PQ as below:

- If $\mathsf{query} = (\mathsf{Insert}, k, v)$, $\mathcal{C}$ runs $\overrightarrow{\mathsf{ref}}, (\widetilde{k}, \widetilde{v}) \leftarrow \mathsf{PQ}(\mathsf{Insert}, k, v, \mathsf{ref}_\perp)$ and outputs the access pattern of that execution. Then, $\mathcal{C}$ updates the dictionary by all references in $\overrightarrow{\mathsf{ref}}$.
- If $\mathsf{query} = (\mathsf{ExtractMin})$, $\mathcal{C}$ runs $\overrightarrow{\mathsf{ref}}, (\widetilde{k}, \widetilde{v}) \leftarrow \mathsf{PQ}(\mathsf{ExtractMin}, \perp, \perp, \mathsf{ref}_\perp)$ and outputs the access pattern of that execution. Then, $\mathcal{C}$ updates the dictionary by all references in $\overrightarrow{\mathsf{ref}}$.
- If $\mathsf{query} = (\mathsf{Delete}, \tau)$, $\mathcal{C}$ searches the dictionary by τ and obtain $\mathsf{ref} = (k, \tau, \mathsf{pos}, \mathsf{lv})$ at first. When there is no reference corresponding to τ, $\mathcal{C}$ sets $\mathsf{ref} = \mathsf{ref}_\perp$. Then, $\mathcal{C}$ runs $\overrightarrow{\mathsf{ref}}, (\widetilde{k}, \widetilde{v}) \leftarrow \mathsf{PQ}(\mathsf{Delete}, \perp, \perp, \mathsf{ref})$ and outputs the access pattern of that execution. Finally, $\mathcal{C}$ updates the dictionary by all references in $\overrightarrow{\mathsf{ref}}$.
- If $\mathsf{query} = (\mathsf{ModKey}, k', \mathsf{ref})$, as same as the above Delete, $\mathcal{C}$ searches ref corresponding to τ at first. $\mathsf{ref} = \mathsf{ref}_\perp$ if there is no reference corresponding to τ. Then, $\mathcal{C}$ runs $\overrightarrow{\mathsf{ref}}, (\widetilde{k}, \widetilde{v}) \leftarrow \mathsf{PQ}(\mathsf{ModKey}, k', \perp, \mathsf{ref})$ and outputs the access pattern of that execution. Finally, $\mathcal{C}$ updates the dictionary by all references in $\overrightarrow{\mathsf{ref}}$.

To show the output of those Sim and $\mathcal{C}$ is identical, assume the following hybrid arguments.
$\underline{\mathsf{Hyb}_0}$: $\mathcal{A}$ receives the output of $\mathcal{C}$.
$\underline{\mathsf{Hyb}_1}$: The same as Hyb_0 except that $\mathcal{C}$ runs PQ while replacing all OTM_i with $\mathcal{F}_{\mathsf{OTM}}$ and $\mathsf{Sim}_{\mathsf{OTM}}$ as follows:

- In Insert;
  1. In line 3 of Algorithm 16, OTM_i.$\textsc{Extract}$ is replaced with $\mathcal{F}_{\mathsf{OTM}}$.$\textsc{Extract}$, and let the access pattern of this step be $\mathsf{Sim}_{\mathsf{OTM}}(\textsc{Extract}, 2^{i-1})$.
  2. In line 4, OTM_ℓ.$\textsc{Build}$ is replaced with $\mathcal{F}_{\mathsf{OTM}}$.$\textsc{Build}$, and let the access pattern of this step be $\mathsf{Sim}_{\mathsf{OTM}}(\textsc{Build}, 2^{\ell-1})$.
- In Delete;
  1. In line 3 of Algorithm 17, OTM_i.$\textsc{Lookup}$ is replaced with $\mathcal{F}_{\mathsf{OTM}}$.$\textsc{Lookup}$, and let the access pattern of this step be $\mathsf{Sim}_{\mathsf{OTM}}(\textsc{Lookup}, 2^{i-1})$.
  2. In line 5 to 7, let $\mathsf{node}_{i,\mathsf{pos}'_i}$ be determined by the above simulation $\mathsf{Sim}_{\mathsf{OTM}}(\textsc{Lookup}, 2^{i-1})$.

$\underline{\mathsf{Hyb}_2}$: $\mathcal{A}$ receives the output of $\mathsf{Sim}(N)$.

Claim 24. Hyb_0 *is identical to* Hyb_1.

Proof. Since PQ performs Insert after Delete in both Hyb_0 and Hyb_1, we can observe that all OTM_ℓ is deconstructed by $\textsc{Extract}$after $2^{\ell-1}$ executions of $\textsc{Lookup}$. Moreover, in Hyb_0, since any

(real) block once removed from OTM_ℓ is then assigned to another position and level (by ModKey) or completely erased from the Hierarchical Tree and the dictionary of $\mathcal{C}$ (by Delete), the same pos is never queried to OTM_ℓ.Lookup twice. Consequently, by Fact 5, all $\mathsf{Sim}_{\mathsf{OTM}}(\textsc{Build}, 2^{\ell-1})$, $\mathsf{Sim}_{\mathsf{OTM}}(\textsc{Extract}, 2^{\ell-1})$, and $\mathsf{Sim}_{\mathsf{OTM}}(\textsc{Lookup}, 2^{\ell-1})$ in Hyb_1 are identical to the access patterns of $\mathsf{OTM}_\ell.\textsc{Build}, \mathsf{OTM}_\ell.\textsc{Extract}$, and $\mathsf{OTM}_\ell.\textsc{Lookup}$ in Hyb_0. □

Claim 25. Hyb_1 *is identical to* Hyb_2.

Proof. For any input $(\mathsf{op}, k, v, \mathsf{ref})$, in Hyb_1, the access patterns provided by $\mathsf{Sim}_{\mathsf{OTM}}$ are independent from the input. Moreover, the part of Insert to update $\mathsf{MinTree}$ (line 6 of Algorithm 16) has the access pattern determined only by ℓ that corresponds to the number of insertions so far. On the other hand, the similar part of Delete (line 5 to 7 of Algorithm 17) has the access pattern determined by $\mathsf{Sim}_{\mathsf{OTM}}$. Therefore, for any input $\mathsf{query} = (\mathsf{op}, k, v, \mathsf{ref})$, the entire access pattern of $\mathsf{PQ}(\mathsf{query})$ is identical to that of $\mathsf{PQ}(\mathsf{Insert}, \perp, \perp, \perp)$. □

Consequently, there exists a simulator $\mathsf{Sim}(N)$ that simulates the access pattern of our PQ described in Algorithm 20.