論文 / 著書情報
Article / Book Information

題目(和文)	
Title(English)	Lightweight Offchain Protocols for UtxO Based Ledgers
著者(和文)	JourenkoMaxim
Author(English)	Maxim Jourenko
出典(和文)	学位:博士(理学), 学位授与機関:東京工業大学, 報告番号:甲第12061号, 授与年月日:2021年9月24日, 学位の種別:課程博士, 審査員:田中 圭介,伊東 利哉,尾形 わかは,鹿島 亮,首藤 一幸,森 立平
Citation(English)	Degree:Doctor (Science), Conferring organization: Tokyo Institute of Technology, Report number:甲第12061号, Conferred date:2021/9/24, Degree Type:Course doctor, Examiner:,,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

Lightweight Offchain Protocols for UTxO Based Ledgers

Maxim Jourenko
Academic Supervisor: Professor Keisuke Tanaka

May 31, 2021

Contents

Publications Included in this Work:

Lightweight virtual payment channels, Maxim Jourenko, Mario Larangeira, and Keisuke Tanaka, In *Cryptology and Network Security*, pages 365–384, Springer International Publishing, December 2020, doi:10.1007/978-3-030-65411-5_18

Payment trees: Low collateral payments for payment channel networks, Maxim Jourenko, Mario Larangeira, and Keisuke Tanaka, In *International Conference on Financial Cryptography and Data Security*. Springer, March 2021.

Chapter 1

Introduction

1.1 Introduction

Since the introduction of Bitcoin by Nakamoto [100] decentralized digital payment systems enjoyed great popularity. The technology behind Bitcoin, namely blockchains, inspired derivatives as well as new approaches to blockchain based payment systems. At the time of writing 8423 blockchains with a market capitalization of a little more than $1.371 trillion exist [1]. This popularity served to highlight scalability limitations of blockchain based systems. Transactions, e.g. payments, have to be processed through a consensus mechanism by a set of parties called *miner* to be included in the ledger. However, the number of transactions that can be processed by the consensus mechanism in any given time is limited which is necessary to ensure security of the system. Transaction issuers include a fees for the miners to incentivize that their transaction is processed by the consensus mechanism and included in the ledger. Moreover, this monetary fee additionally serves to incentivize miners to collaborate in the consensus mechanism and by doing this improve the system's security as a whole. However, in times of popular demand the number of transactions issued outpaces the number of transactions that can be processed through the consensus mechanism, which results in transaction issuer to include higher fees. For instance, the highest average transaction fee on any given day for Bitcoin equalled $55.27 [2].

Improving scalability of blockchain based systems is a research focus of the community. Possible solutions range from improving the network infrustructure [75] using dedicated server or improving the network protocol [3]. These proposals are commonly referred to Layer-0. Layer-1 approaches focus on adjusting the consensus protocol itself and a plethora of proposals have been introduced [56, 73, 52, 125, 22, 59, 107, 106, 134, 84, 77]. Lastly, Layer-2 proposals aim to reduce

[1] https://coinmarketcap.com
[2] https://ycharts.com/indicators/bitcoin_average_transaction_fee
[3] https://en.bitcoin.it/wiki/Stratum_mining_protocol

the number of transactions that are issued by creating shared accounts between parties, having these parties exchange transactions among themselves, and only submitting transactions that summarize the transaction history between the parties. These shared accounts are commonly called channels [44, 108, 112] and approaches to concatenate channels into payment channel networks (PCN) extend this approach. The arguably most prominent PCN is built for the Bitcoin blockchain and is called the Lightning Network [112].

Scalability proposals on Layer-2, commonly called offchain protocols, are particularly promising as they potentially reduce the number of transactions issued to the consensus mechanism by an arbitrary amount. While channels themselve allow two parties to perform an arbitrary amount of payments with another, methods as Hash Timelocked Contracts (HTLC) [112] allow parties to perform payments across a path of payment channels. Parties who join a PCN using a channel can potentially perform payments with any other party in the same PCN without issuing any transaction to the ledger. To our knowledge, as of now HTLCs are the only method to perform payments within the Lightning Network. It has been shown that HTLCs are vulnerable to denial of service and griefing attacks [99, 109]. Smart contracts, which are programs that are executed within the consensus mechanism for Ethereum, can be used to add features and improve operations on channels and PCNs. The payment protocol Sprites [98] allows for payments across PCNs while mitigating the impacts of denial of service and griefing attacks. Moreover Virtual State Channels [46, 49] use existing channels to allow for the creation of new channels in a PCN while not requiring any issuing of a transaction which is in contrast to regular channel constructions. While these works are important contributions to the family of Layer-2 protocols, requiring smart contract capability severely limits the employability of these protocols. The vast majority of blockchains, including Bitcoin, do not possess smart contract capability and as such are barred from utilizing these protocols. The Atomic Multi-Channel Updates (AMCU) protocol [51] attempts to close this gap by providing a payment protocol for PCNs, that are based on blockchains without smart contract capability, that mitigates denial of service and griefing attacks, but the protocol itself is shown to be vulnerable in the face of a malicious adversary.

This work aims to close this gap. Concretely, our contributions are as follows.

- We summarize the related work on Layer-2 scalability solutions and present a taxonomy.
- We present a novel framework to construct offchain protocols for ledgers based on the Unspend Transaction Output (UTxO) paradigm and without smart contract capability such as Bitcoin.
- We present the Channel Closure Attack. An attack on the AMCU protocol that allows a malicious adversary to steal funds from honest participants.
- We present the Lightweight Virtual Channel protocol that allows for the creation of virtual channels

- We prove security of Lightweight Virtual Channels in Canetti's Universal Composability (UC) Framework [29]
- We present the Payment Trees protocol; an payment protocol for PCNs that mitigates denial of service and griefing attacks.

Our work presents Layer-2 scalability solutions previously only available for blockchains with smart contract capability. This work reduces the gap between solutions that require smart contract capability and those that do not, and it shows resulting trade-offs.

1.2 Structure

In the following, first we give an overview of scalability solutions for blockchains with particular focus on Layer-2 protocols. Moreover, we structure the existing body of work and propose a taxonomy in Chapter 2. Afterwards, we formally define our model for blockchains operating under the UTxO paradigm, present our security model and introduce our framework for construction of offchain protocols in Chapter 3. Next, we introduce and discuss two protocols constructed with our framework, namely Lightweight Virtual Payment Channel in Chapter 4 and Payment Trees in Chapter 5. Lastly, we conclude and present possible directions for future work in Chapter 6.

Chapter 2

Background: Offchain Protocols

2.1 Introduction

Blockchains are behind the successful rebirth of digital cash from its first attempts [34, 35], firstly by Bitcoin [100] and now with several decentralized cryptocurrencies [52, 125, 59, 56, 132]. Although Bitcoin's relative success in offering worldwide payment alternatives to the more traditional mechanisms, like VISA Network [102] and Paypal [101], is undeniable, it still has a long way ahead in terms of handling a larger number of transactions.

The technical challenge of increasing the number of transactions per second (TPS) of a blockchain system is urgent, and it is closely related to the inner workings of the system itself. Namely, to its *consensus protocol*. As a more concrete example, we refer to the Bitcoin network [100] whose consensus protocol depends on the joint hash power of its nodes to perform the election of a block leader, i.e. the new block issuer, which is calibrated by design to happen every 10 minutes on average. In Proof-of-Stake (PoS) based systems, like Cardano/Ouroboros [56, 73] and EOS [52], analogous elections exist within a carefully designed (and strongly dependent on security guarantees) time slot for the generation of the new block. In order to confirm a transaction a minimum number of blocks has to be added to the ledger resulting in a *confirmation time*. The confirmation time, despite its central role in the security and stability of the system, imposes severe restrictions to the TPS rate of the overall platform. Alternatives had been suggested, such as alternative structures for blockchains [53, 122, 80, 121, 123], change in the size of a block [131], signature aggregation [24] or different consensus protocols [59, 36, 22, 107, 106]. It is common to divide these approaches in three cases:

- **Layer-0** (the network infra-structure): Here, optimization and special servers are employed to decrease the latency of the network to increase the TPS. An example of such approach is given by BloXroute [75];

- **Layer-1** (the consensus protocol): Arguably the layer which has more approaches in literature. There are different consensus architectures [56, 73, 52, 125, 22, 59, 107, 106], where a complete taxonomy can be found in [57], different data-structures for blockchains [122, 121, 123, 53], valid chain criteria [80], sharding [134, 84, 77] and federation [91];

- **Layer-2** (the off-chain channel/protocol): Protocols that perform minimal interaction with the blockchain. Typically for opening, paying, closing and disputing a channel. In the realm of cryptocurrencies, concrete examples are given by Decker and Wattenhofer [44], the Lighting Network (LN) [112], and others. This layer encompasses many more protocols and algorithms than the channel construction alone, as we review in later sections. One of the contributions of this work is to present a taxonomy, a coherent list of protocols, for different levels in the stack of protocols in this layer.

Layer-2 channel constructions have few names in literature: off-chain, *payment* or, yet, *state* channels, seemingly without consensus among the research community about their meaning. Here, in the face of the lack of standard terminology, we settle to name *payment channels*, constructions that do only store a distribution of coins among two parties, and *state channels* otherwise. Furthermore, we rely on the term *off-chain channel* when referring to either one regardless of its inner workings.

In a nutshell, a payment channel between two players, is when two participants decide to trade several transactions during a period of time, and in the end they, without access to smart-contracts, settle on a final balance based on the transactions exchanged, then the channel is closed. This transaction method is suitable for very small amounts, i.e. *micropayments*, which has a long history of research [130, 83, 96, 115, 54, 128, 103, 133]. A fairly complete survey, which includes centralized solutions, was done by Ali et al. [15]. More recently, micropayments were also studied by Pass and Shelat [105] in the setting of decentralized currencies, and for specific applications [65, 37]. The main advantage of such a setting is that all transactions made during the period of the channel do not need to be published in the blockchain, which is a clear advantage as concluded by McCorry et al. [93]. That is a set of transactions can be settled independently of the confirmation time of the blockchain system.

2.1.1 More than Pairwise Channels and State of Knowledge

The channel constructions themselves are building blocks into a stack of protocols (or algorithms). A simple payment/state channel only paves the way for exchanging funds, which is of limited use since the channel yields a *capacity*, i.e. the sum of funds initially deposited by the two players. A more interesting, and realistic, use is the concatenation of single channels into a *payment-channel network* (PCN). In this setting, a node A can send payment to C without creating

a specific channel from scratch with C, as long as both A and C are connected to a third node B, which relays transactions, typically by collecting fees.

A practical example of such a network is the already mentioned LN [112], which has multiple implementations [6, 9, 2, 110]. Although it extends the channel functionality, a lurking problem persists: *how to find suitable routes within the network*? The resemblance with theory of networks is inevitable, naturally similar problems appear. For example, a node sending a payment needs to find a route, similarly to routing problems in computer networks. On the other hand, payment networks also present differences, for example, the cost of the fees in a particular route. Yet, assuming the mentioned routing problem is solved, another one still exists: *How about the stability of the channel/network? Do all nodes need to be online during all the time of the channel? Is the capacity of each pairwise channel a bottleneck for the whole network?*

All those questions are legit and relevant, and there are more as we will see in later sections. Protocols and ideas to tackle them are currently being considered and studied by the research and practitioner community very often in an independent fashion without a comprehensive framework or compilation. Given that these ideas can be scattered through several journals and conference papers, as well as Internet forums and repositories, compiling them is a major task, let alone relate them to each other accordingly with their approaches and functionalities.

2.1.2 Our Contribution

We provide an extensive coverage payment and state channels, i.e. Layers-2 solutions for scalability, in cryptocurrencies and their related protocols. Namely the off-chain channel constructions, including probabilistic, simplex, and duplex, in addition with payment network constructions and network management protocols, i.e. the upper levels protocols that assume the existence of a network of channels. More concretely, we introduce a single framework (Table 2.1), which structures off-chain protocols therefore our contribution somewhat differs of the work in [62], which does not aim to structure the research literature in such a way. Furthermore, in comparison, we focus on the general description and the functionalities of the protocols.

A Taxonomy for the Layer-2 stack. Our single framework is a taxonomy with further classification for the three cited levels: off-chain construction, channel network, and network management. The levels compose a stack of protocols for Layer-2, as illustrated by Table 2.1 in Section 2.2. As expected, different levels and their respective set of functions define specific technical challenges, which by the best of our knowledge, were never organized in a comprehensive framework. A quick look on Table 2.1 reveals intense research on areas and few works in others, or even no works for specific problems. We believe these observations are of strong interest by the research community, which, otherwise, would have to collect these works through numerous internet forums and repos-

itories in addition to scientific journals, even to distinguish among very distinct functionalities, let alone more subtle differences of implementation.

2.2 A Taxonomy for Layer-2

Here we propose a taxonomy for the existing channel based protocols in Layer-2. We start by presenting an overview of our approach and justification for our choices. Later we present our classification illustrated by Table 2.1.

2.2.1 Overview

We organize the different sets of protocols in three levels [1]. Each level gathers related functions it performs, then we further subdivide the level into "functions". The intuition is that a protocol deployed on Layer-2, is expected to fulfill a certain function within a level.

In order to illustrate and justify this design, take for example the most popular Layer-2 PCN technique: HTLC [112]. Briefly, for the sake of example, HTLC allows LN nodes to establish connections between themselves in order to transfer funds. A major challenge in this setting is to find a suitable route among the nodes, thus the function required is *Routing*. A routing protocol for the LN fits into the "Routing" function within the "Network Management" level. Later, Table 2.1 summarizes our taxonomy and the levels we identify in the literature. However, first, consider the following levels:

- **Off-chain Channels:** Here are the constructions of the channels. In other words, how the nodes, relying on the transaction design of a cryptocurrency, can construct single pairwise-channels. These are the building blocks of the upper levels.

- **Network:** Here are the techniques employed to create the networks themselves. Typically by concatenating existing pairwise channels, or establishing a *network* of more than two nodes right from start.

- **Network Management:** This is the level for the protocols that maintain the channels and network of channels, and allow their efficient use while keeping them "alive" and usable.

2.2.2 Functions on Each Level

Given the earlier identified levels, in this section we further break them down into functions and provide a brief intuition behind the classification.

[1] We justify the choice of the term "level", instead of the more natural "layer", as an alternative to avoid ambiguity with the terms Layer-0, Layer-1 and Layer-2 which permeate this work.

Off-Chain Channels. The terminology in the literature is not standard. However it is possible to observe two big functionalities (1) make payments and (2) keeping state. The first is the simplest, since it does not require to keep a state in order to perform a payment, whereas the second is more general and it does require the aid of smart contracts (typically to support more complex operations). While in our classification (2) cannot be divided, therefore it represents the state channel variety, we further break down (1) into the three existing types: *Duplex*, *Simplex*, and *Probabilistic*.

Network. In order to establish PCN, special techniques are required, in addition to carrying out the payments. There are off-chain channels that do not support network construction capabilities, therefore this level depends on the capabilities of the channels. We further discuss this topic on Section 2.4.

Network Management. Over the network of channels, as further discussed in Section 2.5, several functionalities are required, and we identify the following main ones in the existing literature:

- **Routing:** The payments can be carried over several nodes, however it is necessary to select the set of nodes as the route, and several maybe be available.
- **Re-Balancing:** A difference from regular computer networks is that each pairwise channel has a capacity which may exhaust during the course of the exchange of transactions, and therefore it needs to be rebalanced.
- **Stability:** Another difference from regular computer networks, is that some constructions require the nodes to be online in order to perform the dispute phase of the protocols. Therefore there are protocols that mitigate or solve this requirement.
- **Anonymity/Privacy:** Information of nodes within a network, can be leaked as well as information about the performed transactions. Typically, the nodes in the middle of the channel can see the flow of funds.

A major difficulty on gathering the information on protocols for Layer-2 is that, in contrast to scientific community literature, several protocols had been proposed in forums and Internet repositories, and not in conference proceedings or journals. A unified compilation of issues and open problems is also, apparently, non existent. We address this with our suggested classification in Table 2.1, and further discuss them.

Secure multiparty computation. The reader should note that our definition of channel does not cover, for example, secure multiparty computation protocols, i.e. several users interact off-chain and provide *correctness proof* of the computation. Here, purposely we focus only in protocols that realize a channel between two nodes, and PCNs through concatenation of channels. In other

Level	Function	Protocols
Network Management **Section 2.5**	Routing **Section 2.5.2**	Silent Whispers [85] Speedy Murmur [118] Spider Routing [120] Flare Routing [113] Splitting Payments [111] Hoenisch and Weber [64] Atomic Multi-path [104]
	Re-Balancing Channels **Section 2.5.2**	REVIVE [69]
	Channel Stability **Section 2.5.2**	PISA [92] Avarikioti et al. [19]
	Anonymity/ Privacy **Section 2.5.2**	Fulgor & Rayo [86] Tumblebit [63]
Network **Section 2.4**	Construction **Section 2.4.2**	HTLC [112] Sprites [98] State Assertion [27] Virtual Channels [46] Counterfactual [38]
Off-chain Channels **Section 2.3**	State **Section 2.3.2**	Z-Channel [136] Perun [46] NoCUST [70]
	Duplex Payment **Section 2.3.2**	Raiden [7] Lightning [112] Decker et. al. [44] BOLT [61] Teechan [81] Burchert et al. [28] TumbleBit [63]
	Simplex Payment **Section 2.3.2**	Simplex [13] Dimitrienko et al. [45] Takahashi et al. [126]
	Probabilistic Payment **Section 2.3.2**	Pass et. al. [105] Hu and Zhang [65]

Table 2.1: The channels stack and concrete protocols.

words, we leave out of our classification protocols with multiuser distribution of funds, as used in [21, 41, 43, 42], given that this is a wide research area.

Network construction technique. The reader should note that Table 2.1 arranges LN [112] in the Off-chain Channels Level, not in the Network Level, and this can be considered unusual, given that LN is associated with PCN. We highlight that the construction technique used in both cases is the HTLC, also introduced in [112], therefore a more accurate setting is to locate the technique in the Network level instead. Moreover, this arrangement is consistent with the literature as in [94]. Similarly, we acknowledge that both [46] and [38] could be classified in the *Off-chain channel*.

Comparison on the body of work in each level. Briefly, given the similarities with computer networks, researchers realized that techniques can be

borrowed from currently known network algorithms, albeit the need to adapt them. This trend is promoted by Hoenisch and Weber [64], for *routing*, a crucial functionality within the *Network Management* Level. This trend contrasts with the technicalities of the *off-chain channel* and *network* levels. Our taxonomy decouples these different issues, and we further discuss them in Sections 2.3 (channels), 2.4 (networks) and 2.5 (management).

As expected the off-chain channels body of work, described in Section 2.3, presents a greater number of works, in comparison to, for instance, the *Network* level as described in Section 2.4. The reason is that most of the protocols in Section 2.4 derive from a single technique to realize *conditional payments*, the HTLC, which was recently fully formalized by Kiayias et. al [71]. An example of derived similar technique is Sprites [98] which employs a derived technique but with smart-contracts.

On the other hand, the amount of work on off-chain channels is comparable to that of the *Network Management* level, because of the significant amount of effort put on the functionality of *routing* among the nodes of the network by researchers and developers as later illustrated by Table 2.1 and also described in Section 2.5. It is easy to note that even in this level there are discrepancies on the attention given by the community on the function within the level, as the single work on re-balancing the network, i.e. the REVIVE protocol [69], shows.

Security and privacy preserving constructions. In order to illustrate the type of problems between the levels, we observe that privacy and security permeates the layers, and some problems are well known in the literature. For example, exchange of funds, in off-chain channels, needs to be consistent with the payments, i.e. the channel constructions should not allow parties to steal coins from each other, that is they should provide *balance security*, which is a property common to all safe channel constructions. Similar concerns exist for payment networks, but extended to the intermediate nodes, i.e. the nodes that allow the connection between two other nodes with not common channel. Furthermore, PCN should prevent information to be leaked to intermediate nodes. Finally, network management systems should provide several features such as routing for payers and payees, rebalancing and others while guaranteeing safety, in the sense of balance security, and preventing information leakage.

2.3 Off-Chain Channel Level

Before reviewing the constructions for off-chain channels, it is convenient to review the structure of transactions and how they can be issued on typical cryptocurrency. That clarifies for the reader how channels can be established.

2.3.1 Preliminaries: Transaction Design Review

Payments done on a blockchain are recorded in the form of *transactions*, as in Figure 2.5. Each transaction consists of n inputs and m outputs, $n \in \mathbb{N}$, $m \in$

$\mathbb{N}_0$. Each output consists of coins as well as a spending requirement. A plain payment has as requirement a signature with the private key of the payee. Inputs on the other hand consist of a reference to an output of a previous transaction on the blockchain (which was not referenced by another input before) as well as a witness for the spending requirement, e.g. a witness is a signature with the private key of the payee.

Spending requirements can be more complex than simply asking for the payee's signature. A relevant spending requirement for us is the k-out-of-l multisignature supported by Bitcoin with the opcode **CHECKMULTISIG** within its scripting language. A witness to such a requirement needs to contain signatures of k out of l specified parties. Channel construction techniques rely on the $k = l = 2$ spending requirement. A transaction can specify a *timelock* to enforce that miner will not include a transaction before a specified point in time. Timelocks can either be defined per input or for a whole transaction. Moreover timelocks can specify an absolute time or a point in time relative to when the referenced inputs were included in the blockchain. In Bitcoin an absolute timelock for a whole transaction can be specified with the field nTimelock, a relative timelock can be defined for each input using the field nSequence [1]Lastly, in the following we denote Δ as the maximum time required for a transaction to be committed by a party and subsequently be included in the blockchain, which is critical in the **Dispute** phase of the channel (outlined in Section 2.3.2).

2.3.2 Technical Challenges/Channel Constructions

Most of the interaction in a pairwise channel happens only between the players in the channel. Therefore, the players need to keep the balance through all the transactions performed in the channel, in other words the protocol should offer *balance security*. As we see next, very often, in order to have balance security, the nodes need to be online to take action in the case the partner in the channel misbehaves.

General Description.

The terminology is fuzzy since both terms, "payment" channel and "state" channel are used. Both terms refer to the idea of relying on off-chain direct communication between two players, i.e. a *channel*. That is, initially, both parties open the channel by committing a certain amount of coins, which will be the channel capacity or the maximum amount that is passed from one player to the other. The state of the channel can be tracked by simple signed transactions with or without the aid of *smart-contracts*. Channels utilizing smart-contracts have the capability to storage arbitrary state thus realizing state-channels.

The simplest form of channel relies on instructions for timelocks, e.g. the one described in BIP65 [23], and threshold signatures[2] using the early mentioned **CHECKMULTISIG** opcode. Typically channels have four phases:

[2]Often denoted *multisig* by the Bitcoin community.

- **Setup:** Two parties lock funds into the blockchain which the sum of funds is the capacity of the channel. It is important to note that the parties need to account for the confirmation time of the chain to start the channel.

- **Payment:** This phase the parties can exchange funds without interaction regardless of the confirmation time of the system. During this phase the initially locked funds cannot be used outside of the channel.

- **Dispute:** At any moment, a user can start a dispute in the case the other party do not follow the protocol.

- **Closing:** When two players decide to close the channel, they publish transactions in the blockchain, which reflect the exchange of funds that happened in the *Payment* Phase. Again, in this phase the players need to rely on the confirmation time of the blockchain. Furthermore, this phase can also trigger the **Dispute** Phase, whenever the parties do not agree on the current state.

Typical Channel.

The current balance of the channel is kept between the parties by directly exchanging mutually signed transactions with the amount being passed from one player to the other. When the channel is closed the last pair of exchanged transaction are committed to the blockchain, which makes the coins available to be redeemed by parties. A safety mechanism is in place by locally keeping the signed transactions representing the new state of the channel. When a player does not correctly perform the protocol, e.g. by not confirming a payment, thus the protocol enters in the *dispute* phase. During this phase, for [112], the partner can publish its locally kept (and mutually signed) transaction claiming the coins of the channel. Alternatively, in Decker and Wattenhofer [44], the channel is closed in the previous state. The consistency of all signed transactions are assured to be correct due to the use of timelock operation code in the transaction verification script language [23]. In comparison, *state channels* rely on smart-contracts since they offer more complex operations. A contract, by design, can keep track of the balance and transactions made into it. Therefore it can execute payments accordingly based on previously set conditions. In particular, the execution of a protocol can be triggered by presenting proofs of correctness on secure multiparty protocols [21, 43, 41, 78, 79]. More formally, given witness that a protocol was correctly performed, the contract can be executed.

Payment Channels.

Here we describe the main works on payment channels, from simplex, and probabilistic, to duplex constructions. Afterwards, we review state channels. Finally, we present constructions for privacy preserving channels.

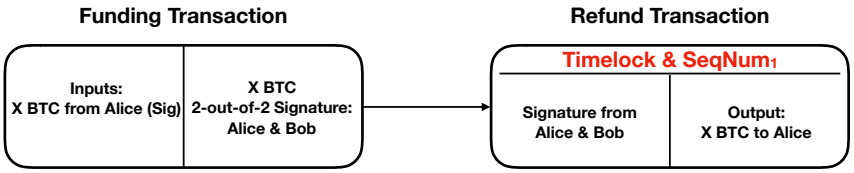


Figure 2.1: Setup

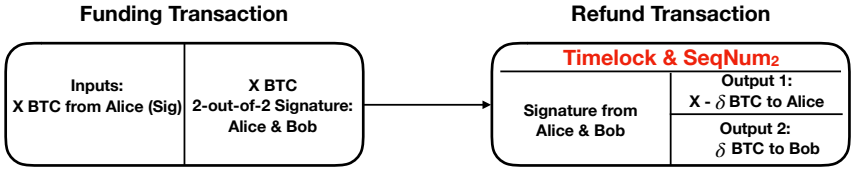


Figure 2.2: Payment

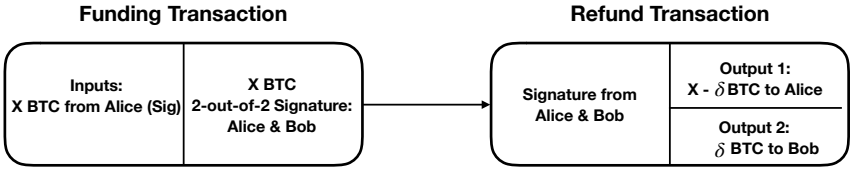


Figure 2.3: Updated version of payment

Figure 2.4: Figure 2.1 shows the initial state of a payment channel upon setup. Figures 2.2 and 2.3 show payments. The former makes use of sequence numbers and timelocks to replace previous payments. For example, for a channel between A and B, with capacity C_A and C_B, the coins committed, respectively, by A and B, after a transaction with value δ from A to B, therefore the new state is $C_A - \delta$, for A's balance and $C_B + \delta$ for B's balance. In 2013 this protocol was updated, as illustrated in Figure 2.3. This update removes the use of the sequence number. Moreover the timelock in the initial Refund Transaction is kept, however, updated Refund Transactions do not contain a timelock.

Simplex Payment. The earliest proposal of payment channels that we could find was posted on the Bitcoin Wiki [13] and is illustrated in Figure 2.4. Note that the first proposal mentioned here makes use of a sequence number to replace previous transactions, however, the respective, and already cited, Bitcoin field, nSequence has been disabled on 20th August 2010 [1, 13] [3].

Figure 2.4 illustrates a simplified form of the protocol in [13] between two parties: The payer and payee, and they initiate the *Setup Phase*. In order to do this the payer prepares two transactions as depicted in Figure 2.1. The first is the so called Funding Transaction which takes as input funds from the payer and spends them within one output that requires the signatures of both payer and payee to spend. The second is the Refund Transaction which spends the output

[3] https://github.com/bitcoin/bitcoin/commit/05454818dc7ed92f577a1a1ef6798049f17a52e7#diff-118fcbaaba162ba17933c7893247df3aR522

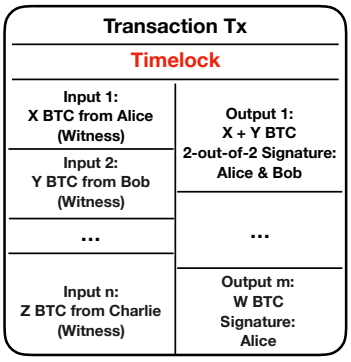


Figure 2.5: A simplified illustration of a transaction, with n inputs (left side) and m outputs (right side).

of the Funding Transaction and pays them back to the payer. Moreover the transaction has the sequence number set to 0, and a timelock in the near future by adjusting the value of nLocktime. Then, without signing the transactions, the payer sends both transactions to the payee who signs them and sends them back. After verifying the payee's signatures, the payer signs both transactions itself and sends them back. Then, one of the parties commits the Funding Transaction to the blockchain.

If the payer wants to pay a certain amount, it happens in the *Payment Phase*, it creates a new version of the Refund Transaction that still spends the output of the funding transaction but instead of giving all funds back to the payer, it contains an output that allocates some funds to the payee as depicted in Figure 2.2. Moreover, the sequence number is set to a higher value to replace the previous transaction. The payer signs this transaction and sends it to the payee. This process can be repeated for subsequent payments that all adjust the funds more and more in Payee's favor and increase the sequence number respectively.

Correctly keeping track of balances. The timelock and the sequence numbers prevent the payer to double-spending the funds. If the payer publish its Refund Transaction, created at Setup Phase which gives all funds back to the payer, to the blockchain, the payee could commit any Refund Transaction with a higher sequence number to replace Alice's committed transaction. However, the payee needs to commit his Refund Transaction before expiration of the timelock on payer's Refund Transaction. Since then the protocol has been adjusted on the Bitcoin-development mailing list [124] and the Bitcoin Wiki entry [13].

Probabilistic Simplex. Pass et al. [105] propose a probabilistic payment system to reduce the number of transactions on the blockchain and subsequently reduce the amount of transaction fees enabling micropayments. Moreover their solution provides near-instantaneous payments without requiring confirmation

delays. Their work is based on the work of Wheeler [129], Rivest [116] as well as subsequent work [82, 97], and proposes three protocols. The first is naive, therefore vulnerable, construction for two parties. The second and third protocols are similar, with both relying on verifiable third party with the difference that the latter is optimistic, and relies on the verifiable third party in case of a dispute. In these protocols the payer sets up an escrow address and put an amount a of coins into it. After the payment is performed the payer can potentially use the same escrow to pay another merchant after some time has passed. This work was improved by Hu and Zhang [65] using a time-locked deposit.

Duplex Payment. In a duplex payment channel both parties can allocate funds into their mutually shared channel and these funds can be redistributed between both parties arbitrarily. Both the constructions for duplex channels proposed by Decker et al. [44] and the LN [112] extend the idea of simplex payment channel[4]. That is, first, parties create a funding transactions in which both parties allocate their funds by paying coins into it. Moreover the funding transaction contains one output which requires signatures of both parties to spend. Second, the parties create a refund transaction which spends the funding transaction and has two outputs which pays back the coins to their respective parties. Note that each party needs to hold off signing the funding transaction until it holds a fully signed refund transaction since, otherwise, the other party might hold the funds in the funding transaction as hostage by committing the funding transaction and refusing to sign the refund transaction. After signing the refund transaction both parties sign the funding transaction and commit it to the blockchain to lock their funds into the channel, for the setup phase of the channel. Parties execute a payment by creating a new refund transaction and exchanging signatures. The payment is considered executed when both parties hold the new fully signed refund transaction and invalidate all previous refund transactions. If both parties cooperate for channel closure the parties sign and commit a closing transaction that spends the funding transaction and pays each party the amount of coins that are denoted in the most recent refund transaction.

Dispute Phase differences. The way previous refund transactions are invalidated differs in [44] and [112]. While in [44], the dispute is solved using timelocks, to enforce the most recent mutually agreed status, in [112] it is done by punishing the party by giving all channel coins to the honest party. Briefly, the former structures channels within a, so-called, invalidation tree. The funding transaction represents the tree's root and refund transactions are the leaves of the tree. Each payment adds a node in form of a transaction into the tree. There are two methods that enforce that only one path in the tree, and therefore only one refund transaction, is valid. For one, each node in the tree has a relative timelock to its parent. A node in the tree can invalidate all of its siblings by setting its timelock to be Δ less than the minimal timelock on all of

[4] A comparison can be found in [95].

its siblings. Note that refund transactions are always set to the highest locktime possible as agreed by both parties.

Channel factories. The general framework for channels imposes restrictions on the creation of channels. Namely, the confirmation time for the **Setup** Phase may still be unacceptable for micropayments. Furthermore, upon channel creation the funds are locked for only that channel. The relaxation in these requirements is the motivation for the creation of *channel factories* proposed by Burchert et al. [28]. The core idea is to introduce a layer between the blockchain and the payments. This translates into a step where a group of collaborators jointly fund a factory. This first step, still requires blockchain interaction, therefore still is subject to the limitations of the consensus algorithm. However any new pairwise channel can be created, among the initial users, from this point, upon communication between the collaborators, hence creating channels. Although the factory creation still requires time and funds locking into the blockchain, the advantage of this design is that it allows reallocation of funds between the pairwise channels.

Embedding script into signature. An interesting idea to preserve space in the blockchains, which is the case for major cryptocurrencies, is to embed more instructions into the signature issuing procedure by relying on a more sophisticated, i.e. one that offers more properties, signature scheme. That is the approach introduced by Poelstra [5, 10] denoted *scriptless script* for the Schnorr signatures and LN, which was recently extended by Malavolta et al. [88] to the more suitable, for cryptocurrencies, ECDSA signature scheme. Alternatively, [5] outlines an approach with the Schnorr signature scheme to embed the value of the pre-image x, i.e. for the HTLC hash challenge, into the signature computation algorithm, jointly carried by the payer and the payee of the channel. Although the formalization effort displayed in [88], the authors point out it is not the case for other proposals. Furthermore, main cryptocurrencies, e.g. Bitcoin and Ethereum, may not be compatible to the Schnorr scheme.

Privacy preserving channels. BOLT [61] is a channel construction focused on anonymity. Assuming either an anonymous blockchain such as Zcash [14] or anonymization techniques such as mixer, the BOLT protocol provides anonymity for one of both parties. A use case for this is a merchant that receives payments, however, has no means to learn the identity of the customer. A drawback is that the BOLT protocol requires the anonymous customer to be honest as they can otherwise block closure of the channel locking the merchant's coins indefinitely.

State Channels. Until now, in this work, the channels are built based on special instructions for transactions, as described in Section 2.3.1. Smart contracts, as in Ethereum, enable more complex offchain structures which has been investigated in [46, 98][5]. A particular use for smart-contracts aims to enforce

[5]We discuss [46] and [98] in more detail in Section 2.4.

fairness and correctness on distributed systems. The approach of relying in penalties to guarantee correctness and fairness of computation, in particular, on *secure multiparty computation protocols*, has been independently explored for efficient protocols on the top of blockchain. For the general case, this idea has been introduced by Andrychowics et al. [17, 16], and later by Bentov and Kumaresan [20]. For specific purposes, e.g. card-games, it was further explored by Bentov et al. [21] and David et al. [41, 43, 42].

Compared to the previously mentioned constructions for payment-channels, the state channel can be implemented more straight forwardly. They are opened by committing the respective smart contract onto the blockchain, whose state can be changed using a message signed by both parties as well as a sequence number. Parties change the state offchain by computing a message to the smart contract that would make transition into that state including a sequence number and exchanged signatures for this message but without committing the message onto the blockchain.

NoCUST [70] is another protocol which relies heavily on smart-contracts. Here two parties wishing to exchange small amounts create a channel by making their deposits into a smart-contract, therefore two on-chain operations. All the payments from that point on, are executed off-chain via a third trusted node which intermediates the off-chain operations between the two participants, and each payment requires issuing request payments and receipts. This design has the advantage, in comparison to regular payment hub over off-chain channels, of not requiring the hub node to allocate/lock a large amount of funds, hence *no custodian*, while intermediating payments between large group of pairs of nodes.

2.4 Network Level

In this level, the goal is to focus on how the channel network can be established, in contrast to how it is managed (which is the topic of Section 2.5). Concretely, how channels can be concatenated securely to carry payments through several nodes.

2.4.1 Technical Challenges

The main function of this level is the construction of a payment route. More concretely, how the pairwise "inner" channels can be concatenated and used to provide a medium for payments between two nodes that are more than one hop apart. Within this setting, it is important to keep, *balance security*, specially in the inner nodes of the route. Here the challenge is harder than in the early sections, because in the network cases, the nodes within the route can also collude against the honest players.

2.4.2 Construction Function

The payment networks and related concepts had been previously studied: trust networks [58, 67, 114], credit networks [40], path-based transaction (PBT) [119] and privacy in PCN [86]. Here we are interested in construction techniques for pairwise channels, in order to concatenate several channels into a *channel network*. Such networks can enhance even more the scalability of a system, because two nodes do not need to contact each other to open a channel. Instead, they can create a "virtual channel", it the sense of Dziembowski et al. [48] via a mutually connected node (more on this, later in the section). The most known technique is the early cited HTLC [112], which is currently being used in the Bitcoin network and paved the way to new propositions for different cryptocurrencies, as in the Raiden Network [7] for the Ethereum [132].

HTLC overview and conditional transfers. Nodes can execute a payment atomically on a set of channels using HTLC. A payment can be routed across a sequence of channels without payer and payee having to create a channel between themselves that would require committing transactions onto the blockchain. Briefly, in [112], two nodes exchange funds by relying on a middle node, say, B, and the two channels between A and B, and B and C. Node A wants to send coins to C, and, as the first step, C computes a hash value on an arbitrarily chosen random number x and shares it with A via a direct channel, however without establishing a payment channel. The actual transaction is carried by A sending the funds to B, using their mutual channel with the extra condition that node B shows the secret value x. Analogously, node B forwards the funds to C in a similar fashion, relying in their mutual channel. Apart from LN based channels [112], several types of channel constructions support HTLC: Raiden [7], Decker et. al. [44], and Perun [46]. Recently a full formalization of the HTLC technique was done by Kiayias et al. [71].

Mitigating Dispute Phase costs. In general, the channel networks are *optimistic*, i.e. they assume the nodes will cooperate, therefore the dispute phase, as outlined in Section 2.3.2, is expected to be rarely triggered. However this phase execution can be costly both in terms of time complexity and financially given that the arbitrage can involve the execution of smart-contract to compute penalties. The Sprites Protocol [98] introduced by Miler et al. is designed for Ethereum's smart contracts, and has the goal of reducing the time complexity of resolving a dispute. Note that this work is independently done in Raiden [7]. The main observation of [98] is that in case of dispute, all of the internal pairwise channels have to solve the dispute in its own respective HTLC agreement, i.e. between each of the two consecutive nodes. Hence, [98] substitutes HTLC values by a single Ethereum smart-contract which solves the dispute. That differs from the HTLC technique since there is no need to increment the time-lock parameter on each hop of the payment route.

A recent idea introduced by Buckland and McCorry [27] named *State Assertion Channels*, and it is analogous to [98] but for computational costs. The goal of [27] is to guarantee that during the Dispute Phase, which can also be triggered by closing a channel, the honest party always can be paid back regardless of the cost of performing the verification of the full state of the application in the blockchain deployed smart contract. What Sprites [98] does for the expiration time of channels, the State Assertion technique does for the cost of computation, which indeed can be significant if done on chain. The main setting for state assertion channels is to deploy two smart contracts, the application (AC) and the assertion (SC) ones, with the requirement that the latter has to receive the funds before the creation of the channel itself. The states are not disputed in the AC, but in SC which can pay back the honest user whenever it challenges an invalid state transition proposed by ta malicious user. The crucial observation is that SC verifies without the *full state* of the application, but, instead, with a digest of it via hash values. The work in [27] provides a simulation on this novel approach, and it is a first step in the rigorous formalization of the idea.

Generalizing State Channels: "Virtual Channels" and "Counterfactuals". There are approaches to construct more general structures over offchain-channels. More notably, we can find in the literature the works of Dziembowski et al. [48] and Coleman et al. [38]. The work in [48] extends their virtual state channel construction in [46] by enabling virtual payment channels exceeding one intermediary party. It is done by recursively applying their construction on top of state channel as well as a "virtual state channel", which allows the creation of a multi-hop state channel across a sequence of single-hop state channels. This technique allows payer and payee to operate on a shared state channel instead of having to setup a new HTLC instance for each individual payment. More concretely, consider the scenario where parties Alice, Bob and Charles, such that Alice and Charles, as well as Charles and Bob share a state channel respectively. For comparison, in [112] and [44], Alice and Bob can issue payments between each other relying on the existing infrastructure and without having to create a new channel between Alice and Bob. However, this requires that Charles joins the protocol for each executed payment between Alice and Bob. In contrast, virtual payment channels allow the creation of a payment channel on the top of the existing state channels, enabling payments between Alice and Bob without Charles' participation outside of virtual channel setup, closure and any dispute. The protocol is proven in the UC framework [29], and requires smart contract capabilities[6] and therefore it is not applicable to cryptocurrencies with limited smart contracts capabilities as Bitcoin.

In a sense, Coleman et al. [38] also aims to build a richer structure on the top of the off-chain channels. They introduced the notion of *counterfactual*, which is, in a nutshell, all the events of the channel, which can, or cannot, be committed to the blockchain. In this paradigm, a payment in the off-chain

[6] A proof-of-concept implemented for Ethereum is available at https://github.com/PERUNnetwork/Perun.

channel, changes a so-called *counterfactual state* of the system. In addition, it is also possible to create *counterfactual contracts* via commitments and signatures, which generalizes the channels even further. The work in [38] presents a framework, focused on practical implementation.

The Wormhole attack and improved HTLC. The attack introduced by Malavolta et al. [88] affects all the PCN constructions based on 2-step interaction between the nodes, therefore HTLC based PCNs are in general, vulnerable to it. The colluding nodes aim to capture the transaction fee which the inner nodes on a payment route would expect to receive in order to carry the payments in the network. Although the gravity of the attack, the authors of [88] succeeded in proposing a fix for the main protocol and even warned the LN team about the protocol weakness, which triggered new developments. In particular the implementation of the two party ECDSA construction from [88], and its later incorporation into the system [12]. Briefly, the attack is carried by two colluding nodes within the same payment route. The two nodes share the pre-image value of the HTLC, while keeping the nodes in between them oblivious to the payment condition value. Consequently, in the point of view of the nodes in the extremities of the path, the payment is carried out, while the nodes in the middle of the path see it as failed. This condition allows the colluding nodes to jointly receive the transactions fee from the nodes which did not executed the payment.

2.5 Network Management Level

This is the upmost level in our classification, and, arguably, this level offers a greater variety of technical issues in comparison to the other levels. Thus, before reviewing the existing protocols, we described the known technical problems.

2.5.1 Technical Challenges

Here we assume that a network of channels is established, and all the nodes have access to it. Therefore we identify the following main functions:

Routing.

Similarly to a regular computer network, in order the find a payment path, it is necessary to probe the network for nodes available to route the payments. In the case of channel networks, variables such as particular availability and *fees* can heavily influence routing. A close look into routing in the LN is given by Di Stasi et al. [60] and McCorry et al. [94]. A more general desiderata for routing over channel networks is given by Hoenisch and Weber [64] based on similarities between channels and Mobile Ad Hoc Networks (MANET).

Re-balancing route.

Each pairwise channel is associated with a channel capacity, which is how much funds the channel can handle. A PCN can concatenate several channels, i.e. different capacities, that need to harmoniously work together. The consequence is that inner channels in a path have its capacity exhausted forming bottlenecks in the whole route.

Stability of the route.

The current techniques for channel constructions very often rely on the assumption that the users of the channel, and therefore of a PCN, will be online during all the lifespan of the channel. This is crucial in the case a Dispute Phase, as described in Section 2.3.2, when the parties need to act timely. Failing to claim the correct state of the channel, lead to the honest user to lose funds.

Privacy and anonymity.

Intermediate nodes within a payment route, in principle, watch all the flow of transactions, since they are intermediating all the payments. Moreover, when enabling a payment path through, for example routing, information may be leaked about the payer and the payee.

2.5.2 Constructions

For each of the technical challenges detailed earlier, we now detail and discuss the existing approaches in the literature.

Routing.

Here the challenges are similar to computer networks, however there are important differences due to the payment network nature. For one, routing should be scalable, i.e. both the stored state per node as well as communication complexity when routing should be logarithmic on the number of nodes. Moreover, the amount of funds that can be routed through a path depends on the capacity of the channel with the lowest capacity on the path. Furthermore, nodes within a path can be either controlled by a malicious adversary or spontaneously go offline. Also the network should be able to handle multiple concurrent routing attempts and payments. In addition to all these challenges intermediate routes that participate in forwarding a payment can ask for a fee and therefore nodes need to consider this and might want to optimize for a reduction of payment fees. Nevertheless, we note that as a fallback nodes can always create a new payment channel between payer and payee, however, this counteracts the scalability efforts of offchain payment networks so most approaches attempt to accomplish routing without relying on the fallback method.

Approaches for routing. A common limitation for long payment channels is that the intermediate nodes need to be online during the transaction period. Therefore, often routing algorithms rely on *landmark routing techniques* [127], where a *landmark*, i.e. a node with extra guarantees regarding its connectivity. Briefly, there are two ways to approach routing in these networks: through *landmarks*, and *embeddings*. In *landmark routing* all nodes know the route to a set of *landmarks*. In such a protocol payments are, first, routed from the payer to a landmark and, second, routed from the landmark to the payee. This approach promotes centralization of payment channel networks. Nodes close to a landmark would have the advantage of being better connected and might need to pay less fees for routing as well as have a better routing success rate. They also might get more income through fees because payments would more often be routed through them compared to nodes far off from landmarks. Another approach is routing using *embeddings*, where nodes, among themselves decide, on an address space of the network and, then they can find routes using these addresses. This approach enables more decentralization however it requires more communication for maintaining routing tables.

SilentWhispers Protocol [85] proposes using landmark routing, where all landmarks are publicly known and semi-honest. All routing is done through these landmarks by computing the shortest routes. This is done in epochs such that node's routing tables can update to changes within the network. The work [85] also defines privacy notions for credit networks and provides security proofs of their construction in the UC framework. The work also proposes multiple extensions. One of these allows for malicious landmarks whereas the other extension proposes routing through nodes that are offline. In the latter case nodes create secret shares of their long-term private keys and distribute those across the landmarks who can then impersonate offline parties, however, this requires that at least one landmark is honest and does not try to reconstruct long-term private keys of which it holds secret shares. The author's of [85] evaluate their work using Ripple's [8] payment network topology from 2013 and 2016 [11]. They compare success ratio, path length of payments and message complexity between their approach, a protocol based on the Ford-Fulkerson max-flow algorithm [55], and SilentWhispers [85] among others. They show a better success ratio as well as smaller paths in average compared to SilentWhispers, albeit a lower success ratio than the one based on [55] which had an unfeasible message complexity.

Similarly, the SpeedyMurmurs routing protocol [118] is an embedding based routing protocol, for path-based transaction networks based on the work in [117]. In [118] some nodes are designated landmarks, and are used as roots of respective spanning trees are computed to create an address space within the network. A route can be computed using a distance function on nodes' addresses to find the next hop on route to the target. In comparison, LN/HTLC [112] relies on a *gossip protocol* for channel maintenance and recovery [7] and onion style

[7] A description can be found in https://github.com/lightningnetwork/lightning-rfc/blob/master/07-routing-gossip.md

routing for privacy [8]. However, it is not clear how nodes can find routes, and an attempt to tackle this is the routing proposal algorithm for LN: Flare [113]. The protocol in [113] is probabilistic, and proceeds in two ways: (1) each node stores the topology of its neighborhood, (2) each node chooses a set of beacon nodes globally and at random according to a uniform distribution. Routing between two nodes is done by first checking both peers neighborhoods for intersections, and if this does not work checking whether a beacon node is within the other peers neighborhood. The nodes continue by using the neighborhoods of a few beacon nodes when searching for mutually known nodes to route through.

Hoenisch and Weber [64] pioneered the adaption of techniques from Mobile Ad Hoc Networks (MANET), by compiling a list of requirements for off-chain channel networks. An interesting feature of their adapted protocol, named On-Demand Distance Vector (AODV), is that it takes into account the balance and fees of intermediate nodes, a feature not present in MANET but highly relevant given that these values can change arbitrarily and without coordination in the Layer-2 scenario.

Splitting payments. Routing success does not only rely on whether a route can be found or not, but also whether the route has enough capacity for the payment to be executed through it. Large payments have low routing success rates [9]. Approaches to tackle this issue aim to split payments through multiple routes as AMP [104] that provides a protocol to split a payment across multiple paths, Moreover [104] also considers privacy as well as sender/receiver anonymity. The Split Payment Protocol [111] tackles similar problem as in [104], however [111] allows for payments to complete only partially. Another approach is the Spider Network [120] which employs packet-based routing including congestion control to do payments through multiple paths. Each payment is split-up into smaller packets which can be routed through separate routes and claimed atomically by the sender as proposed in [104] or non-atomically. Nodes involved in routing payments can employ congestion control techniques when multiple payments are routed concurrently. Channel capacities and possible imbalances are considered when selecting paths for packages.

The work in [120] compares the success ratio of routing using an approach based on [55] as well as SpeedyMurmurs [118] and SilentWhispers [85]. They evaluated these approaches using network topologies existing in the Internet as found in [76], and Ripple's [8] payment network topology from 2013 [11] which has also been used for the work on SpeedyMurmurs [118], showing that the approach proposed in the paper works better on the former topologies than the latter ones where they are outperformed by the algorithm from [55]. Noteworthy is that they report significantly lower success ratios for SpeedyMurmur than have been reported in their paper [118]. Moreover, Speedy Murmurs does not consider privacy notions or malicious adversaries.

[8] A description can be found in https://github.com/lightningnetwork/lightning-rfc/blob/master/04-onion-routing.md

[9] Reported in https://diar.co/volume-2-issue-25/ and https://thenextweb.com /hardfork/2018/06/26/lighting-network-transactions/

Channel Re-balancing.

Given the limited capacity of a channel, it may be necessary to rebalance a set of payment channel to avoid that the channel turns into a simplex one because of exhausted capacity. For instance, let Alice and Charlie, Charlie and Bob as well as Charlie and Alice share a payment channel respectively. Moreover assume that initially everyone has one coin allocated as their funds for each channel. Now, if Alice sends one coin to Bob, Bob sends one coin to Charlie and Charlie sends one coin to Alice, all within their respective channels then we would end up in a situation where in each channel one party has two coins and the other has no coin. In this case we would not be able to repeat this payment without routing the payments over an intermediate node even though the total funds of all nodes remain unchanged. In order to circumvent the skewness in funds, the REVIVE protocol [69] relies on an untrusted third party that creates a block of transactions rebalancing funds. Each peer will lose money on one or more payment channels but gain an equal amount on others. After each peer verifies this on the rebalancing transactions they can apply the changes off-chain.

Channel Stability.

The channel between two nodes assume that both stay online on the duration of the channel. Any misbehaviour of any node, can trigger the response and respective claim of the partner node, which have to be online in order to perform the dispute phase of the network protocol. An approach to address this is to assume that the receiver can assign a *custodian node*, sometimes also referred as *watchtower* [18, 19], to watch over their payment channel while they are offline. Similarly, this is the idea of the PISA [92]. The custodian can dispute malicious commitments of the other peer while its customer is offline. The custodian gets a fee for the service and has a deposit that gets destroyed when it fails to make a dispute on behalf of its customer. However, the custodian's customer are not public and the custodian can use the same deposit as a safety for all customers. This leads to the problem that if there are enough customers the custodian can possibly get more funds by cheating on its customers than it would lose by having its deposit destroyed. Moreover, if the deposit is not destroyed and instead is payed out to all customers a malicious custodian can create customers for itself to mitigate the funds lost.

Privacy and Anonymity.

The network management layer may leak crucial pieces of information from nodes belonging to a route to the origin node of a payment. Malavolta et al. [86] investigate privacy notions on PCN and how to enforce them, and propose routing protocols Fulgor and Rayo [86] that consider concurrent routing processes and attempt to avoid deadlocks in a sense that at least one payment completes. Unfortunately their privacy notions rely on assumptions that the underlying routing protocol does not store the state (capacity) of payment

channels within the network, which if done could be used to circumvent their attempts for privacy.

Fees and node reputation. Since each node in the network can charge an arbitrary fee in order to conduct the payment along the path, an interesting question to investigate is the game-theory behavior of rational nodes within the network and other economic questions. In comparison to other works, as seen in previous transactions, these questions seem to be, to this day, largely understudied. A framework to study network topologies and economic aspects is given by Brânzei et al. [26]. Similarly, a reputation system, which would take account of fees and success payment rate, seems to be missing in the literature and are highly relevant for practical systems.

2.6 Final Remarks

We provided a major review, as shown in Table 2.1, of the body of work on off-chain channels, networks and related protocols for Layer-2 solutions, an emerging area targeting scalability of cryptocurrencies. We believe it provides a significant value for the community because it contains a wide overview of the landscape on the ideas and approaches present in the scientific literature and Internet repository and forums. We focused this work on providing a throughout overview of the landscape with emphasis on problems and approaches of the available protocols.

We highlight the rich literature on routing protocols, for Network Management Level, in comparison to, for instance, channel Re-Balancing protocols. That difference can be explained by the similarities of the off-chain channel and computer networks, which can be seen as a source of already tested ideas to find routes.

More recently, we can also note a trend on network construction protocols for the Network Level, in particular the works on Virtual Channels [46] and Counterfactual [38]. In both cases, they distinguish themselves by providing a richer framework for constructions on the top of off-chain channels. Furthermore, we also highlight that the most established technique, i.e. HTLC, was recently fully thoroughly studied [71].

Along the same lines, works on the economic aspects of the channels, i.e. fees charged by nodes, and *reputation* of nodes seem also to be underreported in the literature. In both cases, they are important, because they are critical to pave the way of more practical systems. Finally, it is important to emphasise that this work left out the topic of *interoperability* of systems, e.g. Cosmo [4], XClaim [135] and Comit [3].

Chapter 3

A Framework for UTxO Based Offchain Protocols

In this chapter, first we introduce the notation that we use throghout this work in section 3.1 we introduce the UTxO model for blockchains in Section 3.2 before introducing our framework for UTxO based offchain protocols in Section 3.3. Lastly we present our security model in Section 3.4.

3.1 Notation

Tuples. Throughout this work we make use of tuples and use short-hand notations as follows. Let $(a_1, a_2, \ldots, a_n)$ be a definition of a tuple of type A and let α be an instantiation of A. Then $\alpha.a_i$ equals the i-th entry of α.

Sets. Let $\cup$, $\cap$ and $\setminus$ denote set union, intersection, subtraction, and $\varnothing$ be the empty set.

3.2 The UTXO Paradigm

In the following we define the model of UTxO based ledgers.

Transaction Outputs. A UTXO is a tuple of the form (b, π) where $b \in \mathbb{N}$ is an amount of coins and $\pi \in \{0,1\}^*$ is a script. The b coins of the UTXO are claimed by providing a witness $w \in \{0,1\}^*$ s.t. $\pi(w) = \mathsf{True}$. The state of the ledger is represented by a set of UTXO S_{utxo}, which can be changed by a transaction of the form (U_{in}, U_{out}, t) where $t \in \mathbb{N}$ is the (absolute) timelock represented as a point in time, U_{out} is the list of unique UTXO for the *outputs* of the transaction, and U_{in} is the set of transaction *inputs* of the form $(\mathsf{ref}(u), w_u)$ where $\mathsf{ref}(u)$ is the pointer to the UTXO u, and w_u is the witness.

Transactions. A transaction (U_{in}, U_{out}, t) needs to fulfill the following conditions. (1) The locktime has passed, i.e. $t \leq \tau$ where τ is the current time, (2) all witnesses are valid, i.e. $\forall(\mathsf{ref}(u), w) \in U_{in} : u.\pi(w) = \mathsf{True}$ (3) the coins within the newly created UTXO are less or equal to those in the transaction's inputs, i.e. $\Sigma_{(\mathsf{ref}(u),w)\in U_{in}} u.b \geq \Sigma_{u\in U_{out}} u.b$, (4) all UTXOs in the transaction's inputs exist and have not yet been spent, i.e. $\forall(\mathsf{ref}(u), w) \in U_{in} : u \in S_{utxo}$. The transaction has the following effect on the ledger. All UTXOs referenced within U_{in} are removed from S_{utxo} and all UTXOs defined in U_{out} are added to S_{utxo}. A transaction T is included in the ledger within a duration $\Delta \in \mathbb{N}$. Condition (4) implies that no UTXO can be claimed by two different transactions. After sending T to the ledger, if within time Δ another transaction T' claiming a subset of the same UTXOs as T is sent to the ledger, it would result in a race condition, in which it is non-deterministic whether T or T' will change the ledger's state. We note that while we use Δ as a ledger parameter in practice this value has to be estimated for real-world implementations. Special care has to be taken when selecting a value. A value that is too low breaks our assumptions and the protocol's security. A value too high increases the collateral and therefore the impact of attacks such as congestion and lockdown[99, 109].

Scripting. Scripts in this work specify a UTXOs owner by requiring a signature of the transaction that spends the UTXO with the recipient's verification key. This is extended to 2-out-of-2 multisignatures that require verification keys of two parties $\mathcal{P}$ and $\mathcal{P}'$ effectively creating a shared wallet between both parties that can only be spent with consent of both parties.

Simplified UTXO. Throughout this work we simplify scripts by only stating the set of parties which need to provide their signatures to spend the respective UTXO, i.e. UTXO are tuples (b, Party), where $b \in \mathbb{N}$ is the amount of coins and Party is a set of parties. UTXOs requiring 2-out-of-2 multisignatures are termed $\mathsf{Funding\ UTXO}$. An instantiation of a Funding UTXO is $\mathsf{F\_UTXO}(x, \mathcal{P}_0, \mathcal{P}_1)$ which is of the form (x, $\{\mathcal{P}_0, \mathcal{P}_1\}$) where $x \in \mathbb{N}$ and $\mathcal{P}_0, \mathcal{P}_1$ are parties.

Partial Mappings. In the case of working with functionalities, we abstract away from transactions and represent them as partial mappings of UTXO of the form (In, Out) where $\mathsf{In}, \mathsf{Out}$ are UTXO that represent the transaction's inputs and outputs respectively. Informally a partial mapping consists of two sets of UTXO representing the inputs and outputs of a transaction. Submitting a partial mapping to a ledger results in a state change where the UTXO in the inputs are replaced by the in the outputs. We assume there is a function ϕ that takes a mapping (In, Out) and time t and outputs a respective transaction with timelock t. Analogously ϕ^{-1} is a function that takes a transaction and outputs a mapping and timelock.

3.3 The Framework for UTxO Ledgers

3.3.1 Offchain Protocols

Transaction Graphs. All transactions included in the ledger form a directed and acyclic graph. The set of all transactions form its vertices. An edge (T_0, T_1) from transaction T_0 to transaction T_1 exists, if T_1's inputs contain a pointer to one of T_0's outputs, i.e. $\exists u : u \in T_0.U_{out} \wedge (\mathsf{ref}(u), w) \in T_1.U_{in}$. Note that a transaction can only be included in a ledger if all of its ancestors have been included in the ledger before. In the remainder of this work we reference sets of transactions that are connected to form a sub-tree as *transaction trees*.

Channels. A channel γ between two parties consists of sub-protocols setup, closure and dispute. In setup both parties create a transaction Tr_f containing a Funding UTXO between each other which locks their funds into the channel. They create a transaction tree with the Funding UTXO as its ancestor that represents the channel which we reference in the remainder of this work as *channel-tree*. Only after the channel-tree is created and either party holds signatures of its transactions, both parties sign and commit Tr_f to the ledger while holding off commitment of transactions within the channel-tree. Both parties can perform closure of the channel by committing a transaction to the ledger that spends the Funding UTXO unlocking the channel's funds according to its most recent state. In case of a dispute, the dispute sub-protocol enforces the channel's state by committing the channel-tree's transactions onto the ledger.

Offchain Protocols perform a state transition of a channel by transforming its channel-tree. Any honest party must be able to enforce the new channel's state which might require an explicit invalidation step that disables commitment of an older version of the channel-tree or allows for punishment of a party that does so. An efficiency requirement of offchain protocols is that performing them $n \in \mathbb{N}$ times grows the channel-tree by at most $\mathcal{O}(1)$ transactions.

3.3.2 Operations on Transaction Trees

Updating a Transaction Tree. Timelocks can be used to define at which point a transaction can be committed to the ledger. Assume there are two transactions that spend the same UTXO, but which have timelocks that are 1) in the future and 2) have a difference of at least Δ. In this case parties can enforce commitment of the transaction with the lower timelock to the ledger. The transaction with the lower timelock *invalidates* the transaction with the higher timelock.

Constructing Transaction Trees Atomically. We observe that committing a transaction to the ledger requires that all of its ancestors are committed to the ledger beforehand. For a transaction to be able to be committed to the ledger it needs to contain all required witnesses, i.e. signatures. Therefore,

we can atomically create a transaction tree rooted in a transaction tr_{root} that is common ancestor to all other transactions. First, we add signatures to all transactions except tr_{root}. Afterwards, adding signatures to tr_{root} makes the whole transaction tree committable to the ledger at the same moment.

Constructing Transaction Trees Atomically Across Two Channels. We assume two transactions, tr_0 and tr_1, that require signatures of Alice and Bob, as well as Bob and Charlie respectively. Bob can enforce that both transactions are created atomically by only providing his signature *after* he received signatures from Alice and Charlie. We note that techniques (1) and (2) can be used in tandem.

3.4 The Security Model

Communication Model. We assume synchronous communication where time is split into communication rounds. If any party sends a message to a receiving party within a round, the message reaches the receiving party at the beginning of the following communication round. The duration of any round has an upper limit.

Indistinguishability. Let $negl(n)$ denote the negligible function. The standard definition for *computational indistinguishability* $\mathsf{X} \approx_c \mathsf{Y}$ is that there is no PPT algorithm D such that D can distinguish between two ensembles of probabilistic distributions $\mathsf{X} = \{X_n\}_{n\in\mathbb{N}}$ and $\mathsf{Y} = \{Y_n\}_{n\in\mathbb{N}}$, in other words $\Pr[D(X_n, 1^n) = 1] - \Pr[D(Y_n, 1^n) = 1]| \leq negl(n)$.

3.4.1 The Adversarial and Computational Model

We model the execution of our protocol π via the Universal Composability (UC) Framework with Global Setup by Canetti et al. [31] where all the entities are PPT Interactive Turing Machines (ITM), and the global setup is given by the global functionality $\mathcal{G}$, and the execution is controlled by the environment $\mathcal{Z}$. In this simulation based model, all parties from π have access to the auxiliary functionality $\mathcal{F}_{aux}$, i.e. $\pi^{\mathcal{F}_{aux}}$, in the hybrid world execution $\mathsf{HYBRID}_{\pi^{\mathcal{F}_{aux}},\mathcal{A},\mathcal{Z}}$ in the presence of the adversary $\mathcal{A}$ which can see and delay the messages within a communication round. Whereas the ideal execution, i.e. $\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$, is composed by the functionality $\mathcal{F}$ in the presence of the simulator $\mathcal{S}$. In both executions, the environment $\mathcal{Z}$ access the global functionality $\mathcal{G}$. Given the randomness r and input z, the environment $\mathcal{Z}$ drives both executions $\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$ and $\mathsf{HYBRID}_{\pi^{\mathcal{F}_{aux}},\mathcal{A},\mathcal{Z}}$, and output either 1 or 0. Therefore, let $\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$ and $\mathsf{HYBRID}_{\pi^{\mathcal{F}_{aux}},\mathcal{A},\mathcal{Z}}$ be respectively the ensembles $\{\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(n,z,r)\}_{n\in\mathbb{N},z\in\{0,1\}^*}$ and $\{\mathsf{HYBRID}_{\pi^{\mathcal{F}_{aux}},\mathcal{A},\mathcal{Z}}(n,z,r)\}_{n\in\mathbb{N},z\in\{0,1\}^*}$ of the outputs of $\mathcal{Z}$ for both executions. Thus, we say that $\pi^{\mathcal{F}_{aux}}$ *realizes* $\mathcal{F}$ *in the* $\mathcal{F}_{aux}$*-Hybrid model* when, there exist a PPT simulator $\mathcal{S}$, such that for all PPT $\mathcal{Z}$, we have $\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}} \approx_c \mathsf{HYBRID}_{\pi^{\mathcal{F}_{aux}},\mathcal{A},\mathcal{Z}}$.

Corruption. Assuming a protocol is executed between a set of $n \in \mathbb{N}$ mutually distinct parties. At the beginning of protocol execution, the adversary can statically corrupt up to $n - 1$ of n parties, receiving their internal state and having all communication to and from these parties be routed through the adversary. The adversary is malicious and can make any corrupted party deviate from a protocol.

Chapter 4

Lightweight Virtual Payment Channels

4.1 Introduction

Blockchains implement decentralized ledgers via consensus protocols run by mutually distrustful parties. Despite the novelty of such design, it has inherent limitations, for example, effectively all transactions committed to the ledger have to be validated by all parties. Croman et al. [39] showed that this severely limits a blockchain's throughput. Moreover, there is a minimal delay between submission of a transaction and verification thereof that is intrinsic to the system's security, e.g. one hour in the case of Bitcoin.

Layer-2 protocols, such as payment channel networks, allow confirmation of transactions outside the consensus protocol while using it as fallback. These protocols are referred as "off-chain" protocols in contrast to processing transactions via the consensus protocol "on-chain". An elementary protocol realizes *channels* and commonly works as follows: Two (or more) parties put together their funds and lock them on-chain by requiring a 2-out-of-2 (n-out-of-n) multisignature to claim them. Then these funds are spent by another transaction or a tree of transactions. These transactions represent the distribution of funds between both parties and are not committed to the blockchain except when parties enforce the fund distribution on-chain and unlock the funds. The parties can perform a payment, i.e. update the balance distribution within the channel, by recomputing that tree of transactions while invalidating previous transaction trees. Payments between parties are processed immediately and only involve interaction between the two parties. Channels can be extended to form channel networks by using Hashed Time Lock Contracts (HTLC) [44, 112]. Payments are performed by finding a path from payer to payee within the network and atomically replicating the payment on each channel along that path. A drawback of HTLCs is that a payment requires interaction with all intermediary nodes within a path. Virtual State Channels as proposed by Dziembowski et al.

[46, 49] devise a technique for creation of channels that allow execution of state machines instead of being limited to payments, and use an off-chain protocol that expands the network with new channels. The latter reduces the network's diameter yielding shorter payment paths, and allowing parties to perform payments without interacting with any intermediary nodes if they are adjacent in the now extended network. However, this construction requires blockchain with smart-contract capability, therefore not applicable to Bitcoin. Later we will see that this work addresses this limitation with a novel construction.

Use cases for virtual channels are manifold. A virtual payment channel provides the same benefits to the two parties sharing one as pairwise payment channels without the need to set it up by committing transactions to the ledger that can incur expensive fees. Payments can be executed offchain, without interaction with a third party and without incurring any fees, e.g. for routing an HTLC, making rapid micro-payments viable. This could enable new services such as a service-gateway. Such a gateway would consist of a node that sets up payment channels with different service provider that operate using micro-transactions, e.g. Video on Demand (VoD) services that bill by watch-time. A user could then create one payment channel with the gateway node and with the use of virtual channels created ad-hoc connections to the different (VoD) services instead of having to set up individual payment channels with each service they want to use. A more general use case is that virtual channels allow payment hubs, that have a high degree within a payment channel network, to interconnect their individual partners in exchange for a fee.

Related Work. HTLCs allow atomic payments across multiple hops. This is done by performing a conditional payment in each channel along a path from payer to payee. Executing a payment requires revealing a secret $x \in \mathbb{N}$ such that $H(x) = y$ where H is a cryptographic hash function. After setup, starting from the payee each node within the payment path reveals x to its predecessor. This proofs that the payment can be enforced on-chain which allows parties to resolve the payment by performing it off-chain. A timelock is used to cancel the transaction after a preset amount of time which unlocks the funds from the conditional payment. Although our construction can be used to enable payments across a payment channel network by creating a virtual channel between payer and payee, we argue that our work is orthogonal to HTLCs and both techniques can be used in tandem. First our construction is used to expand the underlying payment channel network with additional virtual channels and then HTLCs can be used to perform payments across this expanded infrastructure.

Dziembowski et al. introduced Virtual State Channels [46] and State Channel Networks [49]. A *state channel* depends on a smart contract previously committed to the blockchain. It contains (1) application specific code, and (2) code for state channel management. More specifically parties can send messages to the smart contract changing its state according to (1), or compute a state-transition message where the resulting state is computed by the parties and summarized in the state-transition message for (2). The state-transition

message can be kept off-chain, and only committed to the blockchain in case of parties' dispute. A virtual state channel can be built on top of two channels that were previously created in this manner. Similar to our work, virtual channels cannot be open indefinitely but have a *fixed* lifetime that is decided upon construction. In contrast to our work this technique requires a blockchain with smart-contract capability. Chakravarty et al. proposed Enhanced Unspent Transaction Outputs (EUTxO) [32] and constructed the Hydra Protocol [33]. EUTxO enables running constraint emitting state machines on top of a ledger which is used to setup a Hydra heads among a set of parties. This allows them to take their funds off-chain and confirm transactions with these funds among the participants of the Hydra head. Although parties can interact with each other using arbitrary transactions as they would on-chain, no new participants can be added to the Hydra head which is in contrast to payment channel networks. Moreover implementing Hydra requires blockchains with EUTxO capability limiting its applicability.

Our Contributions. This work proposes a new variant of Virtual Channels, we name it *Lightweight Virtual Payment Channels*, that is based on UTXO and requires only multisignatures and timelocks, that is, it does not require smart-contracts, yielding the first virtual channel construction implementable on blockchains such as Bitcoin, which currently has the highest market capitalization of all cryptocurrencies [1] and still is the most widely used, and blockchains operating with the recently introduced EUTxO [32] effectively improving the state of the art in both cases.

In a nutshell, our Layer-2 protocol for Virtual Payment Channels takes two payment channels between three parties as input, and opens three payment channels, i.e. one for each pair of parties. Our protocol can be applied iteratively allowing for virtual payment channels across multiple hops of the underlying payment channel network. Our construction (1) can be used to expand a payment channel network with virtual payment channels, (2) allows payments without interaction with intermediary nodes if payer and payee share a virtual payment channel, (3) can be used in tandem with HTLCs and (4) can be used with different payment channel implementations as Duplex Payment Channel [44], Lightning [112], Eltoo [108]. We formalize our work in Canetti's Universal Composability (UC) Framework [29] by introducing a functionality for lightweight virtual payment channels $\mathcal{F}_{\mathsf{LVPC},\mathcal{F}_{\mathsf{PWCH}}}$. Although formalizations for ledgers, including Bitcoin, within the UC framework exist [49, 46] we present the first global functionality $\mathcal{G}_{\mathsf{UTXO-Ledger}}$ for an Unspent Transaction Output (UTXO) based ledger. Moreover we present an auxilliary functionality $\mathcal{F}_{\mathsf{Script}}$ modeling a scripting language modelling access to *timelocks* and *multisignatures*. Our construction makes use of $\mathcal{G}_{\mathsf{CLOCK}}$ by Katz et al. [68], modified by Kiayias et al. [74, 72] and $\mathcal{F}_{\mathsf{SIG}}$ by Canetti et al. [30]. We present pseudo-code protocols Open_VC, Close_VC and Enforce_VC.

[1] https://coinmarketcap.com

Structure of this Work. In the remainder of this work, first, we briefly introduce notation and the model used in this work in Section **??**. Next we formalize a UTXO based ledger and their components in Section **??** and review pairwise payment channel in Section 4.2. Afterwards we give a high-level description and analysis of our approach in Section 4.3 before presenting pseudocode protocols in Section 4.4. Following this we formalize our approach in the UC framework by, first, introducing auxiliary functionalities in Section 4.5.4, then the pairwise payment channel functionality $\mathcal{F}_{\mathsf{PWCH}}$ in Section 4.6.3 and the virtual channel functionality $\mathcal{F}_{\mathsf{LVPC},\mathcal{F}_{\mathsf{PWCH}}}$ in Section 4.7. Next we introduce formal protocols implementing $\mathcal{F}_{\mathsf{PWCH}}$ and $\mathcal{F}_{\mathsf{LVPC},\mathcal{F}_{\mathsf{PWCH}}}$ namely protocols PWCH and $\mathsf{LVPC}^{\mathsf{v}}_{\mathsf{PWCH}}$ in Section 4.8.3 and Section 4.9 respectively. We provide simulation based proofs that the protocols implement the respective functionalities in Section 4.10. Lastly we discuss directions for future work in Section 4.11.

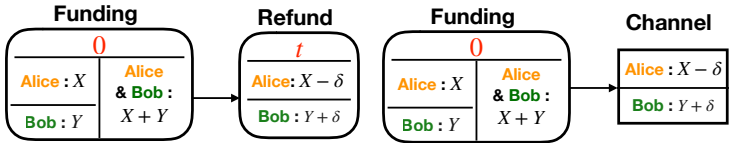
4.2 Pairwise Payment Channel

A pairwise payment channel allows two parties to exchange funds without committing a transaction to the ledger for the individual payments. Such a channel is setup by having parties commit a transaction on the ledger that collects some of each party's UTXO and spends all of it within a Funding UTXO. Committing this transaction on the ledger locks these funds. The Funding UTXO is spent by a transaction subtree representing the channel's state where committing it to the ledger unlocks and returns all of the parties' funds, however, instead, the parties hold off committing them. When executing a payment, they update the transaction subtree to represent the new state while invalidating the previous subtree. Invalidation can be done by spending the Funding UTXO with a transaction that has a timelock of at least Δ less than the previous subtree. We remark that alternative invalidation methods do exist [44, 108, 112]. The channel is closed by committing the transaction subtree or a transaction summarizing it onto the ledger.

4.2.1 Types of Transactions

We design our construction to be agnostic of the underlying pairwise payment channel construction, however, for the sake of having a complete formal treatment we formalize a simple pairwise payment channel construction based on timelocks. This construction consists of two types of transactions called Funding and Refund transactions.

Funding. A Funding transaction is parametrized with $(x, \mathcal{P}_0, \mathcal{P}_1)$ where $x \in \mathbb{N}$ is an amount of coins and $\mathcal{P}_0$, $\mathcal{P}_1$ are parties where $\mathcal{P}_0 \neq \mathcal{P}_1$. It is an arbitrary valid transaction Tr for which holds that there exists a Funding UTXO $\mathsf{f\_out} \in \mathsf{Tr.Out}$ parametrized with $(x, \mathcal{P}_0, \mathcal{P}_1)$ and its timelock is 0.



(a) Simple Payment Channel. (b) Abstract representation.

Figure 4.1: Figure 4.1a depicts a possible implementation of a simple pairwise payment channel whereas Figure 4.1b depicts an implementation independent abstraction.

Refund. A Refund transaction is parametrized with $(\mathsf{ref}, t_r, x_r, y_r)$ where ref is a reference to a Funding UTXO $\mathsf{f\_out} = \mathsf{F\_UTXO}(x_f, \mathcal{P}_0, \mathcal{P}_1)$, $t_r \in \mathbb{N}$ is a point in time, and $x_r, y_r \in \mathbb{N}$ are amounts of coins. It is a transaction Tr of form $(\{\mathsf{ref}\}, \mathsf{Out}, t_r)$ where $\mathsf{Out} = \{(x, \mathsf{f\_out}.\mathcal{P}_0), (y, \mathcal{P}_1)\}$, $x_f \geq x + y$. In the following we denote a Refund transaction with these parameters with $\mathsf{REFUND\_TR}(\mathsf{f\_out}, t_r, x_r, y_r)$ and an analogous mapping with $\mathsf{REFUND\_MAP}(\mathsf{f\_out}, x_r, y_r)$.

4.2.2 Pairwise Payment Channel

The implementation of a timelock-based pairwise payment channel is depicted in Figure 4.1a. It consists of a Funding transaction that locks both parties funds into the channel as well as a Refund transaction that holds the current state, i.e. fund distribution, of the channel. Two parties who want to create such a channel proceed as follows. (1) Create and exchange Funding and Refund transactions, (2) sign and exchange signatures of Refund transaction, (3) sign and commit Funding transaction to the blockchain.

As soon as the Funding transaction is included in the blockchain, a payment can be done by creating a copy of the Refund transaction with a new balance distribution, a timelock that is smaller by at least Δ to the previous Refund transaction's timelock, but higher than the current time, and exchanging signatures for it. The channel is closed by either party by committing the latest refund transaction to the blockchain at expiration of its timelock. Alternatively a channel can be closed by creating a Refund transaction with a timelock of 0 and committing it to the blockchain.

4.3 Overview of the Construction

The construction consists of three protocols, Open_VC, Close_VC and Enforce_VC used for setup, tear-down and dispute of virtual channels respectively. We remark that the executions of Open_VC and Close_VC require consent between all involved parties, and Enforce_VC can be executed by a party unilaterally.

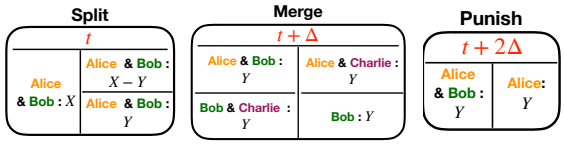


Figure 4.2: Illustrations of transactions used through out this work represented as nodes of a transaction graph. A transaction's inputs are listed on the left-hand-side whereas a transaction's outputs are on the right-hand-side. The value on top represents the transaction's timelock.

Types of Transactions. We use three types of transactions, they are are *Split*, *Merge* and *Punish* transactions as illustrated in Figure 4.2. A Split transaction spends a Funding UTXO and creates two new Funding UTXO between the same pair of parties. A Merge transaction takes two Funding UTXO between three parties and equal balance as input, and creates two UTXO with the same amount of funds as the inputs each. One is a Funding UTXO between the two parties that do not share a Funding UTXO within the inputs, and one is a UTXO that gives funds to the third party. Lastly the Punish transaction takes a Funding UTXO as input and creates an UTXO that gives it all to one of the parties.

Assumptions. Timelocks are used to invalidate transactions. That is, a transaction invalidates another one if it spends the same UTXO within its inputs, but has a timelock that is lower by at least Δ. We assume that the original payment channel between Alice and Bob has a timelock of at least $t_0 + \Delta$, and the one between Bob and Charlie has a timelock of at least $t_1 + \Delta$. After tear-down of our construction the timelocks of both channels will be $t_0 - \Delta$ and $t_1 - \Delta$ respectively. We note that this does not make the construction incompatible with pairwise payment channel constructions that do not rely on timelocks for transaction invalidation, such as lightning network style channels. Such channels can perform a state updates using their invalidation method that introduce a timelock before construction, and remove the timelock after tear-down.

Malicious Behavior. Parties abort protocols Open_VC and Close_VC when they observe another party deviating from the protocol, or if a party delays execution until expiration of the virtual channel, i.e. $t_0 - \Delta$ and $t_1 - \Delta$ respectively.

4.3.1 Intuition of the Protocols

Open_VC takes an amount of coins $\delta \in \mathbb{N}$ and two pairwise payment channel between three parties as input and creates three new pairwise payment channels, one between each pair of parties. In the following we assume the parties are Alice, Bob and Charlie with payment channels between Alice and Bob, and between Bob and Charlie. Our construction creates a set of transactions as

illustrated in Figure 4.3. In a nutshell, the purpose of the construction is to allow parties to enforce payout of all of their funds distributed among the offchain channels, while providing fall-back security of their funds in case all other parties misbehave.

First, two Split transactions are created, each spending one of the Funding UTXO that are spent by the original pairwise payment channels. Their timelocks are t_0 and t_1 respectively s.t. they invalidate the original payment channels. One of the UTXO of each Split transaction contains δ coins and is used as input into a Merge transaction. The other UTXO of each Split transaction is used as Funding UTXO to re-create the original payment channels, albeit each party has $\delta/2$ coins less in these channels. The Merge transaction takes the UTXO with δ coins as input, creates a Funding UTXO for a channel between Alice and Charlie where each possess initially $\delta/2$ coins, and another UTXO gives δ coins to only Bob which represents his collateral. Lastly, two Punish transactions spend the same UTXO as the Merge transaction but give all coins to Alice and Charlie respectively. They have a timelock of $\max(t_0, t_1) + 2\Delta$ such that they are invalidated by the Merge transaction.

Close_VC takes a virtual channel construction as input and closes them while setting up the original pairwise payment channel but with a balance distribution reflecting the balances in the three payment channel built on-top of the construction. Effectively Alice pays Bob the funds she owes Charlie while Bob forwards these funds to Charlie - and vice versa. The channels have timelocks $t_0 - \Delta$ and $t_1 - \Delta$ respectively to invalidate the Split transactions. Note that a virtual channel construction can only be closed until time $\min(t_0, t_1) - \Delta$ as otherwise the newly constructed payment channels cannot invalidate the Split transactions. Note that having Bob take out $\delta/2$ coins out of both of his original channels within the construction ensures that no party has a negative balance within a pairwise payment channel upon tear-down.

Enforce_VC lets a party enforce the current state by having it commit a transaction to the blockchain as soon as its timelock expires.

Atomic Construction. We require that all transactions within our construction are created and respectively invalidated atomically. This is enforced by the order in which transactions are signed. First, parties have to exchange signatures for all transactions except of those spending the original Funding UTXO, i.e. the Split transactions in Open_VC and the root of the pairwise payment channel sub-trees in Close_VC. Afterwards, Alice and Charlie sign these remaining transactions and send the signatures to Bob. Lastly Bob signs them and sends his signatures to Alice and Charlie. Only if a party holds all signatures for all transactions it is involved in, it will consent in performing payments. This ensures security as we will discuss in the following.

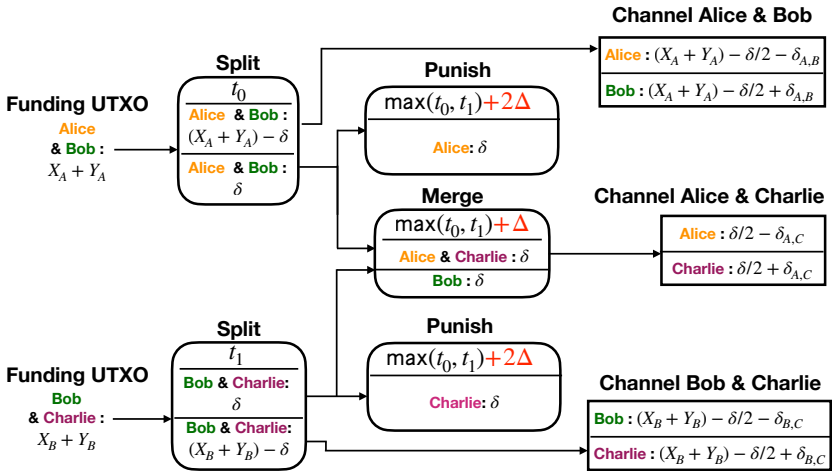


Figure 4.3: Overview of the virtual channel construction as a transaction tree. On the left-hand side are Funding UTXO either on the ledger or within previous virtual channels. Boxes with round corners represent the transactions of our construction while the boxes on the right-hand side abbreviate pairwise payment channel's transaction sub-trees. We omit stating inputs explicitly as they are clear from context.

Only Alice is honest. (1) As Bob is the last one to sign, he might interrupt the protocol before Alice receives a signature for the Split transaction. In this case Alice will not consent to any payments and the construction does not change her total balance. Alice can receive her funds by waiting for expiration of her original payment channel's timelock or commitment of the Split transaction by Bob. (2) Bob and Charlie can collude and spend the Funding UTXO that is referenced by their Split transaction. As such the whole transaction sub-tree with the Split transaction as root cannot be committed to the ledger, including the Merge transaction. In that case Alice can commit the Split transaction, and subsequently the Punish transaction. Alice will receive δ coins which is the maximum amount of coins she can receive within her pairwise payment channel with Charlie, as such she does not lose coins. Note that Alice's channel with Bob is unaffected as it is not within the sub-tree that Bob and Charlie invalidated.

4.3.2 Discussion of Attack Scenarios

Only Bob is honest. (1) As Bob is the last to sign transactions, he can assure either both Split transactions are fully signed and they can be committed to the ledger, or none. Moreover he can assure that either both Split transactions will be invalidated upon lockdown or none. (2) Spending the Funding UTXO referenced by the Split transactions always require Bob's consent by requiring a signature such that Alice and Charlie cannot invalidate any part of

the construction's transaction sub-tree, making Bob to pay out his collateral via a Punish transaction.

Iterative Construction. The pairwise payment channels used as input can either have a Funding UTXO located on a ledger, or a Funding UTXO created by a previous virtual channel construction. In that case timelocks have to be chosen such that within its transaction sub-tree any transaction has a timelock larger than its predecessor's timelock by at least Δ in order to ensure there is sufficient time to commit them to the ledger. Moreover virtual channel constructions have to be torn-down in reverse order in which they were setup. Iterative constructions requires further analysis of security. The key part to make iterative construction work is the design of the Punish transactions as they secure a party's funds, including potential collateral payments, while not over-punishing a potentially honest intermediary party: The punishment amount cannot exceed a party's collateral. Assume the channel between Bob and Charlie is created using a virtual channel construction with channels between Bob and Ingrid and between Ingrid and Charlie. In that case Ingrid and Charlie can collude by spending their Funding UTXO invalidating the Split transaction between Bob and Charlie making Bob have to pay coins within the Punish transaction between him and Alice. However, these funds as well as the funds Bob has in his channel between him and Charlie are covered by a Punish transaction he has between him and Charlie. Indeed this is the reason why the same amount of coins δ has to be paid into the Merge transaction from both of its Funding UTXO and only those coins are covered by the Punish transaction. This ensures that all funds are covered while not over-punishing the intermediary party in case of iterative virtual channel construction.

Mitigating Wormhole attacks. Malavolta et al. [89] showed an attack in which two colluding parties skip intermediary parties within a HTLC payment within a payment channel network (1) withholding fees that would have been paid to the intermediary parties and (2) obtaining the fees themselves instead. A variant of this attack could be applied to our construction as we do not require parties to verify that all pairwise payment channel but the ones they participate in were validly constructed. We discuss how to mitigate possible attacks. Although detailed discussion about payout of fees is beyond the scope of this work, we suggest that fees are paid to the intermediary party as compensation for locking up collateral. We note that due to this attacker cannot obtain more fees than they are owed (2). However, attackers could still collude to withhold fees of intermediary parties (1). A mitigation to this attack is that parties would need to proof that such a payment channel was previously constructed, but showing the Funding UTXO that were used and are located on the ledger as well as the whole transaction subtree originating those. A party that receives this information can do a sanity check and store the sub-tree in case they have to do the same proof. This poof serves to show that fees have been paid to the intermediate parties, however, we note that the information might be out-of-

date as malicious parties can close their pairwise channel effectively invalidating the whole subtrees.

4.4 Protocols

Here we informally introduce the constructions for Open_VC, Close_VC and Enforce_VC for setup, tear-down and dispute protocols of virtual channels. To help intuition these are heavily simplified but derived from the formal protocol $\mathsf{LVPC}_{\mathsf{PWCH}}$ in Section 4.9 that implements Functionality $\mathcal{F}_{\mathsf{LVPC},\mathcal{F}_{\mathsf{PWCH}}}$ from Section 4.7.

In the following protocols we assume that: When executing any protocol all involved honest parties check that execution with the given parameters is permissible, i.e. it will not result in transactions with negative balances, timelocks in the past and that the pairwise payment channel in Open_VC or the virtual channel in Close_VC are not currently in use with another virtual channel construction. Moreover, for protocols Open_VC and Close_VC they check that all parties consent execution. Lastly, they abort execution if they observe a party deviating from the protocol including when their signatures fail verification or when execution times-out. For details we refer to Functionality $\mathcal{F}_{\mathsf{LVPC},\mathcal{F}_{\mathsf{PWCH}}}$.

4.4.1 Types of Transaction

Before introducing the protocols, first we define the individual types of transactions used in our construction as well as pairwise payment channel and virtual payment channel.

Punish. A Punish transaction takes a Funding UTXO as input but gives all funds to one party. It is parametrized with (ref, $\mathcal{P}$, t_p) where ref is a reference to a Funding UTXO $\mathsf{f\_out}$, $\mathcal{P} \in \mathsf{f\_out.Party}$ is a party and $t_p \in \mathbb{N}$ is a round number. It is of form ($\{\mathsf{ref}\}$, $\{\mathsf{out}\}$, t_p) where $\mathsf{out} = (\mathsf{f\_out.b}, \mathcal{P})$. In the following we denote a Punish transaction with these parameter by $\mathsf{PUNISH\_TR}(\mathsf{f\_out}, \mathcal{P}, t_p)$, and an analogous mapping by $\mathsf{PUNISH\_MAP}(\mathsf{f\_out}, \mathcal{P})$.

Split. A Split transaction takes a Funding UTXO as input and splits funds across two funding UTXO. It is parametrized with (ref, δ, t_S) where ref is a reference to a Funding UTXO $\mathsf{f\_out}$, $\delta \in \mathbb{N}, \delta \leq \mathsf{f\_out.b}$ is a balance and $t_S \in \mathbb{N}$ is a point in time. It is of form ($\{\mathsf{ref}\}$, $\{\mathsf{out}_{ch}, \mathsf{out}_{\delta}\}$, t_S) where $\mathsf{out}_{ch} = (\mathsf{f\_out.}b - \delta, \mathsf{f\_out.Party})$ and $\mathsf{out}_{\delta} = (\delta, \mathsf{f\_out.Party})$. In the following we denote a Split transaction with these parameter by $\mathsf{SPLIT\_TR}(\mathsf{f\_out}, \delta, t_S)$ and an analogous mapping by $\mathsf{SPLIT\_MAP}(\mathsf{f\_out}, \delta)$. The routines $\mathsf{OUT\_CH}$ and $\mathsf{OUT\_DELTA}$ take either a Split transaction or analogous mapping as input and return $\mathsf{out_{ch}}$ and out_{δ} respectively.

Merge. A Merge transaction takes two funding UTXO by three parties and creates a new Funding UTXO. It is parametrized with (t_M, $\mathsf{f\_out}_A$, $\mathsf{f\_out}_B$, b)

where $t_M \in \mathbb{N}$ is a round number, $\mathsf{f\_out}_A$, $\mathsf{f\_out}_B$ are two Funding UTXO and $b \in \mathbb{N}, b = \mathsf{f\_out}_A.b = \mathsf{f\_out}_A.b$ is an amount of coins. Moreover for the involved parties $\mathcal{P}_A, \mathcal{P}_B, \mathcal{P}_C$ holds $\mathcal{P}_A, \mathcal{P}_B \in \mathsf{f\_out}_A.\mathsf{Party}, \mathcal{P}_B, \mathcal{P}_C \in \mathsf{f\_out}_B.\mathsf{Party}$. Given, $\mathsf{out_{ch}} = (b, \{\mathcal{P}_A, \mathcal{P}_C\})$ and $\mathsf{out}_B = (b, \{\mathcal{P}_B\})$, then a Merge transaction is of the form $(\{\mathsf{ref}(\mathsf{f\_out}_A), \mathsf{ref}(\mathsf{f\_out}_B)\}, \{\mathsf{out_{ch}}, \mathsf{out}_B\}, t_M)$. We denote a Merge transaction with these parameter by $\mathsf{MERGE\_TR}(\mathsf{f\_out}_A, \mathsf{f\_out}_B, b, t_M)$ and an analogous mapping by $\mathsf{MERGE\_MAP}(\mathsf{f\_out}_A, \mathsf{f\_out}_B, b)$. The routine $\mathsf{OUT\_CH}$ takes a Merge transaction or analogous mapping as input and returns $\mathsf{out_{ch}}$.

4.4.2 Definitions

Function $\mathsf{open\_virtual}(f, \mathcal{P}, \mathcal{P}', b, b', t)$ is used to open a pairwise payment channel with the provided, Funding UTXO f, between the two parties $\mathcal{P}, \mathcal{P}'$, respective balance distribution b, b' and optional timelock t. See Section 4.6.3 and Section 4.8.3 for details.

Definition 4.4.1. *A pairwise payment channel γ is a tuple of form $\gamma = (\mathsf{id}, f, \mathcal{P}_A, \mathcal{P}_B, b_A, b_B, t, t_0)$ where $\mathsf{id} \in \mathbb{N}$ is a unique identifier, f is a funding UTXO, $\mathcal{P}_A, \mathcal{P}_B$ are parties, $b_A, b_B \in \mathbb{N}$ are balances of $\mathcal{P}_A$, $\mathcal{P}_B$ respectively.*

Definition 4.4.2. *A lightweight virtual payment channel γ^v is a tuple of form $(\mathsf{id}, \gamma_0, \gamma_1, \gamma_{A,B}, \gamma_{B,C}, \gamma_{A,C}, \mathcal{P}_A, \mathcal{P}_B, \mathcal{P}_C, \delta, t)$ where $\mathcal{P}_A, \mathcal{P}_B, \mathcal{P}_C$ are three parties, γ_0, γ_1 are pairwise payment channel between $\mathcal{P}_A, \mathcal{P}_B$ and $\mathcal{P}_B, \mathcal{P}_C$ respectively provided as inputs, $\gamma_{A,B}$, $\gamma_{B,C}$, $\gamma_{A,C}$ are pairwise payment channel created by the construction between each pair of parties, δ is the capacity of channel $\gamma_{A,C}$ between $\mathcal{P}_A, \mathcal{P}_C$ and $t \in \mathbb{N}$ is a point in time until which the channel can be closed.*

For simplicity we omit stating id explicitly.

4.4.3 Protocols

We present opening of a virtual channel in Figure 4.4 and closure of a virtual channel in Figure 4.5. A protocol for resolving dispute is presented in Figure 4.6.

4.5 The UC Setting

While focusing on the intuition and readability of our approach up until this point, the remainder of this work is about formal treatment of our protocol in the UC framework including, potentially harder to read, but necessary detail. First we give an overview of the setting followed by introduction of all auxiliary functionalities used throughout this work. We follow up by detailing the pairwise payment channel functionality $\mathcal{F}_{\mathsf{PWCH}}$ the lightweight virtual channel functionality $\mathcal{F}_{\mathsf{LVPC}}$ in the following sections.

Algorithm 1 Open Virtual Channel

function OPEN_VC($\gamma_0, \gamma_1, \delta$)
  $\mathsf{tr}_{0,S} \leftarrow \mathsf{SPLIT\_TR}(\gamma_0.f,\ \delta,\ \gamma_0.t - \Delta)$
  $\mathsf{tr}_{1,S} \leftarrow \mathsf{SPLIT\_TR}(\gamma_1.f,\ \delta,\ \gamma_1.t - \Delta)$
  $\mathsf{tr}_{0,p} \leftarrow \mathsf{PUNISH\_TR}(\mathsf{OUT\_DELTA}(\mathsf{tr}_{0,S}),\ \mathcal{P}_A,\ \mathsf{max}(\gamma_0.t, \gamma_1.t) + \Delta)$
  $\mathsf{tr}_{1,p} \leftarrow \mathsf{PUNISH\_TR}(\mathsf{OUT\_DELTA}(\mathsf{tr}_{1,S}),\ \mathcal{P}_C,\ \mathsf{max}(\gamma_0.t, \gamma_1.t) + \Delta)$
  $\mathsf{tr}_{\mathsf{mrg}} \leftarrow \mathsf{MERGE\_TR}(\mathsf{OUT\_DELTA}(\mathsf{tr}_{0,S}),\ \mathsf{OUT\_DELTA}(\mathsf{tr}_{0,S}),\ \delta,\ \mathsf{max}(\gamma_0.t, \gamma_1.t))$
  $(\gamma_{A,B}, \mathsf{tr}_{\mathsf{root,A,B}}) \leftarrow \mathsf{open\_virtual}(\mathsf{OUT\_CH}(\mathsf{tr}_{0,S}),\ \mathcal{P}_A,\ \mathcal{P}_B,\ \mathsf{balance}(\gamma_0,\ \mathcal{P}_A) - \delta/2,\ \mathsf{balance}(\gamma_0, \mathcal{P}_B) - \delta/2)$
  $(\gamma_{B,C}, \mathsf{tr}_{\mathsf{root,B,C}}) \leftarrow \mathsf{open\_virtual}(\mathsf{OUT\_CH}(\mathsf{tr}_{1,S}), \mathcal{P}_B, \mathcal{P}_C,\ \mathsf{balance}(\gamma_1, \mathcal{P}_B) - \delta/2, \mathsf{balance}(\gamma_1, \mathcal{P}_C) - \delta/2)$
  $(\gamma_{A,C}, \mathsf{tr}_{\mathsf{root,A,C}}) \leftarrow \mathsf{open\_virtual}(\mathsf{OUT\_CH}(\mathsf{tr}_{\mathsf{mrg}}), \mathcal{P}_A, \mathcal{P}_C, \delta/2, \delta/2)$
  $\forall$ transactions except Split: Exchange signatures
  $\mathcal{P}_A$ and $\mathcal{P}_C$: send Split transactions' signatures to $\mathcal{P}_B$
  $\mathcal{P}_B$: Send Split transactions' signatures to $\mathcal{P}_A$ and $\mathcal{P}_C$
  return $\gamma^v = (\gamma_0, \gamma_1, \gamma_{A,B}, \gamma_{B,C}, \gamma_{A,C}, \mathcal{P}_A, \mathcal{P}_B, \mathcal{P}_C, \delta, \min(\gamma_0.t, \gamma_1.t) - 2\Delta)$
end function

Figure 4.4: Creation of a virtual channel. Takes two pairwise payment channel γ_0 and γ_1, and an amount of coins δ as input, and outputs a virtual channel γ^v.

Algorithm 2 Close Virtual Channel

function CLOSE_VC(γ^v)
  $\mathsf{sum}_A = \gamma^v.\gamma_{A,B}.b_A + \gamma^v.\gamma_{A,C}.b_A$
  $\mathsf{sum}_B = \gamma^v.\gamma_{A,B}.b_B + \gamma^v.\gamma_{A,C}.b_B$
  $\mathsf{sum}'_B = \gamma^v.\gamma_{B,C}.b_A + \gamma^v.\gamma_{A,C}.b_A$
  $\mathsf{sum}_C = \gamma^v.\gamma_{B,C}.b_B + \gamma^v.\gamma_{A,C}.b_B$
  $(\gamma_0, \mathsf{tr}_{\mathsf{root,A,B}}) \leftarrow \mathsf{open\_virtual}(\gamma^v.\gamma_0.f, \mathcal{P}_A, \mathcal{P}_B, \mathsf{sum}_A, \mathsf{sum}_B, \gamma^v.t)$
  $(\gamma_1, \mathsf{tr}_{\mathsf{root,B,C}}) \leftarrow \mathsf{open\_virtual}(\gamma^v.\gamma_1.f, \mathcal{P}_B, \mathcal{P}_C, \mathsf{sum}'_B, \mathsf{sum}_C, \gamma^v.t)$
  $\forall$ transactions except $\mathsf{tr}_{\mathsf{root,A,B}}$ and $\mathsf{tr}_{\mathsf{root,B,C}}$: Exchange signatures
  $\mathcal{P}_A$ signs $\mathsf{tr}_{\mathsf{root,A,B}}$, $\mathcal{P}_C$ signs $\mathsf{tr}_{\mathsf{root,B,C}}$. Send signatures to $\mathcal{P}_B$
  $\mathcal{P}_B$ signs $\mathsf{tr}_{\mathsf{root,A,B}}$ and $\mathsf{tr}_{\mathsf{root,B,C}}$. Sends signatures to $\mathcal{P}_A$ and $\mathcal{P}_C$ respectively
  return (γ_0, γ_1)
end function

Figure 4.5: Closing of a virtual channel γ^v by recreating the original channels γ_0 and γ_1. The constructions Split transactions are invalidated by having the roots of the pairwise payment channels have timelocks of at most $\gamma^v.t$.

Algorithm 3 Enforce Virtual Channel

```
1: function ENFORCE_VC(γ^v)
2:     for all tr in transactions of γ^v do
3:         if tr.t < τ ∧ ∀o ∈ tr.In : o is on the ledger then
4:             Commit tr to the ledger
5:         end if
6:     end for
7: end function
```

Figure 4.6: Parties enforce the state presented by the virtual payment channel construction by committing transactions to the ledger whenever possible, i.e. as soon as their timelocks expire and UTXO referenced in their inputs are present on the ledger.

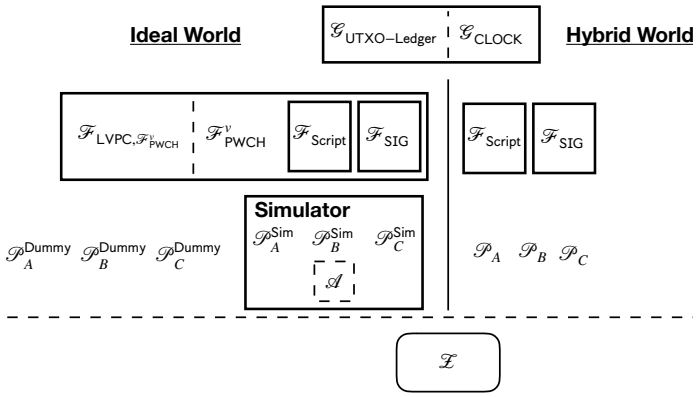


Figure 4.7: Overview of our setup within the UC framework.

Overview. Figure 4.7 depicts an overview of our construction. The setting is split up in an Ideal world and a ($\mathcal{G}_{\mathsf{CLOCK}}$, $\mathcal{G}_{\mathsf{UTXO-Ledger}}$, $\mathcal{F}_{\mathsf{SIG}}$, $\mathcal{F}_{\mathsf{Script}}$) - hybrid world. The global functionality $\mathcal{G}_{\mathsf{UTXO-Ledger}}$ is associated with the global $\mathcal{G}_{\mathsf{CLOCK}}$ functionality and accessible from either world. The lightweight virtual channel functionality $\mathcal{F}_{\mathsf{LVPC}}$ is associated with the pairwise payment channel functionality $\mathcal{F}_{\mathsf{PWCH}}$ receiving access to its internal state and helper functions. $\mathcal{F}_{\mathsf{PWCH}}$ includes and replicates the interfaces and behavior of $\mathcal{F}_{\mathsf{SIG}}$, $\mathcal{F}_{\mathsf{Script}}$.

4.5.1 The Global Clock Functionality $\mathcal{G}_{\mathsf{CLOCK}}$.

We adapt the global clock functionality formalized by Katz et al. [68], modified by Kiayias et al. [74, 72] and is depicted in Figure 4.5.1. The functionality keeps track of a round number τ that can be read by any party. After finishing computations a party sends a clock update request to the functionality. The round number is incremented after the functionality receives update requests from all parties as well as the ledger functionality. Parties agree upon a starting time of their protocol as well as a duration for each round, such that time can be derived from the round number.

Functionality $\mathcal{G}_{\mathsf{CLOCK}}$

The functionality is accessible by any entity and associated with a global functionality $\mathcal{G}_{\mathsf{UTXO\_Ledger}}$.

State: Stores time $\tau \in \mathbb{N}$, a set of parties P, bit $d_{\mathcal{G}_{\mathsf{UTXO\_Ledger}}} \in \{0,1\}$ as well as bits $d_{\mathcal{P}}$ for each party in P.

Initialization: Sets $\tau = d_{\mathcal{G}_{\mathsf{UTXO\_Ledger}}} = 0$ and $\mathsf{P} = \varnothing$.

Register: Upon receiving message $(\mathsf{register}, sid)$ from party $\mathcal{P}$, set $\mathsf{P} = \mathsf{P} \cup \{\mathcal{P}\}$, store a bit $d_{\mathcal{P}} \in \{0,1\}$ initialized with $d_{\mathcal{P}} = 0$ and send message $(\text{REGISTER}, sid, \mathcal{P})$ to the adversary.

Clock Update Ledger: Upon receiving message $(\mathsf{clock\text{-}update}, sid)$ from $\mathcal{G}_{\mathsf{UTXO\_Ledger}}$ set $d_{\mathcal{G}_{\mathsf{UTXO\_Ledger}}} = 1$ and send message $(\text{CLOCK-UPDATE}, sid, \mathcal{P})$ to the adversary.

Clock Update Party: Upon receiving message $(\mathsf{clock\text{-}update}, sid)$ from party $\mathcal{P}$ set $d_{\mathcal{P}} = 1$. If $d_{\mathcal{G}_{\mathsf{UTXO\_Ledger}}} = 1$ and $d_{\mathcal{P}} = 1$ for all honest parties in P, set $\tau = \tau + 1$, $d_{\mathcal{G}_{\mathsf{UTXO\_Ledger}}} = 0$ and $d_{\mathcal{P}} = 0$ for all honest parties in P. Lastly send message $(\text{CLOCK-UPDATE}, sid, \mathcal{G}_{\mathsf{UTXO\_Ledger}})$ to the adversary.

Clock Read: Upon receiving message $(\mathsf{clock\text{-}read}, sid)$ from any entity reply with message $(\text{CLOCK-READ}, sid, \tau)$.

4.5.2 The Global Functionality $\mathcal{G}_{\mathsf{UTXO-Ledger}}$

models a UTXO based ledger maintaining a publicly readable set of UTXO.

The ledger maintains a set U that holds all UTXO. The interface $\mathsf{Transaction}$ is used to modify U by providing a UTXO mapping. It can be called by $\mathcal{Z}$ modeling transactions done by parties outside the protocol, however, parties themselves are only able to change U indirectly by interacting with the $\mathcal{F}_{\mathsf{Script}}$ functionality. When receiving a request the functionality checks that all coins

within the Outputs of the mapping are covered by the coins referenced in the Inputs. Any party can read U by calling the Check UTXO sub-function.

The differences between $\mathcal{G}_{\mathsf{UTXO-Ledger}}$ and the ledger functionality by Kiayias et al. [74] are twofold. For one instead of using a verification predicate to check the validity of transactions, we move this verification into a second functionality $\mathcal{F}_{\mathsf{Script}}$ representing required parts of a blockchains scripting language similar to the separation of ledger and smart contract functionalities in the work of Dziembowski et al. [49, 46]. For another we explicitly make use of UTXO as required in our construction.

Functionality $\mathcal{G}_{\mathsf{UTXO-Ledger}}$

State: Stores set of UTXO U.
Initialization: $\mathcal{Z}$ sends the initial state U_0. Sets $\mathsf{U} := U_0$.
Additional interface: The functionality wraps the $\mathcal{F}_{Script}$ and $\mathcal{F}_{\mathsf{SIG}}$ functionalities internally and replicates their interface. Any messages to these functionalities are processed according to their definition.
Transaction: Upon receiving, from either $\mathcal{Z}$ or functionalities, the message $(\text{TRANSACTION}, sid, M)$ where M is a partial UTXO mapping, do : Let $(\mathsf{In}, \mathsf{Out}) = M$. Check that $\mathsf{In} \subseteq \mathsf{U}$, $\sum_{i \in \mathsf{In}} i.b \geq \sum_{o \in \mathsf{Out}} o.b$. Upon success within Δ rounds set $\mathsf{U} = (\mathsf{U} \setminus \mathsf{In}) \cup \mathsf{Out}$.
Check UTXO: Upon receiving $(\text{CHECK}, sid, \mathsf{out})$ where $\mathsf{out} \in \mathsf{Output}$ reply $(\text{CHECK\_OKAY}, sid, out)$ if $\mathsf{out} \in \mathsf{U}$ and $(\text{CHECK\_FAILURE}, sid, out)$ otherwise.

4.5.3 The signature functionality $\mathcal{F}_{\mathsf{SIG}}$

by Canetti et. al. [30] as depicted in Figure 4.5.3 provides access to signature generation and verification as well as facilities to create verification keys.

Functionality $\mathcal{F}_{\mathsf{SIG}}$

State: Stores set K which contain tuples of form $(\mathcal{P}, v)$ where $\mathcal{P}$ is a party and v is a verification key. Set S with entries of form (m, σ, v, b) where m is a message, σ as signature, v a verification key and $b \in \{0, 1\}$.
Key Generation: Upon receiving message (KeyGen, sid) from party $\mathcal{P}$ verify that $sid = (\mathcal{P}, sid')$ for some sid'. In that case hand (KeyGen, sid) to the adversary. Upon receiving $(\text{VERIFICATIONKEY}, sid, v)$ from the adversary, forward the message to $\mathcal{P}$ and store $(\mathcal{P}, v)$ in K.
Signature Generation: Upon receiving message (SIGN, sid, m) from party $\mathcal{P}$ verify that $sid = (\mathcal{P}, sid')$ for some sid'. If that is true, send (SIGN, sid, m) to the adversary. Upon receiving $(\text{SIGNATURE}, sid, m, \sigma)$ from the adversary, if $(m, \sigma, v, 0) \notin S$ send an error message to $\mathcal{P}$ and halt. Otherwise store $(m, \sigma, v, 0)$ in S and send $(\text{SIGNATURE}, sid, m, \sigma)$ to $\mathcal{P}$.

Signature Verification: Upon receiving message (VERIFY, sid, m, σ, v') from a party $\mathcal{P}$ forward it to the adversary. Upon receiving (VERIFIED, sid, m, ϕ) from the adversary do:

1. If $v' = v$ and $(m, \sigma, v, 1) \in S$ set $f = 1$.
2. Else if $v' = v$, $(m, \sigma', v, 1) \notin S$ for any σ' and $\mathcal{P}$ is not corrupted by the adversary, store $(m, \sigma, v, 0)$ in S and set $f = 0$.
3. Else if $(m, \sigma, v', f') \in S$ for any v', f' set $f = f'$.
4. Else store (m, σ, v', ϕ) in S and set $f = \phi$.

Send (VERIFIED, sid, m, f) to $\mathcal{P}$.

4.5.4 The Script functionality

represents the elements of a blockchain's scripting language we need to enable our construction. Parties interact with it using the Transaction interface providing a transaction as input. Then it does two checks: (1) the time specified in the transaction is lower than the current time. For this matter it interacts with the $\mathcal{G}_{\mathsf{CLOCK}}$ functionality to derive the current time. (2) It checks whether all parties mentioned in the transaction's referenced UTXO provided as inputs, provide a signature of the transaction. For this it interacts with the $\mathcal{F}_{\mathsf{SIG}}$ functionality.

Functionality $\mathcal{F}_{\mathsf{Script}}$

State: Stores set K with entries of form $(\mathcal{P}, v)$ where $\mathcal{P}$ is a party and v is a verification key.

Registering Verification Key: Upon receiving (VERIFICATIONKEY, sid, v) from a party $\mathcal{P}$ store $(\mathcal{P}, v)$ in K.

Transaction: Upon receiving (TRANSACTION, sid, tr) from $\mathcal{P}$, let (In, Out, t, Σ) = tr and $\mathsf{stub} = \mathsf{In}', \mathsf{Out}, \mathsf{t}$ where $\mathsf{In}' = \{(u) | (u, w) \in \mathsf{In}\}$, i.e. the transaction with omitted witnesses.

- Update time: Send (GET-TIME, $sid, \cdot$) to $\mathcal{G}_{\mathsf{CLOCK}}$ and receive (GET-TIME, sid, τ)
- Verify that $\forall \mathsf{utxo} \in \mathsf{In}$: $t \leq \tau$. Halt, otherwise.
- Verify $\forall \mathsf{ref} \in \mathsf{In}$: For each $\mathcal{P} \in \mathsf{ref.u.Party}$ retrieve $(\mathcal{P}, v)$ from K. Verify that $\mathsf{ref.w}$ contains a signature of stub from $\mathcal{P}$. For each $\sigma \in \mathsf{ref.w}$ send (VERIFY, $sid, \mathsf{stub}, \sigma, v$) to $\mathcal{F}_{\mathsf{Sig}}$ and verify that $\mathcal{F}_{\mathsf{SIG}}$ replies with (VERIFIED, $sid, \mathsf{stub}, 1$) exactly once.
- Send (TRANSACTION, sid, Removes, Adds) to $\mathcal{G}_{\mathsf{UTXO-Ledger}}$

4.6 The Pairwise Payment Channel Functionality

The $\mathcal{F}_{\mathsf{PWCH}}$ functionality creates, maintains and closes pairwise payment channel between two parties. For simplicity we opt to model a simple payment channel that uses timelocks to update a channel's state. The functionality consists of functions Open, Close, Channel_Update and Enforce. The Open function creates a Funding transaction based on a Funding transaction stub provided as input, commits it to the blockchain by interacting with $\mathcal{G}_{\mathsf{UTXO-Ledger}}$ and, after verifying that the mapping was applied on the ledger, stores the channel's state. The State_Update function redistributes the channel's funds while reducing its timelock by at least Δ whereas Close removes the channel's timelock while disabling any further updates on it. Lastly Enforce takes a channel as input and checks whether its timelock is lower or equal than the current round number. If that is the case a mapping representing a refund transaction is committed to the ledger.

4.6.1 General Behavior of our Functionalities

General Behavior. Before we detail the functionality's interface, we describe common non function-specific behavior of both functionalities $\mathcal{F}_{\mathsf{PWCH}}$ and $\mathcal{F}_{\mathsf{LVPC},\mathcal{F}_{\mathsf{PWCH}}}$, which is described in the next section.

Update time: At beginning of each round in which functionality is activated send message (clock-read, sid) to $\mathcal{G}_{\mathsf{CLOCK}}$ and receive the reply (CLOCK-READ, sid, τ'). Set internal variable $\tau = \tau'$.

Interactions with simulator: Whenever the functionality receives a message msg from any party or from $\mathcal{G}_{\mathsf{UTXO-Ledger}}$ it leaks the message to the simulator and appends sender and receiver.

Synchronization with the simulation: Interactions with the ledger are used to read its state as well as trigger a state change. The state on the ledger as well as whether a state change is permissible depends on the moment they are done as transactions that change the set of UTXO on the ledger can be sent by a party at any time. Therefore we need to ensure that the functionality's interaction with the ledger are at the same time as they happen in the simulation to achieve the same results and receive the same replies. Whenever the functionality sends a message msg to the ledger it waits for the simulator to leak a similar message by a honest party. Note that a TRANSACTION tagged message from the simulator is processed by the $\mathcal{F}_{Script}$ functionality first. Then the functionality sends the message only once and forwards any replies to the simulator.

Handling corrupted parties: We assume static corruption by a malicious adversary. At the beginning of execution the functionality asks the simulator which parties are controlled by the adversary and stores this information in set COR. The functionality ignores requests from any party in the ideal world of which counterpart in the simulation is corrupted by the adversary. The functionality needs to learn whether a party corrupted by the adversary misbehaved or

delayed execution of a protocol beyond a channel's lifetime. For this matter as soon as the simulator leaks that any simulated honest party $\mathcal{P}'_h$ sends message (FAILURE, sid, msg) to $\mathcal{Z}$ the functionality aborts execution of the function triggered by receiving msg and sends (FAILURE, sid, msg) to $\mathcal{P}'_h$'s dummy-party counterpart $\mathcal{P}_h$ in the ideal world.

4.6.2 The Payment Channel Functionality $\mathcal{F}_{\mathsf{PWCH}}$

Functionality $\mathcal{F}_{\mathsf{PWCH}}$

State: Current time τ. Set Γ of existing pairwise payment channel, set Γ_A of active pairwise payment channel. Set CONS with entries of form $(\mathcal{P}', \mathsf{msg})$ where $\mathcal{P}'$ is a party and msg is a message.
Initialization: Sets $\tau = 0$, $\Gamma = \Gamma_A = \mathsf{COR} = \mathsf{CONS} = \varnothing$.
Helper subfunctions:
$\mathsf{consent}$: A call of this sub-function is of form $\mathsf{consent}(\mathcal{P}, \mathsf{Parties}, \mathsf{msg})$ where $\mathcal{P}$ is a party, $\mathsf{Parties}$ is a set of parties and msg is a message. Let $\mathsf{Parties}_h = \mathsf{Parties} \setminus \mathsf{COR}$.

1. If $\mathcal{P} \notin \mathsf{COR}$, add ($\mathcal{P}$, msg) to CONS
2. **If** $\forall \mathcal{P}_h \in \mathsf{Parties}_h : (\mathcal{P}_h, \mathsf{msg}) \in \mathsf{CONS}$,
   then set $\mathsf{CONS} = \mathsf{CONS} \setminus \{(\mathcal{P}'_h, \mathsf{msg}) | \mathcal{P}'_h \in \mathsf{Parties}_h\}$ and return $\mathsf{is\_consent}$;
   Else return $\mathsf{no\_consent}$

$\mathsf{state\_update}$: A call of this sub-function is of form $\mathsf{state\_update}(\gamma, \mathcal{P}_A, b_A, \mathcal{P}_B, b_B, \delta_t)$.

1. Checks: $\gamma \in \Gamma_A$, $\delta_t \geq \Delta$, $\gamma.t - \delta_t > \mathsf{max}(\gamma.t_0, \tau)$, $b_A + b_B = \gamma.b_B + \gamma.b_A$; If any check fails halt
2. Update channel: $\gamma = (\gamma.f, \mathcal{P}_A, \mathcal{P}_B, b_A, b_B, \gamma.t - \delta_t, \gamma.t_0)$

Revoke: A call of this sub-function is of form $\mathsf{revoke}(\gamma)$. Set $\Gamma = \Gamma \setminus \{\gamma\}$.
$\mathsf{Activate}$: A call of this sub-function is of form $\mathsf{activate}(\gamma)$. Set $\Gamma = \Gamma \cup \{\gamma\}$.
$\mathsf{Balance}$: A call of this sub-function is of form $\mathsf{balance}(\gamma, \mathcal{P})$ where γ is a pairwise payment channel and $\mathcal{P}$ a party.

1. **if** $\mathcal{P} \notin \{\gamma.\mathcal{P}_A, \gamma.\mathcal{P}_B\}$ halt
2. **if** $\mathcal{P} = \gamma.\mathcal{P}_A$ return $\gamma.b_A$
3. **else** $\mathcal{P} = \gamma.\mathcal{P}_B$ return $\gamma.b_B$

Open: Upon receiving message $\mathsf{msg} = (\text{OPEN}, sid, m, \mathcal{P}_A, \mathcal{P}_B, b_A, b_B, t)$ from $\mathcal{P} \in \{\mathcal{P}_A, \mathcal{P}_B\}$ where m is a map, $b_A, b_B \in \mathbb{N}$ are amounts of coins and $t \in \mathbb{N}$ is a round number do:

1. Let $(\mathsf{In}, \mathsf{Out}) = m$, $\mathsf{Parties}_h = \{\mathcal{P}_A, \mathcal{P}_B\} \setminus \mathsf{COR}$.
2. **if** $\mathsf{consent}(\mathcal{P}, \{\mathcal{P}_A, \mathcal{P}_B\}, \mathsf{msg}) = \mathsf{no\_consent}$: halt
3. Verify; if any verification fails send $(\text{FAILURE}, sid, \mathsf{msg})$ to all in $\mathcal{P}_h$:
   - No overspending: $\Sigma_{o \in \mathsf{Out}} o.b \leq b_A + b_B + \Sigma_{o \in \mathsf{In}}$
   - Sufficient funds: $\cup_{o \in \{In\}} o.\mathsf{Party} = \{\mathcal{P}_A, \mathcal{P}_B\}$
   - Parties contribute sufficiently: For $i \in \{A, B\}$ holds $\Sigma_{o \in \mathsf{In}, o.\mathsf{Party} = \mathcal{P}_i} o.b \geq b_i$
   - Valid timelock: $t \geq \tau + \Delta + 1$
   - $\forall o \in \mathsf{In}$ send (CHECK, sid, o) to $\mathcal{G}_{\mathsf{UTXO-Ledger}}$. If $\mathcal{G}_{\mathsf{UTXO-Ledger}}$ replies with $(\text{CHECK\_OKAY}, sid, o)$ for each $o \in \mathsf{In}$ continue, otherwise halt
4. Funding: $m_f = (\mathsf{In}, \mathsf{Out} \cup \{f\})$ where $f = (b_A + b_B, \{\mathcal{P}_A, \mathcal{P}_B\})$ is a funding output
5. Send message $(\text{TRANSACTION}, sid, m_f)$ to $\mathcal{G}_{\mathsf{UTXO-Ledger}}$
6. For all $o \in (\mathsf{Out} \cup \{f\})$ send message (CHECK, sid, o) to $\mathcal{G}_{\mathsf{UTXO-Ledger}}$. If it replies $(\text{CHECK\_OKAY}, sid, o)$ for all $o \in (\mathsf{Out} \cup \{f\})$ continue; otherwise halt and repeat this step next round

Upon receiving message $(\text{SUCCESS}, sid, \mathsf{msg})$ from all parties in $\mathsf{Parties}_h$:

1. Update internal state: $\Gamma = \Gamma \cup \{\gamma\}$, $\Gamma_A = \Gamma_A \cup \{\gamma\}$ where $\gamma = (f, \mathcal{P}_A, \mathcal{P}_B, b_A, b_B, t, 0)$
2. Return message $(\text{SUCCESS}, sid, \mathsf{msg})$ to $\mathcal{P} \in \mathsf{Parties}_h$

Channel_Update: Upon receiving $\mathsf{msg} = (\text{CHANNEL\_UPDATE}, sid, \gamma, \mathcal{P}_0, b_0, \mathcal{P}_1, b_1, \delta_t)$ from $\mathcal{P} \in \{\gamma.\mathcal{P}_A, \gamma.\mathcal{P}_B\}$ where $b_A, b_B, \delta_t \in \mathbb{N}$. Do:

1. Let $\mathsf{Parties}_h = \{\gamma.\mathcal{P}_A, \gamma.\mathcal{P}_B\} \setminus \mathsf{COR}$
2. **if** $\mathsf{consent}(\mathcal{P}, \{\mathcal{P}_A, \mathcal{P}_B\}, \mathsf{msg}) = \mathsf{no\_consent}$: halt
3. Verify: $\{\mathcal{P}_0, \mathcal{P}_1\} = \{\mathcal{P}_A, \mathcal{P}_B\}$. Send $(\text{FAILURE}, sid, \mathsf{msg})$ to all in $\mathcal{P}_h$ if it fails

Upon receiving message $(\text{SUCCESS}, sid, \mathsf{msg})$ from all parties in $\mathsf{Parties}_h$:

1. Execute $\mathsf{state\_update}(\gamma, (\mathcal{P}_0, b_0), (\mathcal{P}_1, b_1), \delta_t)$
2. Return message $(\text{SUCCESS}, sid, \mathsf{msg})$ to all $\mathcal{P} \in \mathsf{Parties}_h$

Close: Upon receiving $\mathsf{msg} = (\text{CLOSE}, sid, \gamma)$ from $\mathcal{P} \in \{\mathcal{P}_A, \mathcal{P}_B\}$ where $\mathcal{P}_A = \gamma.\mathcal{P}_A$ and $\mathcal{P}_B = \gamma.\mathcal{P}_B$ do:

1. Let $\mathsf{Parties}_h = \{\gamma.\mathcal{P}_A, \gamma.\mathcal{P}_B\} \setminus \mathsf{COR}$.
2. **if** $\mathsf{consent}(\mathcal{P}, \{\mathcal{P}_A, \mathcal{P}_B\}, \mathsf{msg}) = \mathsf{no\_consent}$: halt
3. Verify: $\{\mathcal{P}_0, \mathcal{P}_1\} = \{\mathcal{P}_A, \mathcal{P}_B\}$. Send $(\text{FAILURE}, sid, \mathsf{msg})$ to all in $\mathcal{P}_h$ if it fails

Upon receiving message $(\text{SUCCESS}, sid, \mathsf{msg})$ from all parties in $\mathsf{Parties}_h$:

1. Execute $\mathsf{state\_update}(\gamma, (\gamma.\mathcal{P}_A, \gamma.b_A), (\gamma.\mathcal{P}_B, \gamma.b_B), \gamma.t - \gamma.t_0)$
2. $\Gamma_A = \Gamma_A \setminus \{\gamma\}$
3. Return message $(\text{SUCCESS}, sid, \mathsf{msg})$ to all $\mathcal{P} \in \mathsf{Parties}_h$

Enforce: Upon receiving $\mathsf{msg} = (\text{ENFORCE}, sid, \gamma)$ from party $\mathcal{P}$ do: Let $\mathcal{P}_A = \gamma.\mathcal{P}_A$ and $\mathcal{P}_B = \gamma.\mathcal{P}_B$.

1. Do the following. If any check or verification fails, immediately return message $(\text{ENFORCE}, sid, failure)$ to $\mathcal{P}$ and halt.
   - Check: $\mathcal{P} \in \{\mathcal{P}_A, \mathcal{P}_B\}$; $\gamma.t \leq \tau$
   - Send message $(\text{CHECK}, sid, \gamma.f)$ to $\mathcal{G}_{\mathsf{UTXO-Ledger}}$. If it replies $(\text{CHECK\_OKAY}, sid, o)$ continue, otherwise if it replies $(\text{CHECK\_FAILURE}, sid, o)$ halt
2. $m_r = \mathsf{REFUND\_MAP}(\gamma.f, \gamma.b_A, \gamma.b_B)$
3. Send message $(\text{TRANSACTION}, sid, m_r)$ to $\mathcal{G}_{\mathsf{UTXO-Ledger}}$
4. $\Gamma = \Gamma \setminus \{\gamma\}$; $\Gamma_A = \Gamma_A \setminus \{\gamma\}$
5. Return message $(\text{SUCCESS}, sid, \mathsf{msg})$ to $\mathcal{P}$

4.6.3 An Extension to the $\mathcal{F}_{\mathsf{PWCH}}$ Functionality

Functionality $\mathcal{F}^v_{\mathsf{PWCH}}$ is an extension of functionality $\mathcal{F}_{\mathsf{PWCH}}$ by providing an alternative function to open a pairwise payment channel which is required for the virtual payment channel functionality. This function takes a Funding UTXO instead of a Funding transaction stub as input and creates a pairwise payment channel without committing a Funding transaction to the ledger.

Functionality $\mathcal{F}^{v}_{\mathsf{PWCH}}$

Functionality that behaves as $\mathcal{F}_{\mathsf{PWCH}}$ but includes the following helper function to facilitate use with virtual channels.

Open Virtual: A call of this sub-function by an associated functionality is of form $\mathsf{open\_virtual}(f, \mathcal{P}_A, \mathcal{P}_B, b_A, b_B, t, t_0)$ where f is a funding output, $b_A, b_B \in \mathbb{N}$ are amounts of coins and $t, t_0 \in \mathbb{N}$ is a round number. Then:

1. $m_r = \mathsf{REFUND\_MAP}(f,\, b_A,\, b_B)$
2. Channel: $\gamma = (f, \mathcal{P}_A, \mathcal{P}_B, b_A, b_B, t, t_0)$
3. Update internal state: $M = M \cup \{(\gamma, m_r, t)\}$; $\Gamma = \Gamma \cup \{\gamma\}$; $\Gamma_A = \Gamma_A \cup \{\gamma\}$
4. Return γ

4.7 The Ideal Virtual Channel Functionality

In the following we present formal treatment of our protocol in the UC framework by introducing a functionality for lightweight virtual payment channel $\mathcal{F}_{\mathsf{LVPC},\mathcal{F}_{\mathsf{PWCH}}}$, associated with functionality $\mathcal{F}_{\mathsf{PWCH}}$. For this we make use of auxiliary functionality $\mathcal{F}_{\mathsf{Script}}$, global UTXO ledger functionality $\mathcal{G}_{\mathsf{UTXO-Ledger}}$, global clock functionality $\mathcal{G}_{\mathsf{CLOCK}}$, and functionality $\mathcal{F}_{\mathsf{SIG}}$. These functionalities are defined in Section 4.5.4.

The Ideal Virtual Channel Functionality. The lightweight virtual payment channel functionality $\mathcal{F}_{\mathsf{LVPC},\mathcal{F}_{\mathsf{PWCH}}}$ is used to create and close virtual payment channel between three parties. It provides access to functions VC-Open, VC-Close and VC-Enforce. Function VC-Open takes two pairwise payment channel between three parties as input, disables state updates on those, and creates three new pairwise payment channel, one between each pair of parties. To be able to enforce these channels it creates and stores mappings that represent Split, Merge and Punish transactions together with the time at which they become valid. Function VC-Close takes a virtual channel as input. First it checks whether no virtual channel have been created using the pairwise payment channel created with it. If positive it disables state updates on these channels, re-enables state updates for the original channels, updates their balance to reflect the latest balance distribution among the three channels and sets the channel's timelocks to be lower than the one of the Split mappings by Δ. Function Enforce is used to commit any mapping representing Split, Merge or Punish transactions if their timelocks have expired. This disables closure of the virtual channel because the funding UTXO of the original pairwise payment channels are removed from the ledger.

The functionality shares the same non-function specific behavior as $\mathcal{F}_{\mathsf{PWCH}}$, i.e. time management, interactions with the simulator and handling of corrupted

parties. We refer to the previous section for details.

Functionality $\mathcal{F}^{v}_{\mathsf{LVPC},\mathcal{F}_{\mathsf{PWCH}}}$

Has access to $\mathcal{F}^{v}_{\mathsf{PWCH}}$'s helper functions.

State: Set of closable virtual payment channel Γ^v of virtual payment channels. List M^v of entries (γ'^v, m, t) where γ'^v is a virtual payment channel, m is a partial mapping and t is a round number. It has access to the internal state of $\mathcal{F}^{v}_{\mathsf{PWCH}}$ including set Γ of pairwise payment channels. Moreover it shares common state with $\mathcal{F}^{v}_{\mathsf{PWCH}}$ which is the current round number τ, list of corrupted parties COR and set of consent giving parties CONS.

Initialization: Initializes $\mathcal{F}^{v}_{\mathsf{PWCH}}$ and shared state $\tau, \mathsf{COR}, \mathsf{CONS}$. Sets $\Gamma^v = M^v = \varnothing$.

VC-Open: Execute upon receiving message $\mathsf{msg} = (\text{OPEN}, sid, \gamma_0, \gamma_1, \delta, t, \mathcal{P}_A, \mathcal{P}_B, \mathcal{P}_C)$ from $\mathcal{P} \in \{\mathcal{P}_A, \mathcal{P}_B, \mathcal{P}_C\}$ where $\gamma_0, \gamma_1 \in \Gamma$, $\delta \in \mathbb{N}$ is an amount of coins and $t \in \mathbb{N}$ is a round number. Let $\mathsf{Parties}_h = \{\mathcal{P}_A, \mathcal{P}_B, \mathcal{P}_C\} \setminus \mathsf{COR}$:

1. **if** $\mathsf{cns} = \mathsf{consent}(\mathcal{P}, \{\mathcal{P}_A, \mathcal{P}_B, \mathcal{P}_C\}, \mathsf{msg}) = \mathsf{no\_consent}$: halt
2. Verify; if any verification fails send $(\text{FAILURE}, sid, \mathsf{msg})$ to all in $\mathcal{P}_h$:
   - $\{\mathcal{P}_A, \mathcal{P}_B\} = \{\gamma_0.\mathcal{P}_A, \gamma_0.\mathcal{P}_B\}$ and $\{\mathcal{P}_B, \mathcal{P}_C\} = \{\gamma_1.\mathcal{P}_A, \gamma_1.\mathcal{P}_B\}$
   - **for each** $\gamma \in \{\gamma_0, \gamma_1\}$: **if** $\{\gamma.\mathcal{P}_A, \gamma.\mathcal{P}_B\} \setminus \mathsf{COR} \neq \varnothing$ **then**
     - $\gamma \in \Gamma$; $\gamma.t > \tau + 2\Delta$; $\gamma.t_0 > \tau + 2\Delta$; $\gamma.b_A \geq \delta/2$ and $\gamma.b_B \geq \delta/2$

Upon receiving message $(\text{SUCCESS}, sid, \mathsf{msg})$ from all parties in $\mathsf{Parties}_h$:

1. Create Mappings
   - Split: $m_{0,S} = \mathsf{SPLIT\_MAP}(\gamma_0.f,\ \delta)$; $m_{1,S} = \mathsf{SPLIT\_MAP}(\gamma_1.f,\ \delta)$
   - Merge: $m_{\mathsf{mrg}} = \mathsf{MERGE\_MAP}(\mathsf{OUT\_DELTA}\ (m_{0,S}),\ \mathsf{OUT\_DELTA}(\ m_{1,S}),\ \delta)$
   - Punish: $m_{0,p} = \mathsf{PUNISH\_MAP}(\mathsf{OUT\_DELTA}(m_{0,S}),\ \mathcal{P}_A)$
   - $m_{1,p} = \mathsf{PUNISH\_MAP}\ (\ \mathsf{OUT\_DELTA}(m_{1,S}),\ \mathcal{P}_C)$
2. Create new channel, revoke old
   - $\gamma_{A,B} = \mathsf{open\_virtual}(f_{A,B}, \mathcal{P}_A,\ \mathcal{P}_B,\ \mathsf{balance}(\gamma_0, \mathcal{P}_A) - \delta/2,\ \mathsf{balance}(\gamma_0, \mathcal{P}_B) - \delta/2,\ t,\ \gamma_0.t)$
   - $\gamma_{B,C} = \mathsf{open\_virtual}(f_{B,C}, \mathcal{P}_B,\ \mathcal{P}_C,\ \mathsf{balance}(\gamma_1,\ \mathcal{P}_B) - \delta/2,\ \mathsf{balance}(\gamma_1, \mathcal{P}_C) - \delta/2,\ t,\ \gamma_1.t)$

- $\gamma_{A,C} = \mathsf{open\_virtual}(f_{A,C}, \mathcal{P}_A, \mathcal{P}_C, \delta/2, \delta/2, t, \mathsf{max}(\gamma_0.t, \gamma_1.t) + \Delta)$
- Revoke old channel: $\mathsf{revoke}(\gamma_0)$ and $\mathsf{revoke}(\gamma_1)$

Virtual channel: $\gamma^v = (\gamma_0, \gamma_1, \gamma_{A,B}, \gamma_{B,C}, \gamma_{A,C}, \mathcal{P}_A, \mathcal{P}_B, \mathcal{P}_C, \delta,$

3. $\mathsf{min}(\gamma_0.t, \gamma_1.t) - 2\Delta)$
4. Update: $\Gamma^v = \Gamma^v \cup \{\gamma^v\}$ and $M^v = M^v \cup \{(\gamma^v, m_{0,S}, \gamma_0.t - \Delta), (\gamma^v, m_{1,S}, \gamma_1.t - \Delta), (\gamma^v, m_{\mathsf{mrg}}, \mathsf{max}(\gamma_0.t, \gamma_1.t)), (\gamma^v, m_{0,p}, \mathsf{max}(\gamma_0.t, \gamma_1.t) + \Delta), (\gamma^v, m_{1,p}, \mathsf{max}(\gamma_0.t, \gamma_1.t) + \Delta)\}$
5. Return message $(\text{SUCCESS}, sid, \mathsf{msg})$ to $\mathcal{P}$

VC-Close: Upon receiving message $\mathsf{msg} = (\text{CLOSE}, sid, \gamma^v,)$ from $\mathcal{P} \in \{\mathcal{P}_A, \mathcal{P}_B, \mathcal{P}_C\}$ where $\mathcal{P}_A = \gamma^v.\mathcal{P}_A$, $\mathcal{P}_B = \gamma^v.\mathcal{P}_B$ and $\mathcal{P}_C = \gamma^v.\mathcal{P}_C$. Let $(\gamma_0, \gamma_1, \gamma_{A,B}, \gamma_{B,C}, \gamma_{A,C}, \mathcal{P}_A, \mathcal{P}_B, \mathcal{P}_C, \delta, t) = \gamma^v$. Moreover let $\mathsf{Parties}_h = \{\mathcal{P}_A, \mathcal{P}_B, \mathcal{P}_C\} \setminus \mathsf{COR}$. Do:

1. **if** $\mathsf{cns} = \mathsf{consent}(\mathcal{P}, \{\mathcal{P}_A, \mathcal{P}_B, \mathcal{P}_C\}, \mathsf{msg}) = \mathsf{no\_consent}$: halt
2. Verify; if any verification fails send $(\text{FAILURE}, sid, \mathsf{msg})$ to all in $\mathcal{P}_h$:
   - $\gamma^v \in \Gamma^v$; $\{\gamma_{A,B}, \gamma_{B,C}, \gamma_{A,C}\} \subseteq \mathcal{F}^v_{\mathsf{PWCH}}.\Gamma$; $t > \tau$; $\gamma_0.t_0 > \tau + 2\Delta$, $\gamma_1.t_0 > \tau + 2\Delta$

Upon receiving message $(\text{SUCCESS}, sid, \mathsf{msg})$ from all parties in $\mathsf{Parties}_h$:

1. Revoke old channel, reactivate and update original channel:
   - $\mathsf{activate}(\gamma_0)$, $\mathsf{activate}(\gamma_1)$
   - $\mathsf{state\_update}(\gamma_0, \mathcal{P}_A, b_A, \mathcal{P}_B, b_B, 2\Delta)$ and $\mathsf{state\_update}(\gamma_1, \mathcal{P}_B, b'_B, \mathcal{P}_C, b_C, 2\Delta)$ where $b_A = \gamma_{A,B}.b_A + \gamma_{A,C}.b_A$; $b_B = \gamma_{A,B}.b_B + \gamma_{A,C}.b_B$; $b'_B = \gamma_{B,C}.b_A + \gamma_{A,C}.b_A$; $b_C = \gamma_{B,C}.b_B + \gamma_{A,C}.b_B$.
   - $\mathsf{revoke}(\gamma_{A,B})$, $\mathsf{revoke}(\gamma_{B,C})$, $\mathsf{revoke}(\gamma_{A,C})$
2. Update internal state: $\Gamma^v = \Gamma^v \setminus \{\gamma^v\}$
3. Return message $(\text{SUCCESS}, sid, \mathsf{msg})$ to $\mathcal{P}$

VC-Enforce: Triggered upon receiving $\mathsf{msg} = (\text{ENFORCE}, sid, \gamma^v)$ from party $\mathcal{P}$ where γ^v is a lightweight virtual payment channel. Let $\mathcal{P}_A = \gamma^v.\mathcal{P}_A$, $\mathcal{P}_B = \gamma^v.\mathcal{P}_B$ and $\mathcal{P}_C = \gamma^v.\mathcal{P}_C$.

1. Check if $\mathcal{P} \in \{\mathcal{P}_A, \mathcal{P}_B, \mathcal{P}_C\}$, $\gamma \in \Gamma$
2. Let $M_\gamma = \{(\gamma'^v, m', t') | (\gamma'^v, m', t') \in M^v; \gamma'^v = \gamma^v; t' \leq \tau, \exists \mathsf{utxo} \in m'.\mathsf{In} : \mathcal{P} \in \mathsf{utxo.Party}\}$

3. **for each** $m \in M_\gamma$
   - Let $(\mathsf{In}, \mathsf{Out}) = m$. For all $o \in \mathsf{In}$ send message (CHECK, sid, o) to $\mathcal{G}_{\mathsf{UTXO-Ledger}}$. If it replies $(\text{CHECK\_OKAY}, sid, o)$ for all $o \in \mathsf{In}$:
     - Send message $(\text{TRANSACTION}, sid, m)$ to $\mathcal{G}_{\mathsf{UTXO-Ledger}}$
     - Channel cannot be closed: $\Gamma^v = \Gamma^v \setminus \{\gamma^v\}$
4. Return message $(\text{SUCCESS}, sid, \mathsf{msg})$ to $\mathcal{P}$

Definition 4.7.1. *Balance Security: The sum of a honest party's funds only changes with its consent.*

Definition 4.7.2. *Liveness: Eventually all of a party's funds are unlocked and committed to the ledger within UTXO that are spendable by the party alone.*

Security of Funds and Liveness. In the following we briefly argue that functionality $\mathcal{F}_{\mathsf{LVPC},\mathcal{F}_{\mathsf{PWCH}}}$ fulfills these two properties for honest parties by design. We expect a honest party to call sub-function VC-Enforce as soon as they would lead to submission of a mapping to the ledger, i.e. at times $\gamma^v.\gamma_0.t - \Delta$, $\gamma^v.\gamma_1.t - \Delta$, $\mathsf{max}(\gamma^v.\gamma_0.t, \gamma^v.\gamma_1.t)$ and in case a punish transaction has to be committed at time $\mathsf{max}(\gamma^v.\gamma_0.t, \gamma^v.\gamma_1.t) + \Delta$. Eventually all funds that a honest party holds will be accessible over UTXOs on the ledger such that liveness holds. Balance Security holds since only $\mathcal{F}_{\mathsf{PWCH}}$'s channel update function, which requires the party's consent, changes a honest parties' balance.

4.8 The Pairwise Payment Channel Protocol

Analogous to the functionalities we define protocols PWCH and it's extension PWCH^v for pairwise payment channel, as well as a protocol for virtual payment channel $\mathsf{LVPC}_{\mathsf{PWCH}^v}$ in the following section. They are designed similar to their functionality counterparts, however, parties involved in such a protocol need to additionally do:

- Instead of abstract mappings, transactions need to be created. Parties need to exchange signatures as ways to provide consent
- In the Channel_Update method, in addition to updating an internal state, parties need to create a transaction to make it enforceable
- Order matters: The root of a transaction subtree spending the Funding UTXO needs to be signed last. In VC-Open and VC-Close the intermediary party is allowed to sign the root of the transactions subtree only after receiving signatures of those from the other parties

4.8.1 General Behavior

The following behavior is always performed by any honest party and for either protocol and is the analogous component of the functionalities' common non-function specific behavior:

Update time: Sub-function that is executed at the beginning of each round. Send message $(\mathsf{clock\text{-}read}, sid)$ to $\mathcal{G}_{\mathsf{CLOCK}}$ and receive reply $(\text{CLOCK-READ}, sid, \tau')$. Set internal variable $\tau = \tau'$.

Handling corrupted parties: During execution as soon as another party is observed deviating from a protocol send message $(\text{FAILURE}, sid, \mathsf{msg})$ to $\mathcal{Z}$ and halt.

Timeouts: Upon receiving message$(\text{OPEN}, sid, m, \mathcal{P}_A, \mathcal{P}_B, b_A, b_B, t)$, $(\text{STATE\_UPDATE}, sid, \gamma, \delta)$ or $(\text{CLOSE}, sid, \gamma')$, if the execution of the subprotocol triggered by these messages does not finish until round number t, $\gamma.t$ or $\gamma'.t$ respectively the counterparty is considered to be unresponsive. Send $(\text{FAILURE}, sid, \mathsf{msg})$ to $\mathcal{Z}$ and halt the sub-protocol's execution.

4.8.2 The PWCH Protocol

Protocol PWCH

State: Each party stores current time τ, verification key v, set of other party's verification keys V, set Γ of created pairwise payment channel and Γ_A of active pairwise payment channel, set CONS with entries of form $(\mathcal{P}, \mathsf{msg})$, Refund transactions $\mathsf{RFND}(\gamma)$ for each created channel $\gamma \in \Gamma$.

Initialization: Send message $(\mathsf{register}, sid)$ to $\mathcal{G}_{\mathsf{CLOCK}}$ followed by $(\mathsf{clock\text{-}read}, sid)$ to receive $(\text{CLOCK-READ}, sid, \tau', \mathsf{fast})$. Initialize $\tau = \tau'$. Set $\Gamma = \Gamma_A = \mathsf{CONS} = \varnothing$. Lastly send message (KeyGen, sid) to $\mathcal{F}_{\mathsf{SIG}}$ and wait for reply $(\text{VERIFICATION KEY}, sid, v')$. Set verification key $v = v'$. Send message $(\text{VERIFICATION KEY}, sid, v)$ to $\mathcal{F}_{\mathsf{Script}}$ and broadcast $(\text{VERIFICATION KEY}, sid, v)$ to all parties.

Verification Keys: Whenever receiving a message $(\text{VERIFICATION KEY}, sid, v)$ from another party $\mathcal{P}$ store tuple $(\mathcal{P}, v)$ in V.

Subprotocol sign_broadcast: Takes as input signing parties $\mathsf{Parties}_S$, transaction tr, set of receiving parties $\mathsf{Parties}_R$.

1. Set $\mathsf{stub} = (\mathsf{tr}.t, \mathsf{tr.Ref}, \mathsf{tr.Out})$
2. Each $\mathcal{P} \in \mathsf{Parties}_S$:
   - send message $(\text{SIGN}, sid, \mathsf{stub})$ to $\mathcal{F}_{\mathsf{SIG}}$ and receive reply $(\text{SIGNATURE}, sid, \mathsf{stub}, \sigma)$
   - $\forall_{(\mathsf{out},\Sigma)\in \mathsf{Tr.U_{In}}}$: **if** $\mathcal{P} \in \mathsf{out.Party}$ set $\Sigma = \Sigma \cup \{\sigma\}$
   - send message $(\text{SIGNATURE}, sid, \mathsf{stub}, \sigma)$ to all parties in $\mathsf{Parties}$

3. Each party in Parties upon receiving (SIGNATURE, sid, stub, σ) does:
   - Lookup entry $(\mathcal{P}, v') \in V$ and send (VERIFY, sid, m, σ', v') to $\mathcal{F}_{\mathsf{SIG}}$
   - **if** such an entry does not exist, or if $\mathcal{F}_{\mathsf{SIG}}$ replies with (VERIFIED, sid, m, f) where $f = 0$ **then** return failure
   - **else** $\forall_{(\mathsf{out},\Sigma)\in \mathsf{Tr.U_{In}}}$: **if** $\mathcal{P} \in$ out.Party set $\Sigma = \Sigma \cup \{\sigma\}$
4. Return success

Consent: Whenever a party receives a message of form (REQ, sid, msg) from a party $\mathcal{P}''$ they store tuple $(\mathcal{P}'', \mathsf{msg})$ in CONS.
Subprotocol consent_verification:

Inputs: Parties, msg

1. Send message (REQ, sid, msg) to all $\mathcal{P} \in$ Parties
2. If $\exists(\mathcal{P}, \mathsf{msg}) \notin$ CONS wait; proceed upon receiving (REQ, sid, msg) from $\mathcal{P}$
3. Set CONS = CONS $\setminus \{\mathsf{req}\}$

Request Verifications: Upon receiving message msg from $\mathcal{Z}$ triggering execution of a subprotocol below, if any verifications fails send (FAILURE, sid, msg) to $\mathcal{Z}$.
Revoke: A call of this sub-protocol is of form revoke(γ). Set $\Gamma_A = \Gamma_A \setminus \{\gamma\}$.
Activate: A call of this sub-protocol is of form activate(γ, tr). Set $\Gamma_A = \Gamma_A \cup \{\gamma\}$; RFND($\gamma$) = tr.
Balance: A call of this sub-protocol is of form balance($\gamma, \mathcal{P}$) where γ is a pairwise payment channel and $\mathcal{P}$ a party.

1. **if** $\mathcal{P} \notin \{\gamma.\mathcal{P}_A, \gamma.\mathcal{P}_B\}$ halt
2. **if** $\mathcal{P} = \gamma.\mathcal{P}_A$ return $\gamma.b_A$
3. **else** $\mathcal{P} = \gamma.\mathcal{P}_B$ return $\gamma.b_B$

Open: A party $\mathcal{P}$ upon receiving msg = (OPEN, $sid, m, \mathcal{P}_A, \mathcal{P}_B, b_A, b_B, t$) from $\mathcal{Z}$ where $\mathcal{P}_A$ and $\mathcal{P}_B$ are parties, m is a map, $b_A, b_B \in \mathbb{N}$ are amounts of coins and $t \in \mathbb{N}$ is a round number does:

1. Party is addressed: Check $\mathcal{P} \in \{\mathcal{P}_A, \mathcal{P}_B\}$. Otherwise ignore request. In the following let $\mathcal{P}_c$ be $\mathcal{P}$'s counterparty.
2. Consent: Execute sub-protocol consent_verification($\{\mathcal{P}_c\}$, msg)

3. Let $(\mathsf{In}, \mathsf{Out}) = m$. Verify:
   - $\Sigma_{o \in \mathsf{Out}} o.b \leq b_A + b_B + \Sigma_{o \in \mathsf{In}}$; $\cup_{o\ in\{In\}} o.\mathsf{Party} = \{\mathcal{P}_A, \mathcal{P}_B\}$; for $i \in \{A, B\}$ holds $\Sigma_{o \in \mathsf{In}, o.\mathsf{Party}=\mathcal{P}_i} o.b \geq b_i$; $t \geq \tau + \Delta + 1$
   - $\forall o \in \mathsf{In}$, after sending message (CHECK, sid, o) to $\mathcal{G}_{\mathsf{UTXO-Ledger}}$ (CHECK_OKAY, sid, o) is returned
4. If check of verification fails the party sends (FAILURE, sid, msg) to $\mathcal{Z}$
5. Funding: $m_f = (\mathsf{In}, \mathsf{Out} \cup \{f\})$ where $f = (b_A + b_B, \{\mathcal{P}_A, \mathcal{P}_B\})$. Transaction $\mathsf{tr}_f = \phi(m_f, 0)$
6. Refund: $\mathsf{tr}_r = \mathsf{REFUND\_TR}(f,\ t,\ b_A,\ b_B)$
7. Perform sign_broadcast$(\{\mathcal{P}_A, \mathcal{P}_B\}, \mathsf{tr}_r, \{\mathcal{P}_A, \mathcal{P}_B\})$
8. Perform sign_broadcast$(\{\mathcal{P}_A, \mathcal{P}_B\}, \mathsf{tr}_f, \{\mathcal{P}_A, \mathcal{P}_B\})$
9. Send (TRANSACTION, sid, tr_f) to $\mathcal{F}_{\mathsf{Script}}$.
10. Poll result: Send message (CHECK, sid, f) to $\mathcal{G}_{\mathsf{UTXO-Ledger}}$. If it returns (CHECK_OKAY, sid, f) continue; otherwise if it returns (CHECK_FAILURE, sid, f) halt and repeat next round
11. Set $\gamma = (f, \mathcal{P}_A, \mathcal{P}_B, b_A, b_B, t, 0)$
12. Update state: $\Gamma = \Gamma \cup \{\gamma\}$; $\Gamma_A = \Gamma_A \cup \{\gamma\}$; $\mathsf{RFND}(\gamma) = \mathsf{tr}_r$
13. Return message (SUCCESS, sid, msg) to $\mathcal{Z}$

Channel_Update: A party $\mathcal{P}$, upon receiving $\mathsf{msg} =$ (STATE_UPDATE, $sid, \gamma, \mathcal{P}_0, b_0, \mathcal{P}_1, b_1, \delta_t$) from $\mathcal{Z}$ does: Let $\mathcal{P}_A = \gamma.\mathcal{P}_A$ and $\mathcal{P}_B = \gamma.\mathcal{P}_B$.

1. Party is addressed: Check $\mathcal{P} \in \{\mathcal{P}_A, \mathcal{P}_B\}$. Otherwise ignore request. In the following let $\mathcal{P}_c$ be $\mathcal{P}$'s counterparty
2. Verify $\mathcal{P} \in \{\mathcal{P}_A, \mathcal{P}_B\}$. Otherwise ignore request. In the following let $\mathcal{P}_c$ be $\mathcal{P}$'s counterparty
3. Perform $\mathsf{consent\_verification}(\{\mathcal{P}_c\}, \mathsf{msg})$
4. **if** $\mathcal{P}_0 = \mathcal{P}_A$ **then** $b_A = b_0, b_B = b_1$ **else** $b_A = b_1, b_B = b_0$
5. Verify $(\mathcal{P}_A, \mathcal{P}_B, \gamma) \in \Gamma$; $\gamma \in \Gamma$; $\delta_t \geq \Delta$; $\gamma.t - \delta_t > \mathsf{max}(\gamma.t_0, \tau)$; $b_A + b_B = \gamma.b_B + \gamma.b_A$
6. Refund: $\mathsf{tr}_r = \mathsf{tr}_r = \mathsf{REFUND\_TR}(\gamma.f,\ \gamma.t - \delta_t,\ b_A,\ b_B)$
7. Perform sign_broadcast$(\{\mathcal{P}_A, \mathcal{P}_B\}, \mathsf{tr}_r, \{\mathcal{P}_A, \mathcal{P}_B\})$

8. Update state: $\gamma = (\gamma.f, \mathcal{P}_A, \mathcal{P}_B, b_A, b_B, \gamma.t - \delta_t, \gamma.t_0)$; $\mathsf{RFND}(\gamma) = \mathsf{tr}_r$
9. Send message $(\text{SUCCESS}, sid, \mathsf{msg})$ to $\mathcal{Z}$

Close: A party $\mathcal{P}$, upon receiving $\mathsf{msg} = (\text{CLOSE}, sid, \gamma)$ from $\mathcal{Z}$ does: Let $\mathcal{P}_A = \gamma.\mathcal{P}_A$ and $\mathcal{P}_B = \gamma.\mathcal{P}_B$.

1. Verify $\mathcal{P} \in \{\mathcal{P}_A, \mathcal{P}_B\}$. Otherwise ignore request. In the following let $\mathcal{P}_c$ be $\mathcal{P}$'s counterparty.
2. Execute $\mathsf{consent\_verification}(\{\mathcal{P}_c\}, \mathsf{msg})$
3. Verify $(\mathcal{P}_A, \mathcal{P}_B, \gamma) \in \Gamma$
4. Refund: $\mathsf{tr}_r = \mathsf{tr}_r = \mathsf{REFUND\_TR}(\gamma.f,\ \gamma.t,\ \gamma.b_A,\ \gamma.b_B)$
5. Perform sign_broadcast$(\{\mathcal{P}_A, \mathcal{P}_B\}, \mathsf{tr}_r, \{\mathcal{P}_A, \mathcal{P}_B\})$
6. Update state: $\gamma = (\gamma.f, \mathcal{P}_A, \mathcal{P}_B, \gamma.b_A, \gamma.b_B, 0)$; $\mathsf{RFND}(\gamma) = \mathsf{tr}_r$
7. Return message $(\text{SUCCESS}, sid, \mathsf{msg})$ to $\mathcal{Z}$

Enforce: A party $\mathcal{P}$ upon receiving $\mathsf{msg} = (\text{ENFORCE}, sid, \gamma)$ from party $\mathcal{Z}$ does: Let $\mathcal{P}_A = \gamma.\mathcal{P}_A$ and $\mathcal{P}_B = \gamma.\mathcal{P}_B$.

1. Verify:
   - $\gamma \in \Gamma_A$; $\gamma.t \leq \tau$. In the following let $\phi^{-1}(\mathsf{RFND}(\gamma)) = (m, t, \Sigma)$, $(\mathsf{In}, \mathsf{Out}) = m$
2. Let $(\mathsf{In}, \mathsf{Out}) = m$. For all $o \in \mathsf{In}$ send message (CHECK, sid, o) to $\mathcal{G}_{\mathsf{UTXO-Ledger}}$. If it replies $(\text{CHECK\_OKAY}, sid, o)$ for all $o \in \mathsf{In}$ continue, otherwise if it replies $(\text{CHECK\_FAILURE}, sid, o)$ for any $o \in \mathsf{In}$ halt
3. Set $\Gamma = \Gamma \setminus \{\gamma\}$, $\Gamma_A = \Gamma_A \setminus \{\gamma\}$
4. Send $(\text{TRANSACTION}, sid, \mathsf{tr})$ to $\mathcal{F}_{\mathsf{Script}}$
5. Return message $(\text{SUCCESS}, sid, \mathsf{msg})$ to $\mathcal{Z}$.

4.8.3 An Extension to the PWCH Protocol

Similar to the functionality we provide an extension of our protocol which is PWCH^v that includes an interface of opening a pairwise payment channel that can be used with the virtual channel construction. More specifically it allows for the Funding UTXO to not be committed to the ledger.

Protocol PWCH^v

Protocol that behaves as PWCH but is modified to facilitate use with virtual channels by providing following sub-protocol:
Open Virtual: A call of this sub-protocol is of form $\mathsf{open\_virtual}(f, \mathcal{P}_A, \mathcal{P}_B, b_A, b_B, t, t_0)$ where f is a funding output, b_A, $b_B \in \mathbb{N}$ are amounts of coins and $t, t_0 \in \mathbb{N}$ are round numbers. Let $\mathcal{P}$ be caller of this function. Then:

1. Set $\mathsf{tr}_r = \mathsf{REFUND\_TR}(f,\ t,\ b_A,\ b_B)$
2. Perform sign_broadcast$(\{\mathcal{P}_A, \mathcal{P}_B\}, \mathsf{tr}_r, \{\mathcal{P}_A, \mathcal{P}_B\})$
3. Channel: $\gamma = (f, \mathcal{P}_A, \mathcal{P}_B, b_A, b_B, t, t_0)$
4. Update internal state: $\Gamma = \Gamma \cup \{\gamma\}$
5. Return γ, tr_r

4.9 The Formal Virtual Channel Protocol

Protocol $\mathsf{LVPC}_{\mathsf{PWCH}^v}$ utilizes protocols PWCH^v to implement $\mathcal{F}_{\mathsf{LVPC},\mathcal{F}_{\mathsf{PWCH}}}$. In addition to the sub-protocols stated below, parties perform non-function specific behavior to track time, and handle misbehaving parties. This behavior is shared with protocol $\mathsf{LVPC}_{\mathsf{PWCH}^v}$ and we refer to the previous section for details.

Analogous to protocol PWCH^v the protocols' design is derived from functionality $\mathcal{F}_{\mathsf{LVPC},\mathcal{F}_{\mathsf{PWCH}}}$ and follows its structure, however, in turn it has to handle creation and storage of transactions and it has to handle signatures of transactions as well as their order to enforce atomic setup of our construction.

Protocol $\mathsf{LVPC}_{\mathsf{PWCH}^v}$

Has access to PWCH^v's internal state and helper sub-protocols.
State: Each party $\mathcal{P}$ stores the following state. Set of closable virtual payment channel Γ^v, set Tr^v of entries (γ'^v, tr) where γ'^v is a virtual payment channel, tr is a transaction. It has access to the internal state and helper protocols of PWCH^v and shares common state with PWCH^v which is the current round number τ, verification key v list of other parties' verification keys V and set of consent giving parties CONS.
Initialization: Execute PWCH^v's **Initialization** sub-protocol. Moreover set $\Gamma^v = \mathsf{Tr}^v = \varnothing$.
VC-Open: Executed upon receiving message $\mathsf{msg} =$ $(\text{OPEN}, sid, \gamma_0, \gamma_1, \delta, t, \mathcal{P}_A, \mathcal{P}_B, \mathcal{P}_C)$ where $\gamma_0, \gamma_1 \in \Gamma$, $\delta \in \mathbb{N}$ is an amount of coins and $t \in \mathbb{N}$ is a point in time. In the following let $\mathcal{P}_{CMP} = \{\mathcal{P}_A, \mathcal{P}_B, \mathcal{P}_C\} \setminus \{\mathcal{P}\}$ and $\mathsf{Parties}_h = \{\mathcal{P}_A, \mathcal{P}_B, \mathcal{P}_C\} \setminus \mathsf{COR}$.

1. Party is addressed: Check $\mathcal{P} \in \{\mathcal{P}_A, \mathcal{P}_B, \mathcal{P}_C\}$, otherwise ignore request
2. Perform $\mathsf{consent\_verification}(\mathcal{P}_{CMP}, \mathsf{msg})$
3. Verify:
   - $\{\mathcal{P}_A, \mathcal{P}_B\} = \{\gamma_0.\mathcal{P}_A, \gamma_0.\mathcal{P}_B\}$ and $\{\mathcal{P}_B, \mathcal{P}_C\} = \{\gamma_1.\mathcal{P}_A, \gamma_1.\mathcal{P}_B\}$
   - **for each** $\gamma \in \{\gamma_0, \gamma_1\}$: **if** $\mathcal{P} \in \{\gamma.\mathcal{P}_A, \gamma.\mathcal{P}_B\}$ **then**
     - $\gamma \in \mathcal{F}^v_{\mathsf{PWCH}}.\Gamma$; $\gamma.t > \tau + 2\Delta$; $\gamma.t_0 > \tau + 2\Delta$; $\gamma.b_A \geq \delta/2$ and $\gamma.b_B \geq \delta/2$
4. $\mathcal{P}_A$ and $\mathcal{P}_B$: $\mathsf{tr}_{0,S} = \mathsf{SPLIT\_TR}(\gamma_0.f,\ \delta,\ \gamma_0.t - \Delta)$

   $\mathsf{tr}_{0,p} = \mathsf{PUNISH\_TR}(\mathsf{OUT\_DELTA}(\mathsf{tr}_{0,S}),\ \mathcal{P}_A,\ \mathsf{max}(\gamma_0.t, \gamma_1.t) + \Delta)$
5. $\mathcal{P}_B$ and $\mathcal{P}_C$: $\mathsf{tr}_{0,S} = \mathsf{SPLIT\_TR}(\gamma_1.f,\ \delta,\ \gamma_1.t - \Delta)$

   $\mathsf{tr}_{1,p} = \mathsf{PUNISH\_TR}(\mathsf{OUT\_DELTA}(\mathsf{tr}_{1,S}),\ \mathcal{P}_C,\ \mathsf{max}(\gamma_0.t, \gamma_1.t) + \Delta)$
6. $\mathsf{tr}_{\mathsf{mrg}} = \mathsf{MERGE\_TR}(\mathsf{OUT\_DELTA}(\mathsf{tr}_{0,S}),\ \mathsf{OUT\_DELTA}(\mathsf{tr}_{0,S}),\ \delta,\ \mathsf{max}(\gamma_0.t,\ \gamma_1.t))$
7. $\mathcal{P}_A$ and $\mathcal{P}_B$:

   $\gamma_{A,B}, \mathsf{tr}_{A,B} = \mathsf{open\_virtual}(\mathsf{OUT\_CH}(\mathsf{tr}_{0,S}), \mathcal{P}_A, \mathcal{P}_B, \mathsf{balance}(\gamma_0, \mathcal{P}_A) - \delta/2, \mathsf{balance}(\gamma_0, \mathcal{P}_B) - \delta/2, t, \gamma_0.t)$
8. $\mathcal{P}_B$ and $\mathcal{P}_C$: $\gamma_{B,C}, \mathsf{tr}_{B,C} = \mathsf{open\_virtual}(\mathsf{OUT\_CH}(\mathsf{tr}_{1,S}),\ \mathcal{P}_B,\ \mathcal{P}_C,\ \mathsf{balance}(\gamma_1,\ \mathcal{P}_B) - \delta/2,\ \mathsf{balance}(\gamma_1,\ \mathcal{P}_C) - \delta/2,\ t,\ \gamma_1.t)$
9. $\mathcal{P}_A$ and $\mathcal{P}_C$: $\gamma_{A,C}, \mathsf{tr}_{A,C} = \mathsf{open\_virtual}(\mathsf{OUT\_CH}(\mathsf{tr}_{\mathsf{mrg}}),\ \mathcal{P}_A,\ \mathcal{P}_C,\ \delta/2,\ \delta/2,\ t,\ \mathsf{max}(\gamma_0.t,\ \gamma_1.t) + \Delta)$
10. Perform $\mathsf{sign\_broadcast}(\{\mathcal{P}_A, \mathcal{P}_B, \mathcal{P}_C\}, \mathsf{tr}_{\mathsf{mrg}}, \{\mathcal{P}_A, \mathcal{P}_B, \mathcal{P}_C\})$
11. Perform $\mathsf{sign\_broadcast}(\{\mathcal{P}_A, \mathcal{P}_B\}, \mathsf{tr}_{0,p}, \{\mathcal{P}_A, \mathcal{P}_B\})$
12. Perform $\mathsf{sign\_broadcast}(\{\mathcal{P}_B, \mathcal{P}_C\}, \mathsf{tr}_{1,p}, \{\mathcal{P}_B, \mathcal{P}_C\})$
13. Perform $\mathsf{sign\_broadcast}(\{\mathcal{P}_A\}, \mathsf{tr}_{0,S}, \{\mathcal{P}_A, \mathcal{P}_B\})$
14. Perform $\mathsf{sign\_broadcast}(\{\mathcal{P}_C\}, \mathsf{tr}_{1,S}, \{\mathcal{P}_B, \mathcal{P}_C\})$
15. Perform $\mathsf{sign\_broadcast}(\{\mathcal{P}_B\}, \mathsf{tr}_{0,S}, \{\mathcal{P}_A, \mathcal{P}_B\})$
16. Perform $\mathsf{sign\_broadcast}(\{\mathcal{P}_B\}, \mathsf{tr}_{1,S}, \{\mathcal{P}_B, \mathcal{P}_C\})$
17. $\mathcal{P}_A, \mathcal{P}_B$: $\Gamma = \Gamma \cup \{\gamma_{A,B}\}$; $\mathsf{activate}(\gamma_{A,B}, \mathsf{tr}_{A,B})$; $\mathsf{revoke}(\gamma_0)$
18. $\mathcal{P}_B, \mathcal{P}_C$: $\Gamma = \Gamma \cup \{\gamma_{B,C}\}$; $\mathsf{activate}(\gamma_{B,C}, \mathsf{tr}_{B,C})$; $\mathsf{revoke}(\gamma_1)$

19. $\mathcal{P}_A, \mathcal{P}_C$: $\Gamma = \Gamma \cup \{\gamma_{A,C}\}$; $\mathsf{activate}(\gamma_{A,C}, \mathsf{tr}_{A,C})$
20. Setup virtual channel: $\gamma^v = (\gamma_0,\ \gamma_0.t,\ \gamma_1.f,\ \gamma_1.t,\ \gamma_{A,B},\ \gamma_{B,C},\ \gamma_{A,C},\ \mathcal{P}_A,\ \mathcal{P}_B,\ \mathcal{P}_C),\ \delta,\ \mathsf{min}(\gamma_0.t,\ \gamma_1.t) - 2\Delta)$
21. State Update: $\Gamma^v = \Gamma^v \cup \{\gamma^v\}$ and $\mathsf{Tr}^v = \mathsf{Tr}^v \cup \{(\gamma^v, \mathsf{tr}_{0,S}, \mathsf{tr}_{0,S}.t), (\gamma^v, \mathsf{tr}_{1,S}, \mathsf{tr}_{1,S}.t), (\gamma^v, \mathsf{tr}_{\mathsf{mrg}}, \mathsf{tr}_{\mathsf{mrg}}.t), (\gamma^v, \mathsf{tr}_{0,p}, \mathsf{tr}_{0,p}.t), (\gamma^v, \mathsf{tr}_{1,p}, \mathsf{tr}_{1,p}.t)\}$
22. Return message (SUCCESS, sid, msg) to $\mathcal{Z}$

VC-Close: Executed upon receiving message $\mathsf{msg} = (\text{CLOSE}, sid, \gamma^v,)$ from $\mathcal{Z}$. Let $\mathcal{P}_A = \gamma^v.\mathcal{P}_A$, $\mathcal{P}_B = \gamma^v.\mathcal{P}_B$ and $\mathcal{P}_C = \gamma^v.\mathcal{P}_C$. Let $(\gamma_0,\ \gamma_1,\ \gamma_{A,B},\ \gamma_{B,C},\ \gamma_{A,C},\ \mathcal{P}_A,\ \mathcal{P}_B,\ \mathcal{P}_C,\ \delta,\ t) = \gamma^v$; $\mathcal{P}_{CMP} = \{\mathcal{P}_A,\ \mathcal{P}_B,\ \mathcal{P}_C\} \setminus \{\mathcal{P}\}$. Do:

1. Party is addressed: Check $\mathcal{P} \in \{\mathcal{P}_A, \mathcal{P}_B, \mathcal{P}_C\}$. Otherwise ignore request.
2. Perform $\mathsf{consent\_verification}(\mathcal{P}_{CMP}, \mathsf{msg})$
3. Verify:
   - $\gamma^v \in \Gamma^v$; $\{\gamma_{A,B}, \gamma_{B,C}, \gamma_{A,C}\} \subseteq \mathcal{F}^v_{\mathsf{PWCH}}.\Gamma$; $t > \tau$; $\gamma_0.t_0 > \tau + 2\Delta$, $\gamma_1.t_0 > \tau + 2\Delta$
4. $\mathsf{revoke}(\gamma_{A,B})$, $\mathsf{revoke}(\gamma_{B,C})$, $\mathsf{revoke}(\gamma_{A,C})$
5. For $\mathcal{P}_A, \mathcal{P}_B$ do: $\mathsf{tr}_{0,r} = \mathsf{REFUND\_TR}(\gamma_0.f,\ \gamma_0.t - 2\Delta,\ \mathsf{sum}_A,\ \mathsf{sum}_B)$ where $\mathsf{sum}_A = \gamma_{A,B}.b_A + \gamma_{A,C}.b_A$ and $\mathsf{sum}_B = \gamma_{A,B}.b_B + \gamma_{A,C}.b_B$
6. For $\mathcal{P}_B, \mathcal{P}_C$ do: $\mathsf{tr}_{1,r} = \mathsf{REFUND\_TR}(\gamma_1.f,\ \gamma_1.t - 2\Delta,\ \mathsf{sum}'_B,\ \mathsf{sum}_C)$ where $\mathsf{sum}'_B = \gamma_{B,C}.b_A + \gamma_{A,C}.b_A$ and $\mathsf{sum}_C = \gamma_{B,C}.b_B + \gamma_{A,C}.b_B$
7. Perform sign_broadcast$(\{\mathcal{P}_A\}, \mathsf{tr}_{0,r}, \{\mathcal{P}_A, \mathcal{P}_B\})$
8. Perform sign_broadcast$(\{\mathcal{P}_C\}, \mathsf{tr}_{1,r}, \{\mathcal{P}_B, \mathcal{P}_C\})$
9. Perform sign_broadcast$(\{\mathcal{P}_B\}, \mathsf{tr}_{0,r}, \{\mathcal{P}_A, \mathcal{P}_B\})$
10. Perform sign_broadcast$(\{\mathcal{P}_B\}, \mathsf{tr}_{1,r}, \{\mathcal{P}_B, \mathcal{P}_C\})$
11. $\mathcal{P}_A, \mathcal{P}_B$:
    (a) $\gamma_0 = (\gamma_0.f, \mathcal{P}_A, \mathcal{P}_B, b_A, b_B, \gamma_0.t - 2\Delta, \gamma_0.t_0)$
    (b) $\mathsf{activate}(\gamma_0, \mathsf{tr}_{0,r})$
12. $\mathcal{P}_B, \mathcal{P}_C$:
    (a) $\gamma_1 = (\gamma_1.f, \mathcal{P}_B, \mathcal{P}_C, b_B, b_C, \gamma_1.t - 2\Delta, \gamma_1.t_0)$
    (b) $\mathsf{activate}(\gamma_1, \mathsf{tr}_{1,r})$
13. Update internal state: $\Gamma^v = \Gamma^v \setminus \{\gamma^v\}$

14. Return message $(\text{SUCCESS}, sid, \mathsf{msg})$ to $\mathcal{Z}$

VC-Enforce: Triggered upon receiving $\mathsf{msg} = (\text{ENFORCE}, sid, \gamma^v)$ from party $\mathcal{Z}$ where γ^v is a lightweight virtual payment channel. Let $\mathcal{P}_A = \gamma^v.\mathcal{P}_A$, $\mathcal{P}_B = \gamma^v.\mathcal{P}_B$ and $\mathcal{P}_C = \gamma^v.\mathcal{P}_C$.

1. Check there is an enforceable mapping: Let $\mathsf{TR}_\gamma = (\gamma, \mathsf{tr}) \in \{(\gamma', \mathsf{tr}') \mid (\gamma', \mathsf{tr}') \in \mathsf{TR}^v, \gamma' = \gamma, \mathsf{tr}'.t \leq \tau\}$.
2. $\forall (\gamma, \mathsf{tr}) \in \mathsf{TR}_\gamma$:
   - Let $\phi^{-1}(\mathsf{tr}) = (m, t, \Sigma)$, $(\mathsf{In}, \mathsf{Out}) = m$.
   - $\forall o \in \mathsf{In}$ send message (CHECK, sid, o) to $\mathcal{G}_{\mathsf{UTXO-Ledger}}$
   - **if** $\forall o \in \mathsf{In}$ it replies $(\text{CHECK\_OKAY}, sid, o)$ **then** send message $(\text{TRANSACTION}, sid, \mathsf{tr})$ to $\mathcal{F}_{\mathsf{Script}}$.
   - Set $\Gamma^v = \Gamma^v \setminus \{\gamma^v\}$
3. Return message $(\text{SUCCESS}, sid, \mathsf{msg})$ to $\mathcal{Z}$

4.10 Simulation Based Security Proof

In the following we provide simulation based proof of the security of our protocols. First we construct simulators S_{PWCH} and S_{VLPC}. Thereafter, using those we introduce and prove security as stated in Theorems 4.10.1 and 4.10.2 below.

Simulator S_{PWCH}

State: Simulates protocol PWCH creating the internal states of each party and maintaining their view. Moreover it stores a set of corrupted parties COR.

Initialization: Creates and initializes internal state of each of the simulated parties. At beginning of execution the adversary can corrupt any parties in which case the simulator will leak the corrupted parties' internal state to the adversary and stores their identities in COR. Upon request from the functionality, the simulator responds with the set of corrupted parties COR.

Behavior: Whenever the functionality leaks a message with sender and receiver appended, S_{PWCH} simulates sending of that message by the sender to the receiver. If the message's receiver is a corrupted party, S_{PWCH} forwards the message to the respective party. Whenever any simulated party or the adversary send a message to a functionality, i.e. $\mathcal{F}_{Script}$, $\mathcal{F}_{SIG}$, $\mathcal{G}_{UTXO-Ledger}$, $\mathcal{G}_{\mathsf{CLOCK}}$ or to $\mathcal{Z}$, the simulator leaks it to $\mathcal{F}_{\mathsf{PWCH}}$ annotating sender and receiver. If the sending entity expects a reply the simulator waits for $\mathcal{F}_{\mathsf{PWCH}}$ to leak it.

Simulator S_{VLPC}

State: Simulates protocol $\mathsf{LVPC}_{\mathsf{PWCH}^v}$ creating the internal states of each party and maintaining their view. Moreover it stores a set of corrupted parties COR.

Initialization and **Behavior** are analogous to S_{PWCH}, however it interacts with $\mathcal{F}_{\mathsf{LVPC},\mathcal{F}^v_{\mathsf{PWCH}}}$ instead of $\mathcal{F}_{\mathsf{PWCH}}$.

Theorem 4.10.1. *Protocol* PWCH *realizes* $\mathcal{F}_{\mathsf{PWCH}}$ *in the (*$\mathcal{G}_{\mathsf{CLOCK}}$*,* $\mathcal{G}_{\mathsf{UTXO-Ledger}}$*,* $\mathcal{F}_{\mathsf{SIG}}$*,* $\mathcal{F}_{\mathsf{Script}}$*) - hybrid world*

Sketch of Proof. First, we show that the probability that the simulated parties and parties in the hybrid world have different state changes with the same requests from $\mathcal{Z}$ is in $\mathcal{O}(negl(n))$. The probability that the channel states stored by $\mathcal{F}_{\mathsf{PWCH}}$ is different to those stored by the parties is in $\mathcal{O}(negl(n))$. The probability that the global functionality $\mathcal{G}_{\mathsf{UTXO-Ledger}}$ has different state changes depending on whether $\mathcal{Z}$ sends the same requests to either the ideal or the hybrid world is in $\mathcal{O}(negl(n))$. We argue that following this the simulation by S_{PWCH} is indistinguishable from the execution in the hybrid world and $\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}} \approx_c \mathsf{HYBRID}_{\pi^{\mathcal{F}_{aux}},\mathcal{A},\mathcal{Z}}$.

We briefly handle the case that two parties corrupted by the adversary are instructed to setup a pairwise payment channel. Note that $\mathcal{F}_{\mathsf{PWCH}}$ is aware on which parties are corrupted by the adversary by inquiring this information from S_{PWCH}. In this case $\mathcal{F}_{\mathsf{PWCH}}$ forwards requests from $\mathcal{Z}$ to S_{PWCH} where they are handed to the adversary, but ignores them otherwise. Any communication between corrupted parties and $\mathcal{G}_{\mathsf{CLOCK}}$, $\mathcal{G}_{\mathsf{UTXO-Ledger}}$, $\mathcal{F}_{\mathsf{SIG}}$, $\mathcal{F}_{\mathsf{Script}}$ or $\mathcal{Z}$ are forwarded by S_{PWCH} and $\mathcal{F}_{\mathsf{PWCH}}$ to the appropriate interfaces, resulting in the same state changes in $\mathcal{G}_{\mathsf{UTXO-Ledger}}$ and the same messages received by corrupted parties and the adversary as in the hybrid world.

In the following we assume that all pairwise payment channel created by instructions of $\mathcal{Z}$ have at least one party participating that is not corrupted by the adversary. Any requests sent by $\mathcal{Z}$ are forwarded by $\mathcal{F}_{\mathsf{PWCH}}$ to S_{PWCH} to be simulated such that all requests are received in the simulation. The same requests are permissible in either world, either by having access to the same functionalities, as $\mathcal{F}_{\mathsf{SIG}}$, or by $\mathcal{F}_{\mathsf{PWCH}}$ providing an interface for the sub-protocols in PWCH. Each request is subject to the same checks in either world. Upon each request, the functionality as well as the protocol verify consent between all honest parties. Consent of corrupted parties is implicit by their cooperation or lack thereof. Afterwards checks on the parameter within a request are performed which are the same. Attention has to be paid to verification that an $\mathsf{utxo} \in \mathsf{UTXO}$ is logged on the ledger. These require sending messages to $\mathcal{G}_{\mathsf{UTXO-Ledger}}$ tagged with check. The replies on these messages are time-dependent as messages tagged to $\mathcal{G}_{\mathsf{UTXO-Ledger}}$ tagged $\mathsf{transaction}$ can alter its state at any time and the adversary might try to delay execution of the protocol to provoke receiving different replies from $\mathcal{G}_{\mathsf{UTXO-Ledger}}$. However, $\mathcal{F}_{\mathsf{PWCH}}$ waits with sending any such message for the simulation to catch up such that they are

sent at the same time and the same replies are processed by functionality and simulation. The same holds for messages sent to $\mathcal{G}_{\mathsf{UTXO-Ledger}}$ tagged transaction in sub-functions open and enforce to account for delays created by the adversary. At the end of execution, successful checks have to lead to the same state changes to be applied between the simulated parties and the analogous state in $\mathcal{F}_{\mathsf{PWCH}}$. However, a corrupted party might deviate from the protocol in which case any honest party will abort the sub-protocol. If such behavior is observed any honest party will send a message (FAILURE, sid, msg) to $\mathcal{Z}$ where msg is the message with $\mathcal{Z}$'s request which is forwarded to $\mathcal{F}_{\mathsf{PWCH}}$ by the simulator. Respectively $\mathcal{F}_{\mathsf{PWCH}}$ will abort execution of the respective sub-function and forward the message to the respective dummy party. However, if no such behavior was observed any simulated honest party will output a message to $\mathcal{Z}$ of form (SUCCESS, sid, msg) which indicates to $\mathcal{F}_{\mathsf{PWCH}}$ that the respective party finished execution by the sub-protocol including performing a state change. Only then $\mathcal{F}_{\mathsf{PWCH}}$ performs an analogous state change.

In the enforce sub-function a refund mapping is created on the fly depending on the channel's state. This is not possible in the protocol as a corrupted party could refuse collaboration. Respectively, in the protocol when a channel is setup as well as when a channel changes state, a refund transaction representing the latest state has to be created and signed by both involved parties to be able to execute enforce unilaterally by any honest party. We note that whether execution of enforce results in applying a state change to the ledger depends on when it is executed. A honest party always attempts to apply a channel's latest state to the ledger. If enforce is instructed as soon as a channel γ's lifetime expires, i.e. at time $\gamma.t$, this will always result in the appropriate state change as there is no transaction with timelock of less than $\gamma.t + \Delta$. However, if enforce is executed later, a corrupted party might attempt to send a transaction representing an older channel's state applying it to the ledger. Nevertheless, as such a transaction would be simply forwarded by $\mathcal{F}_{\mathsf{PWCH}}$ this would be the same in either ideal or hybrid world.

Lastly, an adversary can attempt to forge the signature of a honest party to create a transaction that spends a channel's Funding utxo. However it is shown that the probability for this is in $\mathcal{O}(negl(n))$ [30].

Theorem 4.10.2. *Protocol* $\mathsf{LVPC}_{\mathsf{PWCH}}$ *realizes* $\mathcal{F}_{\mathsf{LVPC},\mathcal{F}_{\mathsf{PWCH}}}$ *in the* ($\mathcal{G}_{\mathsf{CLOCK}}$, $\mathcal{G}_{\mathsf{UTXO-Ledger}}$, $\mathcal{F}_{\mathsf{SIG}}$, $\mathcal{F}_{\mathsf{Script}}$) *- hybrid world.*

Sketch of Proof. The proof is analogous to the proof of Theorem 4.10.1, however, in addition we need to analyze the interaction between the parties because we moved from a protocol between two parties to a protocol between three parties.

Functionality $\mathcal{F}_{\mathsf{LVPC},\mathcal{F}^{v}_{\mathsf{PWCH}}}$ creates and stores all mappings of a virtual channel construction and creates, re-activates and disables pairwise payment channels simultaneously. However, corrupted parties might try to create only a subset of transactions by selectively providing or withholding signatures of transactions. In the protocol, the order in which transactions are signed enforces that from

the perspective of a honest party, either all transactions they hold are fully signed or none of the transaction created in a sub-protocol can be committed to the ledger before it successfully terminates.

In the case of the VC-Open protocol, all transactions a honest party is involved in creating depends on whether the split transactions it holds can be committed to the ledger. Respectively the party needs to make sure that it holds all other transactions fully signed before proceeding to sign any split transactions. As the intermediate party holds two split transactions, it has to make sure that either all or no split transaction is signed. For this reason it has to wait signing split transactions and broadcasting these signatures until it received signatures for each split transaction from the respective counterparties. For the same reason, in the case of the VC-Close protocol, the intermediate party has to make sure that it holds the signatures of refund transactions of the reactivated pairwise payment channel before signing any of these itself to ensure that either both or no pairwise payment channel is reactivated.

4.11 Conclusion

We use timelocks to create an order in which transactions in our constructions are valid. However, different techniques for invalidating transactions or replacing transactions offchain might be used instead to have less restrictions on the lifetime of a virtual channel. For this we can adapt techniques as introduced for the Lightning Network [112] or eltoo [108].

Lastly we argue that our construction provides incentive for research into route discovery protocols that yield multiple paths. We reason that while virtual channel expand the payment channel network topology, they also reduce the total capacity of the channels involved. Even though virtual channel allow for shorter paths as well as increase the number of possible paths from a source to a target, larger payments that exceed the capacity of a virtual channel have to be routed through multiple paths to be able to ulitize this, for instance using the AMP protocol [104].

Chapter 5

Payment Trees

5.1 Introduction

Blockchain based decentralized ledgers as introduced by Nakamoto [100] have enjoyed popularity and received interest from the research community and practitioners. Consensus protocols allow these ledgers to be operated by mutually distrustful parties at the cost of limited throughput. For example, Visa as a centralized system can process orders of magnitude more transactions within a given time frame than the most prominent blockchains as Bitcoin and Ethereum.

The main motivation for the development of offchain protocols is to close the gap in transaction throughput. The idea is to allow parties to interact with each other without interacting with the ledger, while still being able to use it to resolve disputes. Offchain protocols operate on *channels* that are created between two parties. Channels hold a state which can be enforced on the ledger. Payment channels [44, 108, 112] store the number of coins the two parties have locked inside that channel. Offchain protocols provide a means to alter this state arbitrarily often and thus improving the transaction throughput in the overall system.

Individual channels can be extended to channel networks, e.g. PCNs Lightning [112] and Raiden [7]. This is done using techniques, such as HTLC [25, 112], that allow for payments of $b \in \mathbb{N}$ coins across a path of payment channels of length $n \in \mathbb{N}$. This is performed by executing the same payment on each channel within the payment path atomically. All parties on the payment path have to lock the payment amount for a duration of up to *locktime*. The opportunity cost a party has to invest is the *collateral* [98] which equals the payment amount b multiplied by the locktime. In turn, parties can impose fees to invest collateral. In the case of HTLC, a party's collateral equals $\mathcal{O}(nb\Delta)$ in the worst-case where Δ is a parameter of the underlying ledger and is the upper limit of the time it takes for a transaction to be included in the ledger.

High collateral investments can be exploited by malicious adversaries to perform *grieving* and *denial-of-service* attacks[99, 109]. For example, an attacker

might operate a channel to collect fees by forwarding payments. However, payments might be routed through competing channels instead. To sabotage the competitor, the attacker can route a payment through these channels without the intent of executing it, locking the competing channel's coins for the entirety of the locktime. These channels experience a denial-of-service scenario by being unable to forward any other payments, losing fees that the attacker can collect through their own channel. Performing this attack on a large scale can result in denial-of-service for the whole PCN. On a lower scale, a griever might force parties to lock away their funds for as long as possible by delaying their cooperation until the last moment. An alternative form of this attack involves routing multiple low value payments through a competing channel, up until a point where the channel cannot add any further HTLCs even though it contains enough coins. In the case of the Lightning network, these types of denial-of-service attacks can lock all of a channel's coins for up to around 2 weeks [99]. [1]

For HTLC the total collateral locked over a whole payment path is $\mathcal{O}(n^2 b\Delta)$ and therefore quadratic in the payment paths length. Sprites [98] reduce the collateral of each party to $\mathcal{O}(b(n+\Delta))$ and the total collateral to $\mathcal{O}(bn(n+\Delta))$ by utilizing a smart contract. This is considered to be constant and linear respectively, since $n << \Delta$ such that $n + \Delta < 2\Delta$. Sprites mitigate the damage done by a possible attacker but its implementation is limited to ledgers with smart contract capability. The Atomic Multi-Channel Updates (AMCU) protocol [51] is an attempt to close this gap and enable payments with constant collateral on ledgers without smart contract capabilities. However, even though AMCU is formalized as a functionality within Canetti's UC Framework [31], the very last, but crucial step, of the updateState function *does not seem to be presented* in the description of the AMCU protocol, and neither addressed by the simulator [51]. This gap results in a vulnerability that can be exploited by a malicious adversary to steal funds from honest parties.

Related Work. Payment channels [44, 108, 112] themselves allow only for offchain payments between two parties. Offchain protocols such as HTLCs [25, 112] and Sprites [98] allow to perform payments across paths of channels allowing for the implementation of PCNs. Prominent examples are the Lightning Network [112] and Raiden [7]. Although offchain protocols exist that create new *virtual* channels out of two existing channels as Perun [47, 50] and Lightweight Virtual Payment Channels [66], this work focuses on performing individual payments across a PCN. In the following we consider a payment of $b \in \mathbb{N}$ coins across a path of $n \in \mathbb{N}$ channels involving parties $\mathcal{P}_0, \ldots, \mathcal{P}_n$.

The most prominent technique is based on HTLCs [25, 112], which are scripts that perform conditional payments within a channel: The payer locks funds into the contract that are paid out if the payee can present a secret x such that $y = \mathcal{H}(x)$ where $\mathcal{H}$ is a cryptographic hash function. Otherwise, after time

[1] https://cointelegraph.com/news/developer-reveals-biggest-unsolvable-lightning-attack-vector.

locktime the payment times-out and the payer can reclaim their funds. This contract is replicated along all channels within a payment path. The payment is performed as soon as $\mathcal{P}_n$ reveals x to their predecessor who then learns the value of x allowing them to claim the payment from their predecessor in turn. An attacker $\mathcal{P}_i, 0 < i \leq n$ might attempt to delay revelation of x to their predecessor until briefly before expiration of the *locktime*. To allow $\mathcal{P}_{i-1}$ to forward x in time, their locktime needs to be increased by at least Δ. This results in a locktime in $\mathcal{O}(n\Delta)$ and a total locktime in $\Theta(n^2\Delta)$.

Sprites [98] aim to reduce the locktime of a party up to a constant $\mathcal{O}(n+\Delta)$ where $n << \Delta$. This is done by setting up a smart contract entity called *PreimageManager*, s.t. submitting x to the PreimageManager allows to broadcast it to all nodes within a payment path in at most n communication rounds. The protocol requires creation of a smart contract, making it unavailable to script based ledgers as Bitcoin. AMCU [51] attempts to close this gap, i.e. compatibility with Bitcoin, by introducing an approach for constant locktime payments without the need of smart contracts. AMCU sets up payments on each channel within a payment path that are performed on the condition that an *Enable* transaction is created, upon which all payments are performed atomically. However, this Enable transactions results in several issues. For one, its size grows linearly in the payment path's length, making its implementation prohibitive for ledgers which have an upper limit for block size and transaction size. Moreover, no party has control over all of the Enable transaction's inputs. A malicious adversary can make two parties collaborate to double spend one of the Enable transaction's inputs, such that no party is able to enforce the payment on the ledger. If the double-spending is timed appropriately, this can lead to an attacker stealing funds from honest parties. Details are shown in Appendix 5.3.3.

Jourenko et al. [66] proposed an offchain protocol that takes two channels γ_A and γ_B as input, one between $\mathcal{P}_A$ and $\mathcal{P}_I$ and one between $\mathcal{P}_I$ and $\mathcal{P}_B$ and creates a new channel γ^v between $\mathcal{P}_A$ and $\mathcal{P}_B$. As this approach is not optimized for individual payments, using it for this purpose would result in excessive collateral as parties would need to lock away more coins for a longer duration as in existing approaches. However, we re-use techniques from the lightweight virtual payment channel construction for the Payment Tree protocol.

Our Contributions. Our contributions are threefold. 1) We present an attack on AMCU performed by a malicious adversary. 2) We present *Payment Trees* that allow for payments across paths within a PCN without the need of smart contracts, requiring *only* logarithmic individual collateral $\mathcal{O}(b\Delta \log n)$ while requiring only linear total collateral $\mathcal{O}(nb\Delta)$ such that its performance is comparable to Sprites. 3) We provide efficiency and security analysis of Payment Trees, proving the properties *Balance Security* and *Liveness*.

Structure. In the remainder of this work, first, we provide background to this work in Section 5.2. We give an outline of the Channel Closure attack in Section 5.3 while supplementing a formal description in Appendix 5.3.3. Next, we give

an informal overview of the Payment Tree protocol in Section 5.4. Afterwards, we introduce the types of transactions used for our construction in Section 5.5 before introducing Payment Trees in Section 5.6 followed by efficiency and security analysis in Section 5.7. We conclude in Section 5.8.

5.2 Background

5.2.1 Hashed Timelock Contracts

Hashed Timelock Contracts. Let $\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_n, n \in \mathbb{N}$ be parties where parties $\mathcal{P}_{i-1}$ and $\mathcal{P}_i, i \in \{1, \ldots, n\}$ control channel γ_i. HTLCs are used to perform payments of $b \in \mathbb{N}$ coins from $\mathcal{P}_0$ to $\mathcal{P}_n$ by replicating the payment on each channel γ_i within a payment path $\gamma_1, \ldots, \gamma_n$ from $\mathcal{P}_0$ to $\mathcal{P}_n$. (1) On a channel γ_j, $j \in \{1, \ldots, n\}$ the payment is performed by extending the channel-tree with a conditional payment: If the payee $\mathcal{P}_j$ can show the pre-image $x \in \mathbb{N}$ of a hashed value $y = H(x)$, where H is a cryptographic hash function, they will receive b coins from the payer $\mathcal{P}_{j-1}$. However, after expiration of a locktime t_j the payment expires and the payer $\mathcal{P}_{j-1}$ will have their coins refunded instead. (2) Only after the conditional payments are set up on all channels, the payment is executed atomically by having $\mathcal{P}_n$ show the pre-image x to $\mathcal{P}_{n-1}$, proving that they have the *capability* to claim the coins on the ledger through the conditional payment. In turn, $\mathcal{P}_{n-1}$ learns the pre-image x s.t. they can show it to party $\mathcal{P}_{n-2}$ reclaiming the coins they forwarded to $\mathcal{P}_n$. The information on x propagates through the whole payment path in this manner. (3) Lastly, to keep the payment offchain, the parties need to consolidate the payment on each channel respectively. This is done by updating the channel-tree. The conditional-payment is removed and the b coins that were locked into the channel are credited to the payee. At this point the channel-tree has the same form as before the payment, but with updated balance distribution to account for the payment. This ensures that the channel-tree does not grow in size with each payment, thus fulfilling the efficiency requirements of an offchain protocol. Note that, if the payer $\mathcal{P}_{j-1}$ does not cooperate with consolidation, payee $\mathcal{P}_i$ can reclaim their coins by resolving the conditional payment on the ledger instead. Due to this the timelock t_j has to be chosen s.t. $\mathcal{P}_i$ has enough time to do so before the conditional payment expires, even if they learn the pre-image from $\mathcal{P}_{i+1}$ at the last moment just shortly before expiration of timelock t_{j+1}. Thus the relation $t_i \geq t_{i+1} + \Delta$ has to hold, making the locktime grow linearly with the payment path's length. This results in a collateral cost of $bt_i \in \mathcal{O}(bn^2\Delta)$ which is quadratic in the path's length.

The Wormhole Attack. The HTLC protocol is vulnerable to the wormhole attack [90]. An adversary controlling two parties $\mathcal{P}_i$, $\mathcal{P}_j$, $1 \leq i \leq j + 2 \leq n - 1$ within a payment path can prevent intermediaries $k, i < k < j$ to participate at the payment and receive their fees by having $\mathcal{P}_i$ forward pre-image x to $\mathcal{P}_{i-1}$ after $\mathcal{P}_j$ learns it from $\mathcal{P}_{j+1}$ and without forwarding it to party $\mathcal{P}_{j-1}$.

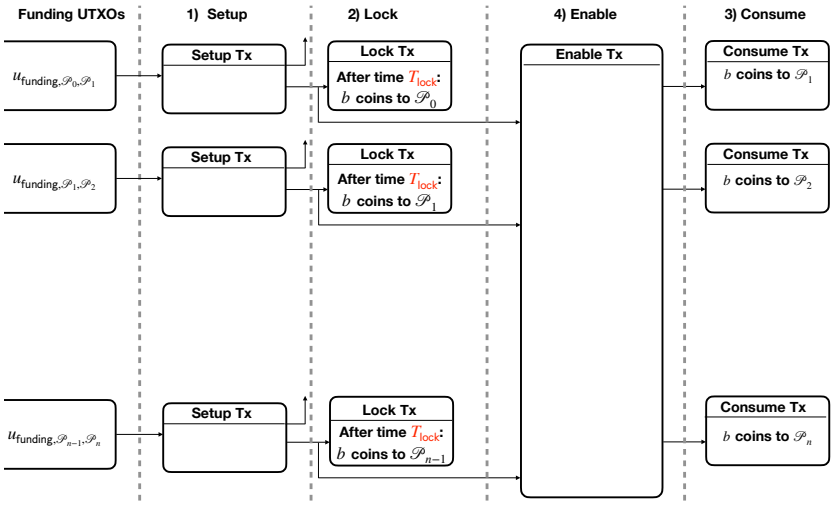


Figure 5.1: Informal illustration of AMCU as a transaction-tree.

5.2.2 Atomic Multi-Channel Updates

Although the Sprites protocol reduces a party's collateral to $\mathcal{O}(b(n+\Delta))$, AMCU is the first proposal to reduce the collateral for UTXO based ledger that do not use smart contracts.

To perform an offchain payment AMCU operates in four phases in which the transaction tree shown in Figure 5.1 is created. 1) In a Setup phase b coins from $\mathcal{P}_{i-1}$'s balance are split up from the channel using the Setup transaction. 2) In the Lock phase the Lock transaction is created which spends $\mathcal{P}_{i-1}$'s b coins from the Setup transaction and pays out all coins back to $\mathcal{P}_{i-1}$ after expiration of time T_{lock}. 3) In the Consume phase the parties create a Consume transaction paying the b coins to $\mathcal{P}_i$, however, instead of spending the Setup transaction it spends a not-as-of-yet created Enable transaction. 4) In the Finalize phase, first a Disable transaction is created that spends the Enable transaction after expiration of time T_{lock} and returns the b coins to $\mathcal{P}_{i-1}$ in the same manner as the Lock transaction does. Lastly the Enable transaction is created spending b coins of all Setup transactions on the payment path and creating the UTXOs that are spent by each channel's Consume and Disable transactions.

AMCU achieves atomic payment across the whole payment path by creating the Consume transactions to have the Enable transaction as common ancestor. As soon as it is signed, all Consume transactions can be committed to the ledger, thus rendering each payment on the payment path enforceable. However, this approach is impractical. The size of the Enable transaction grows as the number of its inputs and outputs increases and therefore its size grows linearly with the payment path's length n. Payments across long parts cannot be performed if it exceeds the limits of a transaction's size.

The security of the AMCU protocol is attempted to be proven within Canettis UC Framework [31] by presenting a simulator that shows that the AMCU protocol realizes an ideal functionality PCN^+. However, while its updateState function, that is used to perform payments across a payment path, concludes with a *consolidation* step that atomically applies the payment on each individual channel within the payment path, this step is skipped within the AMCU protocol and not addressed by the simulator. Exactly this gap between ideal functionality and protocol is the vulnerability that allows a malicious adversary to have corrupted parties potentially steal funds from honest parties. We introduce the *Channel Closure attack* formally in Appendix 5.3.3, in which a pair of intermediate parties within a payment path can steal funds from honest parties executing the AMCU protocol.

5.3 The Channel Closure Attack on AMCU

In the following we present the Channel Closure attack informally to present an overview of the approach. A formal definition, discussion and cost analysis of the attack is provided in Section 5.3.3.

5.3.1 The Vulnerability

While the Enable transaction is the core of the AMCU construction, it also seems to be its vulnerability. While the Enable transaction receives inputs from each channel, no party has control over all channels within the payment path. At any time, two parties sharing a channel can maliciously spend a UTXO that is provided as input of the transaction, or as input to any of its ancestors within the transaction tree. When this happens, the Enable transaction cannot be committed to the ledger and all parties have their coins refunded through Lock transactions. Effectively, no party can enforce payment after execution of the AMCU protocol. On top of that, an adversary can take this further, performing a Channel Closure attack to steal funds from honest parties. We remark that PCN payments require a consolidation step in which a payment is included within the parties' individual channels. While the functionality PCN^+ modeling AMCU performs a consolidation step atomically on all channels, this step is omitted by the AMCU protocol. Second, performing the consolidation step atomically on all channels is highly non-trivial as atomic operations on multiple channels is exactly the problem statement that protocols such as HTLCs, Sprites and AMCU themselves attempt to solve.

5.3.2 Intuition on The Channel Closure Attack

The attack is performed by abusing exactly the two observation from the previous section. First, the adversary corrupts two parties within a payment path $\mathcal{P}_i$ and $\mathcal{P}_{i+1}$. These parties cooperate in execution of the AMCU protocol right up until the consolidation step. Then, $\mathcal{P}_i$ performs the consolidation step with

$\mathcal{P}_{i-1}$ on channel γ_{i-1} while $\mathcal{P}_{i+1}$ does not cooperate with $\mathcal{P}_{i+2}$ to consolidate the payment on channel γ_{i+1}. Next, $\mathcal{P}_i$ and $\mathcal{P}_{i+1}$ close their channel γ_i such that the Enable transaction cannot be committed to the ledger. This allows $\mathcal{P}_{i+1}$ to reclaim coins from $\mathcal{P}_{i+2}$ using their shared Lock transaction. Effectively, $\mathcal{P}_i$ received the payment amount from $\mathcal{P}_{i-1}$ on γ_i through consolidation, while $\mathcal{P}_{i+1}$ did not forward the payment.

5.3.3 The Formal Channel Closure Attack

Informally, an adversary can attack AMCU by corrupting two parties $\mathcal{P}_i$ and $\mathcal{P}_{i+1}$ that share channel γ_i along a payment path. First, parties cooperate in execution of the protocol right until after creation of the Enable transaction at which point the protocol concludes. We observe that if the protocol is not followed up by a consolidation step as in the ideal functionality PCN^+, $\mathcal{P}_i$ and $\mathcal{P}_{i+1}$ can close their channel γ_i maliciously, e.g. by double-spending the UTXO used as input into their Setup transaction. This prohibits commitment of the Setup transaction to the ledger and, as it is the ancestor of the Enable transaction which in-turn is common ancestor of all Consume transactions, no Consume transaction can be committed to the ledger, effectively reverting the payment. After the execution of the protocol, no party can enforce the payment by committing the Consume transactions. Performing the payment requires a final consolidation step, as defined in functionality PCN^+, allowing any party to enforce the payment on their ledger through the channels they participate in.

It is essential that the consolidation step is done atomically on all channels within a payment path, as otherwise this could lead to honest parties losing funds. However, this step is non-trivial as performing a state transition on multiple channels atomically is the very problem statement HTLCs, Sprites and AMCU approach to solve. In the following we present the *Channel Closure attack* that allows a malicious adversary to have corrupted parties steal funds from honest parties as long as at most $n-2$ out of n channels are consolidated atomically.

In the following let $\mathcal{P}_0, \ldots, \mathcal{P}_n$ be parties where parties $\mathcal{P}_{i-1}$ and $\mathcal{P}_i, i \in \{1, \ldots, n\}$ control channel γ_i. Let S_i be the setup transaction and L_i be the Lock transaction for channel γ_i respectively. The parties perform a payment of $b \in \mathbb{N}$ coins over the payment channel path $\gamma_1, \ldots, \gamma_n$ using the AMCU protocol. If the adversary can influence the order of channel consolidation then the attack can be performed with $n \geq 2$ where at most $n-1$ channels are consolidated atomically. Otherwise, we require $n \geq 3$ where at most $n-2$ channels are consolidated atomically.

The Channel Closure Attack

The Adversary. The adversary is created according to AMCU's adversarial model. At beginning of the protocol, the adversary can corrupt $n-1$ parties s.t. it receives the party's internal state and all subsequent incoming and outgoing communication is routed through them instead. This corruption is static and

the adversary cannot switch corrupted parties or corrupt any additional parties during execution of the protocol. The adversary is malicious and can deviate from the protocol arbitrarily, however, it is computationally polynomially bounded.

The adversary succeeds if the set of corrupted parties holds strictly more funds compared to when all Consume transactions are committed to the ledger.

The Approach. The corrupted parties steal coins by, first, executing the protocol correctly until the consolidation phase. They pick a party $\mathcal{P}_i, i \in \{1, \ldots, n\}$ where channel γ_i is consolidated before γ_{i+1}. After they receive coins through consolidation of γ_i, two parties γ_{j-1} and γ_j close channel j such that $\mathcal{P}_i$ has their money returned through the Lock transaction L_{i+1} instead of forwarding the coins. There are a few edge cases: (1) If $i = j + 1$ then channel γ_{i+1} is controlled by the corrupted parties, so we require γ_i is consolidated before γ_{i+2} instead; (2) if $i = n$ then $\mathcal{P}_i$ is already the payment's recipient. In this case, we require $\mathcal{P}_0$ to be corrupted as well, such that $\mathcal{P}_n$ receives their funds before $\mathcal{P}_0$ pays them out.

Channel Closure Attack. The adversary picks $i, j \in \{1, \ldots, n\}, i \neq j$ such that following conditions hold: (1) If $i \neq j - 1$, then γ_i is consolidated before $\gamma_{(i+1) \mod n}$, otherwise γ_i is consolidated before $\gamma_{(i+2) \mod n}$; (2) if $\gamma_{(j+1) \mod n}$ is consolidated before $\gamma_{(i+1) \mod n}$ then $\gamma_{(j-1) \mod n}$ is consolidated before the channel $\gamma_{(i+1) \mod n}$. The adversary corrupts $\mathcal{P}_i$, $\mathcal{P}_{j-1}$ and $\mathcal{P}_j$. If $i = n$ the adversary also corrupts $\mathcal{P}_0$. Upon starting the protocol, the adversary behaves honestly and collaborates with the execution of the AMCU protocol up until the Consolidation step. After $\mathcal{P}_i$ receives funds through consolidation of γ_i, they do not respond to any parties requesting consolidation of their channels, but instead the adversary orders $\mathcal{P}_{j-1}$ and $\mathcal{P}_j$ to close γ_j by spending the UTXO that is the input of their Setup transaction S_j.

Discussion

In the general case the adversary needs to corrupt at least 4 parties, thus the attack requires $n \geq 4$. However, if the adversary can influence the order in which channels are consolidated, in the case of $n \geq 3$ they can always pick $1 \leq i = j-1 \leq n-2$ and reduce the parties they need to corrupt to 2. Moreover, note that if the order in which channels are consolidated is not known a-priori, the adversary has to guess values for i and j. We assume the adversary picks values for i and j randomly out of a uniform distribution of all possible values, i.e. $1, \ldots, n$. The probability to guess one out of n parties for the value for i equals at least $1/n$. As $i \neq j$, the value of j has to be guessed out of $n - 1$ parties which equals a probability of at least $1/(n - 1)$. Thus the probability for the adversaries success is at least $1/((n - 1)n)$ which is not negligible.

Cost Analysis. Performing the attack requires the adversary to commit transactions to the ledger which incurs costs to the adversary. As such the attack

is only attractive to a rational adversary if it would not result in a net loss. In the following we denote the adversary's cost by the amount of transactions that have to be committed to the ledger. We observe that, to perform the attack the adversary picks two channels γ_i and γ_j and succeeds if they guessed the order in which channels are consolidated correctly. If the adversary fails to guess correctly they can abort the attack and proceed with the protocol honestly and avoid cost. Otherwise the adversary has to spent the Funding UTXO of γ_j requiring one transaction. In the highest cost scenario the adversary has to commit all Lock transactions they are involved in to the ledger. In the worst case the adversary has to corrupt 4 parties, i.e. $\mathcal{P}_i$, $\mathcal{P}_{i+1}$, $\mathcal{P}_{j-1}$, $\mathcal{P}_j$, such that they are involved in up to three Lock transactions, i.e. L_{i+2}, L_{j-1}, L_{j+1}. As γ_i itself is already consolidated L_i is not committed to the ledger. Moreover since the adversary controls channels γ_{i+1} and γ_j they can opt to modify the channel states arbitrarily offchain and avoid commitment of L_{i+1} and L_j. Committing a Lock transaction requires commitment of two transactions, i.e. the Setup and the Lock transaction itself. Overall, the adversary's cost equals committing up to $1 + 2 \cdot 3 = 7$ transactions to the ledger.

5.4 Protocol Overview

In the following, we define communication and adversarial models, before giving an overview of the protocol. Lastly we define the properties of our construction.

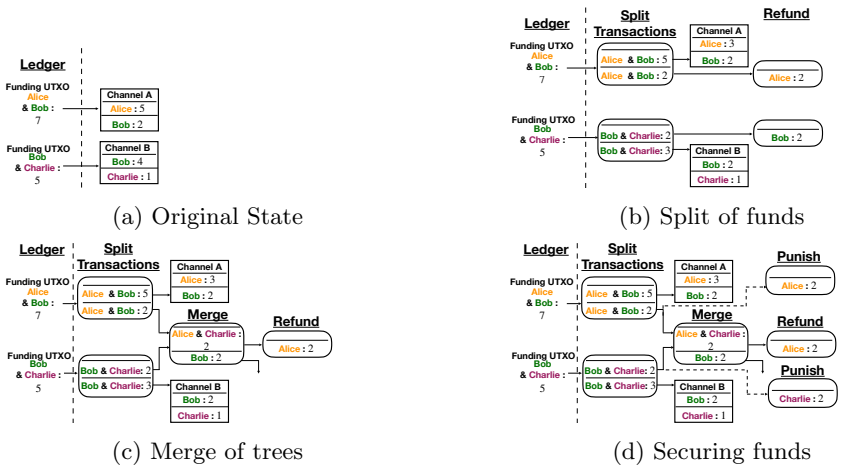
5.4.1 System Model

Communication Model. Communication between parties occurs in rounds. Any message sent within one round is available to the recipient at the beginning of the next round. The duration of any round has an upper limit.

Adversarial Model. We define an Adversary $\mathcal{A}$ consistent with related work [66, 51, 98]: At the beginning of protocol execution, the adversary can statically corrupt up to n of $n + 1$ parties, receiving their internal state and having all communication to and from these parties be routed through the adversary. The adversary is malicious and can make any corrupted party deviate from the protocol. Moreover, within each communication round, the adversary can delay and re-order all messages sent.

5.4.2 Overview

We illustrate the life-cycle of the Payment Tree protocol for a payment of 2 coins from Alice to Charlie across two channels using Figure 5.2 and Figure 5.3. The protocol's approach is to take two channels, one between parties Alice and Bob, one between parties Bob and Charlie and construct a transaction tree that effectively creates a virtual channel [66] optimized for a one-time payment between Alice and Charlie. Our construction utilizes two approaches to perform updates



(a) Original State

(b) Split of funds

(c) Merge of trees

(d) Securing funds

Figure 5.2: Stepwise construction of a Payment Tree across two channels. Boxes with straight corners represent channel trees displaying their state. Boxes with round corners represent transactions displaying output UTXOs. Edges indicate which transactions spend the UTXO at their origin.

to transaction trees atomically. On the one hand, we use these techniques to empower the intermediary Bob to ensure correctness of the protocol, while on the other hand, we incentivise Bob to actually do so by means of punishment. Our construction consists of multiple transaction tree updates. Updates are done using the *invalidation by timelock* technique, but for simplicity we leave the details to Section 5.6.

Payment Tree Construction. Figure 5.2 depicts construction of a Payment Tree between Alice, Bob and Charlie. Construction consists of three atomic transaction tree updates. We note that the balance distribution between the parties remains unchanged between the updates and no payment is executed. Alice and Bob as well as Bob and Charlie share a channel as depicted in Figure 5.2a. (1) Then, as shown in Figure 5.2b we update both trees by introducing a *Split* transaction that spends the channels' Funding UTXOs and creates two new Funding UTXOs each. One UTXO contains the payment amount and is funded by coins from Alice, who is payer, and Bob, who is intermediary, respectively. The other UTXO contains the remaining coins and is used as Funding UTXO to reopen both channels which can be used for further payments within the channels or further Payment Tree constructions. (2) Next as shown in Figure 5.2c both separate transaction trees are combined using a Merge transaction. This transaction creates two UTXOs. One UTXO requires Bob's signature to be spent and contains his collateral. The other UTXO is a Funding UTXO requiring the signatures of Alice and Charlie and it contains Alice's payment to Charlie. At this point, the coins are given to Alice. (3) Lastly, as shown

(a) Payment (b) Consolidation

Figure 5.3: Payment and Consolidation using Payment Trees. Figure 5.3a modifies the Payment Tree to forward the funds in the Merge transaction's Funding UTXO to the payee. Figure 5.3b splits up the Payment Tree and distributes funds according to the Payment Tree's state in Figure 5.3a.

in Figure 5.2d, before we can proceed with a payment, the funds within the Merge transaction's Funding UTXOs need to be secured in case two parties, for example Bob and Charlie, collude to spend their Merge transaction's or Split transaction's input with a different transaction. This attack is similar to the Channel Closure attack described in Section 5.3 and would disable commitment of the Merge transaction. However, we observe that all UTXOs that can be spent for this attack require Bob's signature. Respectively, in this scenario we can uniquely identify Bob as malicious. In order to punish Bob and secure the funds of Alice and Charlie respectively we create *Punish* transactions. These transactions spend the same Funding UTXOs as the Merge transaction but have a timelock that is higher than that of the Merge transaction by at least Δ. Due to this, Bob can always avoid commitment of a Punish transaction by committing the Merge transaction to the ledger. However, in case Bob acted maliciously such that the Merge transaction cannot be committed to the ledger, Alice and Charlie can reclaim their coins from Bob through the Punish transactions.

Payment and Consolidation. Figure 5.3 depicts payment and consolidation using a Payment Tree. Assuming a fully constructed Payment Tree as shown in Figure 5.2d a payment is executed by giving the coins within the Merge transaction's Funding UTXO to Charlie instead of Alice. As shown in Figure 5.3a this changes the balance distribution represented by the transaction tree, reducing Alice's coins by 2 and adding those to Charlie's balance. A consolidation requires one atomic transaction tree update as shown in Figure 5.3b. This update spends the UTXOs within the Merge transaction's inputs and gives the coins to Bob and Charlie respectively. Note that this step does not change the balance distribution between the parties. Bob needs to make sure that this update is done atomically s.t. he avoids commitment of a Punish transaction. At this point the transaction trees are separate and in control of each channels' members respectively. Both pair of parties can now perform a last transaction tree update that replaces the respective transaction tree with a channel as shown in Figure 5.2a but that now represents the new balance distribution instead.

5.4.3 System Goals

In the following we define the desired properties of our protocol.

Theorem 5.4.1 (Balance Security)**.** *Outside of performing the intended payment, the sum of a honest party's coins is not reduced by participation in the Payment Tree protocol.*

Theorem 5.4.2 (Liveness)**.** *Eventually any honest party receives access to their coins through UTXOs spendable with a witness consisting of a signature corresponding to their verification key.*

5.5 Transactions

We use three types of transactions. Split transactions are used to split off coins from one channel, making them available to our construction in form of a Funding UTXO. Payout transactions take a Funding UTXO as input and pay the money to one of the two parties involved in it. Lastly, the Merge transaction is used to combine the Funding UTXOs that were split off two channels by taking them as input, paying out the intermediary's coins out as collateral and creating a Funding UTXO between the two remaining non-intermediary parties.

5.5.1 Split Transactions

Split transactions are of form $Tr_{\mathsf{split}} = (U_{\mathsf{in}}, U_{\mathsf{out}}, t)$ where $U_{\mathsf{in}} = \{\mathsf{ref}(f_\gamma)\}$ consist of one Funding UTXO provided by the channel-tree of γ, $U_{\mathsf{out}} = \{f_{\mathsf{change}}, f_{\mathsf{pay}}\}$ consists of two Funding UTXOs. It holds that $f_{\mathsf{change}}.b + f_{\mathsf{pay}}.b = f_\gamma$ and $f_{\mathsf{pay}}.b = b$. Moreover, $f_\gamma.\pi = f_{\mathsf{change}}.\pi = f_{\mathsf{pay}}.\pi$, i.e. all Funding UTXOs are shared between the same parties. The function call $\mathsf{SPLIT}(\gamma, b, t)$ creates a Split transaction as described above and returns f_{pay}. A function call to $\mathsf{UNSPLIT}(\gamma)$ consolidates the transaction into the channel by updating the channel's balance distribution with the split off balance. Additionally it sets up a channel between both parties by constructing a channel-tree with Funding UTXO f_{change} as root. *Split* transactions are used to take off b coins from each channel to be used for our construction. They are used to avoid that the existing channels are affected in case a corrupted intermediary misbehaves. Although we represent this by using a Split transactions as done with Virtual Channels and AMCU, it could be included similarly as conditional payments from HTLCs by placing a Funding UTXO instead of a HTLC contract.

5.5.2 Merge Transactions

Merge transactions are of form $Tr_{\mathsf{merge}} = (U_{\mathsf{in}}, U_{\mathsf{out}}, t)$ where $U_{\mathsf{in}} = \{f_{\mathsf{pay},0}, f_{\mathsf{pay},1}\}$ and $U_{\mathsf{out}} = \{f_{\mathsf{pay}}, u_{collateral}\}$. The two Funding UTXOs that are provided as input $f_{\mathsf{pay},0}$ and $f_{\mathsf{pay},1}$ are shared between parties $\mathcal{P}_A$ and $\mathcal{P}_B$ as well as between parties $\mathcal{P}_B$ and $\mathcal{P}_C$ respectively. The newly created Funding UTXOs f_{pay} in the output is shared between parties $\mathcal{P}_A$ and $\mathcal{P}_C$. The other UTXO within

the outputs is $u_{collateral}$ which pays out funds to $\mathcal{P}_B$. Lastly it holds that the coins in all UTXOs are equal, i.e. $f_{\mathsf{pay},0}.b = f_{\mathsf{pay},1}.b = f_{\mathsf{pay}}.b = u_{collateral}.b = b$. The function call $\mathsf{MERGE}(f_{\mathsf{pay},0}, f_{\mathsf{pay},1}, t)$ is a short-hand notation to construct a Merge transaction. We extend helper function $\mathsf{OUT\_UTXO}$ to accept a Merge transaction as input as well. In this case it returns UTXO f_{pay}. The helper function $\mathsf{IN\_UTXO}$ takes a Merge transaction as input and outputs the UTXOs that are used within its inputs, i.e. $f_{\mathsf{pay},0}, f_{\mathsf{pay},1}$. *Merge* transactions are used to combine transaction trees into one, essentially opening up a virtual channel between Alice and Charlie that can be used for a one-time payment.

5.5.3 Payout Transactions

Payout transactions are of form $Tr_{\mathsf{payout}} = (U_{\mathsf{in}}, U_{\mathsf{out}}, t)$ where $U_{\mathsf{in}} = \{f\}$ is a Funding UTXO and $U_{\mathsf{out}} = \{u_{\mathsf{payout}}\}$. It holds that u_{payout} pays out funds to a party $\mathcal{P}$ and $f.b = u_{\mathsf{payout}}.b$. The function call $\mathsf{PAYOUT}(f, \mathcal{P}, t)$ constructs a Payout transaction as described above. We extend helper function $\mathsf{IN\_UTXO}$ to take a Payout transaction as input in which case it outputs the UTXO f. Payout transactions are used at several points within our construction to serve different roles as shown in Figure 5.4. *Refund* transactions are used whenever Funding UTXOs are created. They are used to ensure that no funds are locked away within Funding UTXOs indefinitely even when any other party stops collaboration, which is essential to ensure the liveness property. *Punish* transactions are used to incentivise an intermediary to collaborate and ensure Merge transactions can be committed to the ledger. Without those, in case a Merge transaction is not committed to the ledger it could result in the loss of coins for Charlie in case the Refund transaction between Bob and Charlie is committed to the ledger instead and after the payment between Alice and Charlie has been performed. The *Payment* transaction is used to perform a change of the state, i.e. balance distribution, represented by the transaction tree, effectively performing a payment. Lastly, *Consolidation* transactions are used to deconstruct the transaction tree by applying the payment on both original transaction trees atomically. Without these, we cannot enforce the payment outside of committing the transaction tree to the ledger itself because of which the protocol would not fulfill the efficiency requirements for offchain protocols and thus not being classified as such. We note that the Refund and Punish transactions between Alice and Bob represent the same state s.t. the Punish transaction is redundant. However, for simplicity we opted to include both transactions making the construction symmetric. Whereas similarly the Consolidation and Punish transactions between Bob and Charlie do represent the same state in Figure 5.4, it is not possible to remove any of the transactions in the case where fees are paid to Bob which would be included within the Consolidation but not the Punish transactions.

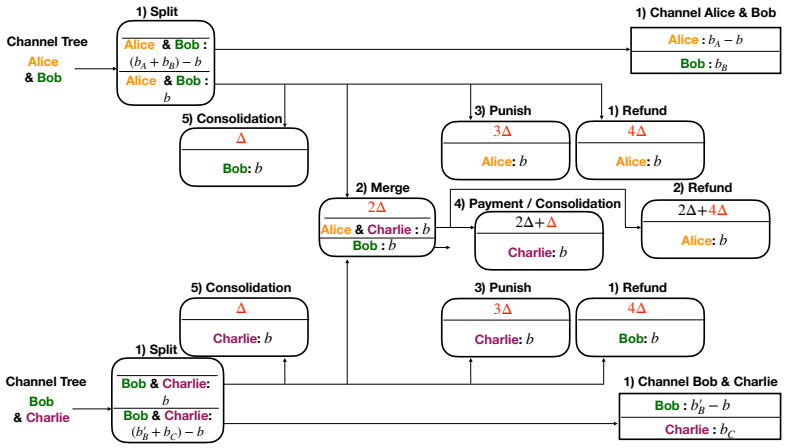


Figure 5.4: Transaction tree of a payment of b coins across 2 hops. Beforehand, the respective balances are b_A and b_B for Alice and Bob, b'_B and b_C for Bob in Charlie within their channels. Transactions are boxes with round corners containing the UTXOs they create, whereas referenced UTXOs in inputs are indicated implicitly by arrows originating from the UTXO that is spent. Red numbers indicate timelocks. Numbers atop the transaction indicate order of construction whereas transactions with same numbers are constructed atomically. Channel trees are boxes with straight edges forming a black box.

5.6 Our Payment Tree Construction

We describe the construction of a payment tree in respect to our running example. Let $\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_n, n \in \mathbb{N}$, be parties where parties $\mathcal{P}_{i-1}$ and $\mathcal{P}_i, i \in \{1, \ldots, n\}$ control channel γ_i. The protocol performs a payment of $b \in \mathbb{N}$ coins from $\mathcal{P}_0$ to $\mathcal{P}_n$. The value $\tau \in \mathbb{N}$ represents the current time, whereas $\Delta \in \mathbb{N}$ is the maximum time it takes for a transactions to be included in the ledger after committing it. We illustrate our approach in Figure 5.4 for a two-hop payment, i.e. for the case of $n = 2$. It is designed such that it can be extended to payment paths of arbitrary lengths. The construction is based on the overview given in Section 5.4. Numbers indicate the order in which transactions are created, whereas transactions with the same numbers are created atomically.

The Payment Tree Protocol. The protocol for constructing a Payment Tree across a path of n channels is depicted in Algorithm 7. It makes use of Algorithm 4 that allows an intermediary to atomically create two transactions, Algorithm 5 that performs a construction step of the Payment Tree, and Algorithm 6 that performs a consolidation step of the Payment Tree.

Algorithm 4 Atomically signing two Payout transaction

1: **function** ATOMIC_SIGN(Tr_0, Tr_1)
Require: Tr_0, Tr_1 are Payout transactions between three parties.
2: $f_0, f_1 \leftarrow \mathsf{FUTXO}(Tr_0), \mathsf{FUTXO}(Tr_1)$
3: $\mathcal{P}_I \leftarrow \mathsf{INTERMEDIARY}(f_0, f_1)$
4: $\mathcal{P}_A, \mathcal{P}_B \leftarrow \mathsf{COUNTERPARY}(f_0, \mathcal{P}_I), \mathsf{COUNTERPARY}(f_1, \mathcal{P}_I)$
5: $\mathsf{SIGN}(Tr_0, \{\mathcal{P}_A\}, \{\mathcal{P}_I\})$, $\mathsf{SIGN}(Tr_1, \{\mathcal{P}_B\}, \{\mathcal{P}_I\})$
6: $\mathsf{SIGN}(Tr_0, \{\mathcal{P}_I\}, \{\mathcal{P}_A\})$, $\mathsf{SIGN}(Tr_1, \{\mathcal{P}_I\}, \{\mathcal{P}_B\})$
7: **end function**

Figure 5.5: Algorithm that takes two Payout transactions as input and allows the intermediary party to enforce that either both or no transactions are fully signed.

5.6.1 Helper Functions and Sub-Protocols

Helper Functions. Function $\mathsf{SIGN}(Tr, P_S, P_R)$ is used to sign and exchange signatures of transactions. It takes a transaction Tr and two sets of parties P_S and P_R as input. Each party in P_S signs Tr and sends the signature to each party in P_R. This includes verification of signatures by the recipients. Function $\mathsf{PARTIES}$ takes a Funding UTXO as input and outputs a set containing the two parties of which a signature is required to spend the UTXO. Function $\mathsf{INTERMEDIARY}(f_0, f_1)$ takes two Funding UTXOs f_0, f_1 as input, if an intermediary exists, i.e. $|\mathsf{PARTIES}(f_0) \cap \mathsf{PARTIES}(f_1)| = 1$, then it returns the intermediary $\mathcal{P} \in \mathsf{PARTIES}(f_0) \cap \mathsf{PARTIES}(f_1)$. Otherwise it returns $\perp$. Function $\mathsf{COUNTERPARY}(f, \mathcal{P})$ takes a Funding UTXO and a party as input, if $\mathcal{P} \in \mathsf{PARTIES}(f)$, then it returns its counterparty $\mathcal{P}_C \in (\mathsf{PARTIES}(f)) \setminus \{\mathcal{P}\}$.

Atomic Signatures. We assume a setting with two channels between three parties. Protocol *ATOMIC_SIGN* is shown in Algorithm 4. It enables the intermediary party to enforce that two transactions – one on each channel – are created atomically. This is done by having the intermediary party provide signatures to both transactions only after they received all signatures from its counterparties.

Merging Channels. Protocol *MERGE* as shown in Algorithm 5 takes two Funding UTXOs f_0, f_1, an amount of coins b and a time t as input where f_0 is shared between parties $\mathcal{P}_A$ and $\mathcal{P}_I$, f_1 is shared between parties $\mathcal{P}_I$ and $\mathcal{P}_B$ and it holds that $f_0.b = f_1.b = b$. It creates a Merge transactions with timelock $t_m = t + 2\Delta$ spending both Funding UTXOs, paying out b coins to $\mathcal{P}_I$ and containing a Funding UTXO holding b coins, which are paid out to $\mathcal{P}_A$ after time $t_m + 4\Delta$ by means of a Payout transaction. This transaction tree is created atomically as its root, which is the Merge transaction, is signed last.

Algorithm 5 Construction Step of a Payment Tree

1: **function** MERGE(f_0, f_1, b, t)
2: $\mathcal{P}_I \leftarrow \mathsf{INTERMEDIARY}(f_0, f_1)$
3: $\mathcal{P}_A, \mathcal{P}_B \leftarrow \mathsf{COUNTERPARY}(f_0, \mathcal{P}_I), \mathsf{COUNTERPARY}(f_1, \mathcal{P}_I)$
4: $Tr_{mrg} \leftarrow \mathsf{MERGE}(f_0, f_1, t + 2\Delta)$
5: $Tr_{refund} \leftarrow \mathsf{PAYOUT}(\mathsf{OUT\_UTXO}(Tr_{mrg}), \mathcal{P}_A, t + 6\Delta)$
6: $Tr_{punish,A} \leftarrow \mathsf{PAYOUT}(f_0, \mathcal{P}_A, t + 3\Delta)$
7: $Tr_{punish,B} \leftarrow \mathsf{PAYOUT}(f_1, \mathcal{P}_C, t + 3\Delta)$
8: $\mathsf{SIGN}(Tr_{refund}, \{\mathcal{P}_A, \mathcal{P}_B\}, \{\mathcal{P}_A, \mathcal{P}_B\})$
9: $\mathsf{SIGN}(Tr_{mrg}, \{\mathcal{P}_A, \mathcal{P}_B, \mathcal{P}_I\}, \{\mathcal{P}_A, \mathcal{P}_B, \mathcal{P}_I\})$
10: ATOMIC_SIGN($Tr_{punish,A}, Tr_{punish,B}$) **return** Tr_{mrg}
11: **end function**

Figure 5.6: Creation of a Funding UTXO between two counterparties. The intermediary can enforce atomic construction while Punish transactions provide incentive.

Only after each party holds a fully signed instance of the Merge transaction, two Punish transactions spending f_0 and f_1 and paying out b coins to $\mathcal{P}_A$ and $\mathcal{P}_B$ respectively are created atomically using *ATOMIC_SIGN*. These have timelocks equal to $t + 3\Delta$. Note that the creation of the Merge transaction must not redistribute funds, i.e. the funds in f_0 are paid by $\mathcal{P}_A$ and the funds in f_1 are paid by $\mathcal{P}_I$. The Punish transactions are used to secure the funds within the Merge transaction by paying out funds to $\mathcal{P}_A$ and $\mathcal{P}_B$, if the Merge transaction cannot be committed to the ledger. Timelocks are selected to perform transformations on the existing transaction through the invalidation by timelock technique and also to allow the construction to be performed iteratively. Timelock t_m is selected s.t. a Consolidation transaction can be placed with timelock $t+\Delta$ during the protocol's consolidation phase. Timelocks of the Punish transactions are selected s.t. they are invalidated by the Merge transaction *conditionally*, i.e. only if the Merge transaction can be committed to the ledger, the Punish transactions are invalid. The Payout transaction acts as a Refund transaction for the new Merge transaction. Respectively we assign it a timelock of $t_m + 4\Delta$ such that Consolidation, Merge and Punish transactions can be placed with timelocks $t_m + \Delta$, $t_m + 2\Delta$ and $t_m + 3\Delta$ respectively. Note that if the Merge transaction is on top of the Payment Tree s.t. it is not used for further channel merges, the Refund transaction's timelock can be reduced to $t_m + 2\Delta$. Lastly, if a transaction spends another transaction, its timelock needs to be larger by at least Δ to ensure that all transactions can be committed to the ledger as soon as their timelocks expire.

Consolidation. Algorithm 6 takes a Merge transaction as input, invalidates it by creating two Payout transactions atomically using the ATOMIC_SIGN

Algorithm 6 Deconstructing Step of a Payment Tree

1: **function** CONSOLIDATE(Tr_{mrg})
2: $f_0, f_1 \leftarrow \mathsf{IN\_UTXO}(Tr_{mrg})$
3: $\mathcal{P}_I \leftarrow \mathsf{INTERMEDIARY}(f_0, f_1)$
4: $\mathcal{P}_A, \mathcal{P}_B \leftarrow \mathsf{COUNTERPARY}(f_0, \mathcal{P}_I), \mathsf{COUNTERPARY}(f_1, \mathcal{P}_I)$
5: $Tr_A \leftarrow \mathsf{PAYOUT}(f_0, \mathcal{P}_B, t + \Delta)$
6: $Tr_B \leftarrow \mathsf{PAYOUT}(f_1, \mathcal{P}_C, t + \Delta)$
7: ATOMIC_SIGN(Tr_A, Tr_B)
8: **end function**

Figure 5.7: Invalidating a Merge transactions and atomically updating the state on the two original Funding UTXOs.

protocol that spend the Merge transaction's inputs. Both consolidation transactions perform a payment by giving the funds to the payee. Note that the protocol can be adjusted to cancel a payment by refunding the funds to the payer instead.

5.6.2 The Complete Payment Trees Protocol

Algorithm 7 performs a payment from $\mathcal{P}_0$ to $\mathcal{P}_n$ by iteratively merging Funding UTXOs, s.t. the Merge transactions form the nodes of a balanced binary tree as illustrated in Figure 5.9. The algorithm takes the following inputs: (1) The payment path $\gamma_1, \ldots, \gamma_n$, (2) the payment amount b, and (3) time t_{min}. The value t_{min} is negotiated by the parties and represents the maximum amount of time the parties have to execute the protocol. The dispute protocol starts if the protocol is not concluded until t_{min}. Note that even existing methods as HTLCs have to account for t_{min}.

In the following we refer to a certain depth within this binary tree as *level*, beginning with Split transactions on level 0. The algorithm maintains lists of Funding UTXOs F_UTXO_i for each level $i \geq 0$ of the binary tree, as well as lists of Merge transactions MRG_j for each level $j \geq 1$ of the binary tree. The algorithm proceeds as follows. Add a Funding UTXO from each Split transaction to F_UTXO_0 in order (4 - 7) and create the Payment Tree by iterative use of the *MERGE* protocol level-by-level (8 - 18). The Merge transactions and Funding UTXOs created on level j are added to lists MRG_j and F_UTXO_j respectively and in order (12 - 13). Note that if there is an uneven amount of Funding UTXOs within a level, we leave the odd one to be used in the level above instead (15 - 17). The payment is executed after construction is concluded (19). Afterwards the payment tree is deconstructed in reverse order by executing the *CONSOLIDATE* protocol on each Merge transaction (20 - 24). Lastly the Split transactions are removed and consolidation within all original channels concludes (25 - 27).

Algorithm 7 Payment Tree Construction

function PAYMENTTREE($\gamma_1, \gamma_2, \ldots, \gamma_n$, b, t_{min})
  $F\_UTXO_i \leftarrow [], 0 \leq i \leq \lceil (\log n) - 1 \rceil$
  $MRG_i \leftarrow [], 1 \leq i \leq \lceil \log n \rceil$
  for $1 \leq i \leq n$ **do**
    $f_i \leftarrow \mathsf{SPLIT}(\gamma_i, b, t_{min})$
    Append f_i to F_UTXO_0
  end for
  for $i = 0$ until $i = \lceil (\log n - 1) \rceil$ **do**
    for $0 \leq j \leq \lfloor |F\_UTXO_i|/2 \rfloor$ **do**
      Retrieve f_{2j}, f_{2j+1} from F_UTXO_i
      $Tr_{mrg,j} \leftarrow \text{MERGE}(f_{2j}, f_{2j+1}, b, t_{min} + 2i\Delta)$
      Append $\mathsf{OUT\_UTXO}(Tr_{mrg,j})$ to F_UTXO_{i+1}
      Append $Tr_{mrg,j}$ to MRG_{i+1}
    end for
    if $|F\_UTXO_i| \mod 2 = 1$ **then**
      Remove last entry of F_UTXO_i and append to F_UTXO_{i+1}
    end if
  end for
  $Tr_{Payment} \leftarrow \mathsf{PAYOUT}(\mathsf{OUT\_UTXO}(MRG_{\lceil \log n \rceil}[0]), \mathcal{P}_n, t_{min} + 2\Delta \log n + \Delta)$
  for $i = \lceil \log n \rceil$ until $i = 1$ **do**
    for Tr_{mrg} in MRG_i **do**
      CONSOLIDATE(Tr_{mrg})
    end for
  end for
  for $1 \leq i \leq n$ **do**
    $\mathsf{UNSPLIT}(\gamma_i)$
  end for
end function

Figure 5.8: The full Payment Tree protocol from construction to consolidation.

Dispute. This protocol is executed at time t_{min} if the payment tree protocol has not come to conclusion in an orderly manner. Every honest party submits their transactions to the ledger as soon as their respective timelocks expire. This will result in commitment of the payment tree onto the ledger where transactions are committed in order of their priority. If a Merge transaction cannot be committed to the ledger, refunds and payments are done via Punish transactions.

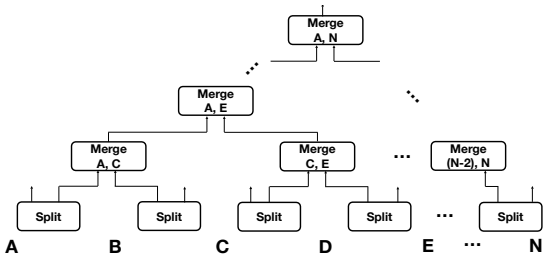


Figure 5.9: Payment tree in the shape of a balanced binary tree.

5.6.3 Handling of Fees

Fees can be paid by the payer $\mathcal{P}_0$ and payee $\mathcal{P}_n$ or either of them alone to the intermediaries to compensate for their invested collateral. Our approach to handling fees is similar to the approach used for HTLCs, however, adapted to the binary tree structure of Payment Trees. Any party acting as intermediary when creating a Merge transaction receives cumulative fees from the other two parties participating in the Merge transaction's construction. The cumulative fee paid to the intermediary is composed of two parts. For one, it contains the fees paid to the intermediary themselves, and for another, it contains coins the party has to forward to the parties who act as intermediaries of Merge transactions on the lower levels of the Payment Tree. For simplicity, in the following we assume that the path's length is a power of 2, i.e. $n = 2^i, i \in \mathbb{N}$, the paid fees f are equal for each intermediary and all fees are shared between payer $\mathcal{P}_0$ and payee $\mathcal{P}_n$ equally. Then, a party that acts as intermediary of level i of the Payment Tree receives $f_i = f + 2\frac{f_{i-1}}{2} = f + f_{i-1}$ coins, where $f_1 = f$. The fee f_i is paid equally by the other two parties involved in the Merge transaction's construction. Payment of fees happens within Merge transactions by adding a fee to the collateral the intermediary receives. However, this raises the challenge that we have to ensure that all transactions receive sufficient funding: The coins within a transaction's inputs have to cover all coins within their outputs. Moreover, to ensure that the consolidation step can be performed, the collateral of the intermediary within a Merge transaction has to be at least as high as the coins within the Merge transaction's Funding UTXO [66]. Therefore, when performing the merge step on level i every party has to have an additional balance of $f_{i,cum} = \sum_{j=i}^{h} f_j$, whereas the collateral of a Merge transaction's intermediary equals $b + f_{i,cum} + f_i$ where f_i is paid equally from balances brought by the other two parties.

5.7 Collateral Efficiency and Security Analysis

In this section we discuss properties of the Payment Tree construction.

5.7.1 Efficiency Analysis

Figure 5.10 depicts the efficiency properties of Payment Trees, comparing it to existing approaches. We compare two metrics: (1) The collateral, and (2) the number of transactions that have to be committed to the ledger in case of dispute. We do this for individual parties, as well as for the whole payment.

Commitment of each Merge transaction unlocks the collateral of one party. To commit a Merge transaction located on level i of the payment tree it needs to commit i transactions beforehand, i.e. $i-1$ Merge transaction as well as a Split transaction. This will happen at time $2\Delta i$. As the height of the Payment tree is limited by $\lceil \log n \rceil$ it follows that any party invests $b2\Delta i \in \mathcal{O}(b\Delta \log n)$ collateral and has to commit $i+1 \in \mathcal{O}(\log n)$ transactions. Regarding the total payment, we observe that there are $\frac{n}{2^i}$ Merge transactions on level i of the payment tree. It follows that the total collateral equals the sum $\sum_{i=1}^{\lceil \log n \rceil} b2\Delta i \frac{n}{2^i} = b2\Delta n \sum_{i=1}^{\lceil \log n \rceil} \frac{i}{2^i}$. As $\sum_{i=1}^{\infty} \frac{i}{2^i} = 2$ and each part of the sum is positive, it follows that the total collateral $b2\Delta n \sum_{i=1}^{\lceil \log n \rceil} \frac{i}{2^i} < 4b\Delta n \in \mathcal{O}(b\Delta n)$ is linear in the length of the payment path n. The number of transactions can be computed in a similar fashion, however, an intuitive approach is to recall that the transactions form a balanced binary tree of height $1 + \lceil \log n \rceil$ which has at most $2^{1+\lceil \log n \rceil} \leq 2n \in \mathcal{O}(n)$ nodes. Although the collateral any individual party has to invest is logarithmic, therefore higher than Sprites but lower than HTLCs, the total collateral incurred over the whole payment is linear in the path's length. This is comparable to the performance of Sprites and is by a factor of n lower than the total collateral of HTLCs. A trade-off of Payment Trees is that an individual party might have to commit up to $\mathcal{O}(\log n)$ many transactions. Nevertheless the total number of transactions over the whole payment is comparable to both, HTLCs and Sprites. Payment Trees provide a performance comparable to Sprites without requiring a ledger with smart contract capability.

5.7.2 Anonymity

There are two metrics for anonymity for offchain payments. (1) Sender and Receiver privacy denotes whether sender and receiver of a payment are unknown for any intermediary, (2) path privacy denotes whether intermediaries are anonymous to other intermediaries within a payment path. We compare these privacy notions of existing approaches in Figure 5.11. While HTLCs in themselves do not provide privacy guarantees, further work [87] introduced adjustments to provide privacy notions. Similarly, smart contracts do not provide privacy guarantees and Sprites does explicitly not address any privacy notions, however, privacy might be addressed in future work.

The Payment Trees protocol leaks no information on intermediaries within a path if all intermediaries use pseudonyms when creating the transaction and do not re-use their identities within the network. However, if the intermediary at the topmost node of the Payment Tree is forcing a dispute that leads to the commitment of all merge transactions to the ledger, all intermediaries, i.e.

Method	pp Collateral	pp Tr.	Total Collateral	Total Tr.
HTLC [112, 25]	$\mathcal{O}(b\Delta n)$	$\mathcal{O}(1)$	$\mathcal{O}(b\Delta n^2)$	$\mathcal{O}(n)$
Sprites [98]	$\mathcal{O}(b(n+\Delta))$	$\mathcal{O}(1)$	$\mathcal{O}(b(n+\Delta)n)$	$\mathcal{O}(n)$
Payment Tree	$\mathcal{O}(b\Delta \log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(b\Delta n)$	$\mathcal{O}(n)$

Figure 5.10: Comparison of the performance of Payment Trees across the whole payment (Total) and individually per party (pp).

Method	Sender / Receiver Pr.	Path Pr.	Smart Contracts
HTLC [112, 25]	Yes	$\mathcal{O}(1)$	No
Sprites [98]	No	$\mathcal{O}(n)$	Yes
Payment Tree	Yes	$\mathcal{O}(1)$ / $\mathcal{O}(n)$	No

Figure 5.11: Comparison of the performance of Payment Trees regarding anonymity and ledger requirements.

$\mathcal{O}(n)$, are leaked. However, if any part of the tree is committed to the ledger no information is leaked on whether it is the topmost node of the tree s.t. Sender and Receiver of the payment remain private. Thus Payment Trees has Sender and Receiver privacy.

Nevertheless, we denote that if an adversary attempts to learn the balances of the channels within a channel network they can potentially observe any payments done within the network, breaking all privacy independent of the protocol used.

5.7.3 Discussion of Attacks from the Literature

Denial of Service Attacks. The Payment Tree protocol mitigates existing attacks such as the congestion and lockdown attacks[99, 109] on HTLCs that aim to lock a channel's coins within unfulfilled HTLCs. This is done by reducing the total and individual collateral of payments. While a large scale DoS attack on multiple channels is difficult as the total collateral of Payment Trees is linear in the payment path's length, a specific intermediary can be targeted to act a intermediary on the highest level of the Payment Tree to pay a logarithmic collateral. Another aspect of the Lockdown attack is that a channel is blocked by saturating the number of HTLCs applicable to a channel which is limited by the maximum size of a transaction. The Payment Trees protocol mitigates this by using Split transactions. Each pending payment requires the construction of a Split transaction. This prevents that there is any transaction that increases in size depending on the number of pending transactions. However, a tradeoff to using Payment Trees is the increased number of transactions that would need to be committed to the ledger in case of a dispute.

Wormhole Attacks. The Payment Tree protocol pays coins to intermediaries of a Merge transaction and they include fees for all intermediaries on lower levels of respective sub-trees. An attack similar to the wormhole attack can be performed by a corrupted intermediary when creating a Merge transaction by replacing the Merge transaction's inputs with UTXOs they control. Doing this they could take all fees that were intended to be forwarded to other parties while preventing them to participate in the protocol. In contrast to the wormhole attack on HTLCs a wormhole-like attack on the Payment Trees protocol requires making changes to the transaction tree which in-turn can be detected and prevented. We assume that either $\mathcal{P}_0$ and $\mathcal{P}_n$ are honest. Otherwise, if both are corrupted the attack would only redistribute coins between corrupted parties resulting in no net gain to the adversary. During creation of the Payment Tree all intermediaries send their view of the protocol to $\mathcal{P}_0$ and $\mathcal{P}_n$, i.e. the Merge transactions they are involved in. Having this information $\mathcal{P}_0$ and $\mathcal{P}_n$ can verify correctness of the construction and abort the payment in the negative case.

5.7.4 Security Proofs

In the following we prove the protocol's security properties defined in Section 5.4.3.

Theorem 5.4.1 (Balance Security). *Outside of performing the intended payment, the sum of a honest party's coins is not reduced by participation in the Payment Tree protocol.*

Sketch of Proof. First, we consider the case in which the adversary does not deviate from the protocol, but stops collaboration mid-way. We observe that due to the order in which transactions are (atomically) created, the funds accessible for any party within Merge- and Payment transaction is unchanged, except when executing the payment between $\mathcal{P}_0$ and $\mathcal{P}_1$ explicitly. Any party receives their Funds by having the transaction tree be committed to the ledger. Even if a corrupted party acts as intermediary and stops collaboration after receiving signatures and before providing signatures themselves. As only they risk losing funds due to Punish transactions, having them selectively commit and withhold transactions does not result in the loss of funds of their counterparties.

Next, we consider the case where the adversary corrupts two parties to double-spend a Funding UTXO that is the input of a Merge transaction. Assume $\mathcal{P} \in \{\mathcal{P}_1, \ldots, \mathcal{P}_{n-1}\}$ is neither payer or payee of the overall payment and is honest. Moreover, the adversary double spends a Funding UTXO that is the input of a Merge transaction $Tr_{mrg,\mathcal{A}}$ on level i. Note that for this to happen, the party that acts as intermediary of $Tr_{mrg,\mathcal{A}}$ must be corrupted as either Funding UTXO that is input of $Tr_{mrg,\mathcal{A}}$ requires its signature to spend it. If $\mathcal{P}$ is not part of a transaction that is descendant of $Tr_{mrg,\mathcal{A}}$ they are unaffected and do not lose funds. Otherwise, if they are part of $Tr_{mrg,\mathcal{A}}$ they are not the intermediary party as they are honest and they will receive b coins through a Punish transaction. If they are not part of $Tr_{mrg,\mathcal{A}}$, let $j, i < j \geq \log n$ be the

lowest level on which they are part of a Merge transaction that has $Tr_{mrg,\mathcal{A}}$ as descendant. Then they must not be the intermediary, as otherwise, they would have a descendant of $Tr_{mrg,\mathcal{A}}$ on a lower level. Therefore they are not intermediary and receive b coins through a Punish transaction on that level. However, as they are neither $\mathcal{P}_0$ nor $\mathcal{P}_n$ they act as intermediary of a Merge transaction on level $k, j < k < \log n$ which is descendant of $Tr_{mrg,\mathcal{A}}$. On that level $\mathcal{P}$ has to pay out b coins through one Punish transaction. Note that $\mathcal{P}$ does not pay out b coins through two Punish transactions as otherwise they could commit the Merge transaction instead to avoid payout of any Punish transaction. Moreover, any party is intermediary of a Merge transaction only once within the transaction. Overall, $\mathcal{P}$'s balance equals $b - b = 0$ s.t. they do not lose funds. The reasoning for $\mathcal{P}_0$ and $\mathcal{P}_n$ is analogous. As they are on the top level of the Payment Tree, they have a Merge transaction that is descendant of $Tr_{mrg,\mathcal{A}}$ and therefore do receive b coins. However, they are never intermediary of a Merge transaction, s.t. their balance is b coins. Therefore, $\mathcal{P}_0$ and $\mathcal{P}_n$ do not lose coins independently of whether they performed the payment between each other or not. □

Theorem 5.4.2 (Liveness). *Eventually any honest party receives access to their coins through UTXO spendable with a witness consisting of a signature corresponding to their verification key.*

Sketch of Proof. All honest parties commit the transactions they are involved in as soon as their timelocks expire. First, we note that any transaction containing a Funding UTXO is created atomically with a Payout transaction that pays out the funds to a party that receives exclusive access to it. Therefore, the adversary cannot have funds being locked within a Funding UTXO they share control of indefinitely. Although all transactions have increasingly higher timelocks, all transactions can be committed to the ledger by time $t_{min}+2\Delta \log n+4\Delta$. By this time, no funds are locked within a Funding UTXO and any funds can be claimed by one party exclusively through Payout transactions. As *Balance Security* holds, no party loses funds when all Payout transactions are committed to the ledger s.t. for any party it holds that by time $t_{min} + 2\Delta \log n + 4\Delta$ they have exclusive access to all their funds. □

5.8 Conclusion

Payment Trees provide competitive performance to state-of-the-art approaches as Sprites, while having fewer restrictions to its employability by not requiring smart contract capability. Thus providing the first secure alternative to HTLCs for the Lightning Network.

Chapter 6

Conclusion

In the following we conclude our work in the context of the related work in Section 6.1, and highlight possible directions for future work in Section 6.2.

6.1 Conclusion

In this work we present a framework for Offchain Protocols for UTxO based ledger. We used this protocol to construct two protocols. (1) We constructed a protocol to create Lightweight Virtual Payment Channels and (2) we constructed a protocol for low collateral payments in PCNs. Both protocols are the first secure solutions for virtual channel construction and low collateral payments that do not rely on Smart Contracts. Thus we show the feasibility of our framework to deduct protocols that otherwise would only be possible to be implemented on ledgers with smart contract capability. Moreover we presented the first secure alternative to HTLCs for UTxO Ledger based PCNs.

However, this comes with tradeoffs. The largest trade-off is the size of the transaction trees, i.e. the number of transactions they contain, that our constructions require, in contrast to solutions based on smart contracts. We note that the total amount of transactions is linear with the payment path for Payment Trees and similarly linear for the number of hops a lightweight virtual channel spans across the underlying PCN, is both comparable to the related work. However, practically the number of transactions that would need to be committed to the ledger in case of a dispute is larger than in the related work by a small constant. Moreover, while the virtual channel construction by Dziembowski et. al [46, 49] can be extended to use state channels our construction is limited to payment channels alone. However, we note that, to our knowledge, there exists no construction for state channels that does not require smart contracts. Overall our work succeeded in reducing the gap between solutions for UTxO based ledger and solutions relying on smart contracts, however, we do not completely bridge the difference.

6.2 Potential Future Work

There are several future directions of our work. (1) We argue that our framework can be used to re-visit problem statements in the related work and used to present alternative constructions with different trade-offs. (2) The framework could be used to further present alternatives to protocols previously only available for smart contract based ledgers. (3) While the Payment Trees protocol presents an approach with total collateral similar to Sprites, the individual collateral of any party is up to logarithmic. We argue that it constitutes an interesting research question to verify whether this collateral is a limit for PCNs on UTxO based ledgers. (4) While EUTxOs [32] are strict extensions of the UTxO model such that our approach can be used directly with EUTxO without any changes, the higher expressiveness of the EUTxO model might improve the performance of our protocols. Adapting our protocols to the EUTxO model might present the first solution that allows inter-head communication for Hydra [33].

Chapter 7

Bibliography

[1] Bip 68. https://github.com/bitcoin/bips/blob/master/bip-0068.mediawiki. Accessed: 2018-11-28.

[2] c-lightning a lightning network implementation in c. https://github.com/ElementsProject/lightning. Accessed: 2019-03-26.

[3] Comit whitepaper. https://www.comit.network/doc/COMIT/white/paper/v1.0.2.pdf. Accessed: 2019-03-27.

[4] Cosmo. cosmos.network. Accessed: 2019-03-27.

[5] Lightning in scriptless script. Mailing list port: https://github.com/lightningnetwork/lnd. Accessed: 2019-03-26.

[6] Ligthing network daemon. https://github.com/lightningnetwork/lnd. Accessed: 2019-03-26.

[7] Raiden network. raiden.network. Accessed: 2018-09-03.

[8] Ripple. ripple.com. Accessed: 2018-10-17.

[9] A scala implementation of the lightning network. https://github.com/ACINQ/eclair. Accessed: 2019-03-26.

[10] Scriptless script. Presentation slides: https://download.wpsoftware.net/bitcoin/wizardry/mw-slides/2017-05-milan-meetup/slides.pdf. Accessed: 2019-03-26.

[11] Speedymurmurs. https://crysp.uwaterloo.ca/software/speedymurmurs. Accessed: 2018-10-17.

[12] tpec: 2p-ecdsa signatures. Github repository, https://github.com/cfromknecht/tpec. Accessed: 2019-03-27.

[13] Rapidly-adjusted (micro)payments to a pre-determined party. https://en.bitcoin.it/wiki/Contract, 2011. Accessed: 2018-09-03.

[14] Zcash. https://z.cash, 2016. Accessed: 2018-09-26.

[15] Syed Taha Ali, Dylan Clarke, and Patrick McCorry. The nuts and bolts of micropayments: A survey. *CoRR*, abs/1710.02964, 2017.

[16] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Fair two-party computations via bitcoin deposits. In Rainer Böhme, Michael Brenner, Tyler Moore, and Matthew Smith, editors, *FC 2014 Workshops*, volume 8438 of *LNCS*, pages 105–121. Springer, Heidelberg, March 2014.

[17] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Secure multiparty computations on bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 443–458. IEEE Computer Society Press, May 2014.

[18] Georgia Avarikioti, Felix Laufenberg, Jakub Sliwinski, Yuyi Wang, and Roger Wattenhofer. Incentivizing payment channel watchtowers. In *Scaling Bitcoin*, 2018.

[19] Georgia Avarikioti, Felix Laufenberg, Jakub Sliwinski, Yuyi Wang, and Roger Wattenhofer. Towards secure and efficient payment channels. *arXiv preprint arXiv:1811.12740*, 2018.

[20] Iddo Bentov and Ranjit Kumaresan. How to use bitcoin to design fair protocols. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 421–439. Springer, Heidelberg, August 2014.

[21] Iddo Bentov, Ranjit Kumaresan, and Andrew Miller. Instantaneous decentralized poker. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part II*, volume 10625 of *LNCS*, pages 410–440. Springer, Heidelberg, December 2017.

[22] Iddo Bentov, Rafael Pass, and Elaine Shi. Snow white: Provably secure proofs of stake. Cryptology ePrint Archive, Report 2016/919, 2016. `http://eprint.iacr.org/2016/919`.

[23] Bitcoin Bip. Bip-0065. `https://github.com/bitcoin/bips/blob/master/bip-0065.mediawiki`, 2018. [Online; accessed 30-October-2018].

[24] Dan Boneh, Manu Drijvers, and Gregory Neven. Compact multi-signatures for smaller blockchains. Cryptology ePrint Archive, Report 2018/483, 2018. `https://eprint.iacr.org/2018/483`.

[25] Sean Bowe and Daira Hopwood. Hashed Time-Locked Contract transactions. https://github.com/bitcoin/bips/blob/master/bip-0199.mediawiki, 2017. [Online; accessed 29-August-2020].

[26] Simina Brânzei, Erel Segal-Halevi, and Aviv Zohar. How to charge lightning. *arXiv preprint arXiv:1712.10222*, 2017.

[27] Chris Buckland and Patrick McCorry. Two-party state channels with assertions. Third Workshop on Trusted Smart Contracts, 2019.

[28] Conrad Burchert, Christian Decker, and Roger Wattenhofer. Scalable funding of bitcoin micropayment channel networks-regular submission. In *SSS*, pages 361–377, 2017.

[29] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

[30] Ran Canetti. Universally composable signature, certification, and authentication. In *Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004.*, pages 219–233. IEEE, 2004.

[31] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In Salil P. Vadhan, editor, *TCC 2007*, volume 4392 of *LNCS*, pages 61–85. Springer, Heidelberg, February 2007.

[32] Manuel MT Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Michael Peyton Jones, and Philip Wadler. The extended utxo model. In *4th Workshop on Trusted Smart Contracts*, 2020.

[33] Manuel MT Chakravarty, Sandro Coretti, Matthias Fitzi, Peter Gazi, Philipp Kant, Aggelos Kiayias, and Alexander Russell. Hydra: Fast isomorphic state channels.

[34] David Chaum. Blind signature system. In David Chaum, editor, *CRYPTO'83*, page 153. Plenum Press, New York, USA, 1983.

[35] David Chaum, Amos Fiat, and Moni Naor. Untraceable electronic cash. In Shafi Goldwasser, editor, *CRYPTO'88*, volume 403 of *LNCS*, pages 319–327. Springer, Heidelberg, August 1990.

[36] Jing Chen, Sergey Gorbunov, Silvio Micali, and Georgios Vlachos. ALGORAND AGREEMENT: Super fast and partition resilient byzantine agreement. Cryptology ePrint Archive, Report 2018/377, 2018. `https://eprint.iacr.org/2018/377`.

[37] Alessandro Chiesa, Matthew Green, Jingcheng Liu, Peihan Miao, Ian Miers, and Pratyush Mishra. Decentralized anonymous micropayments. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part II*, volume 10211 of *LNCS*, pages 609–642. Springer, Heidelberg, April / May 2017.

[38] Jeff Coleman, Liam Horne, and Li Xuanji. Counterfactual: Generalized state channels, 2018.

[39] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, et al. On scaling decentralized blockchains. In *International Conference on Financial Cryptography and Data Security*, pages 106–125. Springer, 2016.

[40] Pranav Dandekar, Ashish Goel, Ramesh Govindan, and Ian Post. Liquidity in credit networks: A little trust goes a long way. In *Proceedings of the 12th ACM Conference on Electronic Commerce*, EC '11, pages 147–156, New York, NY, USA, 2011. ACM.

[41] Bernardo David, Rafael Dowsley, and Mario Larangeira. Kaleidoscope: An efficient poker protocol with payment distribution and penalty enforcement. Cryptology ePrint Archive, Report 2017/899, 2017. `http://eprint.iacr.org/2017/899`.

[42] Bernardo David, Rafael Dowsley, and Mario Larangeira. 21 - bringing down the complexity: Fast composable protocols for card games without secret state. In Willy Susilo and Guomin Yang, editors, *ACISP 18*, volume 10946 of *LNCS*, pages 45–63. Springer, Heidelberg, July 2018.

[43] Bernardo David, Rafael Dowsley, and Mario Larangeira. ROYALE: A framework for universally composable card games with financial rewards and penalties enforcement. Cryptology ePrint Archive, Report 2018/157, 2018. `https://eprint.iacr.org/2018/157`.

[44] Christian Decker and Roger Wattenhofer. A fast and scalable payment network with bitcoin duplex micropayment channels. In *Symposium on Self-Stabilizing Systems*, pages 3–18. Springer, 2015.

[45] Alexandra Dmitrienko, David Noack, and Moti Yung. Secure wallet-assisted offline bitcoin payments with double-spender revocation. In Ramesh Karri, Ozgur Sinanoglu, Ahmad-Reza Sadeghi, and Xun Yi, editors, *ASIACCS 17*, pages 520–531. ACM Press, April 2017.

[46] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, and Daniel Malinowski. PERUN: Virtual payment channels over cryptographic currencies. Cryptology ePrint Archive, Report 2017/635, 2017. `http://eprint.iacr.org/2017/635`.

[47] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, and Daniel Malinowski. Perun: Virtual payment hubs over cryptocurrencies. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 106–123. IEEE, 2019.

[48] Stefan Dziembowski, Sebastian Faust, and Kristina Hostakova. Foundations of state channel networks. Cryptology ePrint Archive, Report 2018/320, 2018. `https://eprint.iacr.org/2018/320`.

[49] Stefan Dziembowski, Sebastian Faust, and Kristina Hostáková. General state channel networks. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 949–966. ACM Press, October 2018.

[50] Stefan Dziembowski, Sebastian Faust, and Kristina Hostáková. General state channel networks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 949–966. ACM, 2018.

[51] Christoph Egger, Pedro Moreno-Sanchez, and Matteo Maffei. Atomic multi-channel updates with constant collateral in bitcoin-compatible payment-channel networks. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 801–815. ACM Press, November 2019.

[52] EOS. EOS. `https://eos.io/`, 2018. [Online; accessed 6-May-2018].

[53] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert Van Renesse. Bitcoin-ng: A scalable blockchain protocol. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, pages 45–59, Berkeley, CA, USA, 2016. USENIX Association.

[54] Simon N. Foley. Using trust management to support transferable hash-based micropayments. In Rebecca Wright, editor, *FC 2003*, volume 2742 of *LNCS*, pages 1–14. Springer, Heidelberg, January 2003.

[55] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399404, 1956.

[56] Cardano Foundation. Cardano Hub. `https://www.cardano.org/`, 2018. [Online; accessed 28-March-2018].

[57] Juan Garay and Aggelos Kiayias. Sok: A consensus taxonomy in the blockchain era. Cryptology ePrint Archive, Report 2018/754, 2018. `https://eprint.iacr.org/2018/754`.

[58] Arpita Ghosh, Mohammad Mahdian, Daniel M. Reeves, David M. Pennock, and Ryan Fugger. Mechanism design on trust networks. In Xiaotie Deng and Fan Chung Graham, editors, *Internet and Network Economics*, pages 257–268, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[59] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. Cryptology ePrint Archive, Report 2017/454, 2017. `http://eprint.iacr.org/2017/454`.

[60] Roberto Canonico Giovanni Di Stasi, Stefano Avallone and Giorgio Ventre. Routing payments on the lightning network. http://wpage.unina.it/giovanni.distasi/pub/blockchain2018-main.pdf, 2018. Accessed: 2019-03-28.

[61] Matthew Green and Ian Miers. Bolt: Anonymous payment channels for decentralized currencies. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 473–489. ACM Press, October / November 2017.

[62] Lewis Gudgeon, Pedro Moreno-Sanchez, Stefanie Roos, Patrick McCorry, and Arthur Gervais. Sok: Off the chain transactions. Cryptology ePrint Archive, Report 2019/360, 2019. `https://eprint.iacr.org/2019/360`.

[63] Ethan Heilman, Leen Alshenibr, Foteini Baldimtsi, Alessandra Scafuro, and Sharon Goldberg. Tumblebit: An untrusted bitcoin-compatible anonymous payment hub. In *Network and Distributed System Security Symposium*, 2017.

[64] Philipp Hoenisch and Ingo Weber. Aodv–based routing for payment channel networks. In Shiping Chen, Harry Wang, and Liang-Jie Zhang, editors, *Blockchain – ICBC 2018*, pages 107–124, Cham, 2018. Springer International Publishing.

[65] Kexin Hu and Zhenfeng Zhang. Fast lottery-based micropayments for decentralized currencies. In Willy Susilo and Guomin Yang, editors, *ACISP 18*, volume 10946 of *LNCS*, pages 669–686. Springer, Heidelberg, July 2018.

[66] Maxim Jourenko, Mario Larangeira, and Keisuke Tanaka. Lightweight virtual payment channels. Cryptology ePrint Archive, Report 2020/998, 2020. `https://eprint.iacr.org/2020/998`.

[67] Dean Karlan, Markus Mobius, Tanya Rosenblat, and Adam Szeidl. Trust and social collateral. *The Quarterly Journal of Economics*, 124(3):1307–1361, 2009.

[68] Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. Universally composable synchronous computation. In *Theory of Cryptography Conference*, pages 477–498. Springer, 2013.

[69] Rami Khalil and Arthur Gervais. Revive: Rebalancing off-blockchain payment networks. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 439–453. ACM, 2017.

[70] Rami Khalil and Arthur Gervais. NOCUST - A non-custodial 2nd-layer financial intermediary. Cryptology ePrint Archive, Report 2018/642, 2018. `https://eprint.iacr.org/2018/642`.

[71] Aggelos Kiayias and Orfeas Stefanos Thyfronitis Litos. A composable security treatment of the lightning network. Cryptology ePrint Archive, Report 2019/778, 2019. `https://eprint.iacr.org/2019/778`.

[72] Aggelos Kiayias and Orfeas Stefanos Thyfronitis Litos. A composable security treatment of the lightning network. *IACR Cryptology ePrint Archive*, 2019:778, 2019.

[73] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 357–388. Springer, Heidelberg, August 2017.

[74] Aggelos Kiayias, Hong-Sheng Zhou, and Vassilis Zikas. Fair and robust multi-party computation using a global transaction ledger. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, pages 705–734, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[75] Uri Klarman, Soumya Basu, Aleksandar Kuzmanovic, and Emin Gün Sirer. bloxroute: A scalable trustless blockchain distribution network whitepaper.

[76] S. Knight, H.X. Nguyen, N. Falkner, R. Bowden, and M. Roughan. The internet topology zoo. *Selected Areas in Communications, IEEE Journal on*, 29(9):1765 –1775, october 2011.

[77] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. OmniLedger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy*, pages 583–598. IEEE Computer Society Press, May 2018.

[78] Ranjit Kumaresan and Iddo Bentov. Amortizing secure computation with penalties. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 418–429. ACM Press, October 2016.

[79] Ranjit Kumaresan, Vinod Vaikuntanathan, and Prashant Nalini Vasudevan. Improvements to secure computation with penalties. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 406–417. ACM Press, October 2016.

[80] Yoad Lewenberg, Yonatan Sompolinsky, and Aviv Zohar. Inclusive block chain protocols. In Rainer Böhme and Tatsuaki Okamoto, editors, *FC 2015*, volume 8975 of *LNCS*, pages 528–547. Springer, Heidelberg, January 2015.

[81] Joshua Lind, Ittay Eyal, Peter Pietzuch, and Emin Gün Sirer. Teechan: Payment channels using trusted execution environments. *arXiv preprint arXiv:1612.07766*, 2016.

[82] Richard J Lipton and Rafail Ostrovsky. Micro-payments via efficient coin-flipping. In *International Conference on Financial Cryptography*, pages 1–15. Springer, 1998.

[83] Richard J. Lipton and Rafail Ostrovsky. Micropayments via efficient coin-flipping. In Rafael Hirschfeld, editor, *FC'98*, volume 1465 of *LNCS*, pages 1–15. Springer, Heidelberg, February 1998.

[84] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 17–30. ACM Press, October 2016.

[85] Giulio Malavolta, Pedro Moreno-Sanchez, Aniket Kate, and Matteo Maffei. Silentwhispers: Enforcing security and privacy in credit networks. NDSS, 2017.

[86] Giulio Malavolta, Pedro Moreno-Sanchez, Aniket Kate, Matteo Maffei, and Srivatsan Ravi. Concurrency and privacy with payment-channel networks. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 455–471. ACM Press, October / November 2017.

[87] Giulio Malavolta, Pedro Moreno-Sanchez, Aniket Kate, Matteo Maffei, and Srivatsan Ravi. Concurrency and privacy with payment-channel networks. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 455–471, New York, NY, USA, 2017. ACM.

[88] Giulio Malavolta, Pedro Moreno-Sanchez, Clara Schneidewind, Aniket Kate, and Matteo Maffei. Multi-hop locks for secure, privacy-preserving and interoperable payment-channel networks. Cryptology ePrint Archive, Report 2018/472, 2018. `https://eprint.iacr.org/2018/472`.

[89] Giulio Malavolta, Pedro Moreno-Sanchez, Clara Schneidewind, Aniket Kate, and Matteo Maffei. Anonymous multi-hop locks for blockchain scalability and interoperability. In *NDSS*, 2019.

[90] Giulio Malavolta, Pedro Moreno-Sanchez, Clara Schneidewind, Aniket Kate, and Matteo Maffei. Anonymous multi-hop locks for blockchain scalability and interoperability. In *Network and Distributed Systems Security Symposium*, 2019.

[91] David Mazieres. The stellar consensus protocol: A federated model for internet-level consensus. *Stellar Development Foundation*, 2015.

[92] Patrick McCorry, Surya Bakshi, Iddo Bentov, Andrew Miller, and Sarah Meiklejohn. Pisa: Arbitration outsourcing for state channels. Cryptology ePrint Archive, Report 2018/582, 2018. `https://eprint.iacr.org/2018/582`.

[93] Patrick McCorry, Chris Buckland, Surya Bakshi, Karl Wüst, and Andrew Miller. You sank my battleship! a case study to evaluate state channels as a scaling solution for cryptocurrencies.

[94] Patrick McCorry, Malte Möser, Siamak Fayyaz Shahandashti, and Feng Hao. Towards bitcoin payment networks. In Joseph K. Liu and Ron Steinfeld, editors, *ACISP 16, Part I*, volume 9722 of *LNCS*, pages 57–76. Springer, Heidelberg, July 2016.

[95] Patrick McCorry, Malte Möser, Siamak F Shahandasti, and Feng Hao. Towards bitcoin payment networks. In *Australasian Conference on Information Security and Privacy*, pages 57–76. Springer, 2016.

[96] Silvio Micali and Ronald L. Rivest. Micropayments revisited. In Bart Preneel, editor, *CT-RSA 2002*, volume 2271 of *LNCS*, pages 149–163. Springer, Heidelberg, February 2002.

[97] Silvio Micali and Ronald L Rivest. Micropayments revisited. In *Cryptographers Track at the RSA Conference*, pages 149–163. Springer, 2002.

[98] Andrew Miller, Iddo Bentov, Surya Bakshi, Ranjit Kumaresan, and Patrick McCorry. Sprites and state channels: Payment networks that go faster than lightning. In Ian Goldberg and Tyler Moore, editors, *FC 2019*, volume 11598 of *LNCS*, pages 508–526. Springer, Heidelberg, February 2019.

[99] Ayelet Mizrahi and Aviv Zohar. Congestion attacks in payment channel networks. *arXiv preprint arXiv:2002.06564*, 2020.

[100] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[101] Paypal Network. Paypal Network. `https://www.paypal.com/jp/home`, 2018. [Online; accessed 17-October-2018].

[102] Visa Network. Visa Network. `https://www.visa.ca/`, 2018. [Online; accessed 17-October-2018].

[103] Andrew M. Odlyzko. The case against micropayments. In Rebecca Wright, editor, *FC 2003*, volume 2742 of *LNCS*, pages 77–83. Springer, Heidelberg, January 2003.

[104] Olaoluwa Osuntokun. [lightning-dev] amp: Atomic multi-path payments over lightning. https://lists.linuxfoundation.org/pipermail/lightning-dev/2018-February/000993.html. Accessed: 2018-10-11.

[105] Rafael Pass and abhi shelat. Micropayments for decentralized currencies. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015*, pages 207–218. ACM Press, October 2015.

[106] Rafael Pass and Elaine Shi. The sleepy model of consensus. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part II*, volume 10625 of *LNCS*, pages 380–409. Springer, Heidelberg, December 2017.

[107] Rafael Pass and Elaine Shi. Thunderella: Blockchains with optimistic instant confirmation. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 3–33. Springer, Heidelberg, April / May 2018.

[108] Christian PDecker, Rusty Russel, and Olaoluwa Osuntokun. eltoo: A simple layer2 protocol for bitcoin. *See https://blockstream.com/eltoo.pdf*, 2017.

[109] Cristina Pérez-Solà, Alejandro Ranchal-Pedrosa, Jordi Herrera-Joancomartí, Guillermo Navarro-Arribas, and Joaquin Garcia-Alfaro. Lockdown: Balance availability attack against lightning network channels. In *International Conference on Financial Cryptography and Data Security*, pages 245–263. Springer, 2020.

[110] Dennis Peterson. Sparky: A lightning network in two pages of solidity. http://wpage.unina.it/giovanni.distasi/pub/blockchain2018-main.pdf, 2018. Accessed: 2019-03-30.

[111] Dmytro Piatkivskyi and Mariusz Nowostawski. Split payments in payment networks. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pages 67–75. Springer, 2018.

[112] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments. *See https://lightning. network/lightning-network-paper. pdf*, 2016.

[113] Pavel Prihodko, Slava Zhigulin, Mykola Sahno, Aleksei Ostrovskiy, and Olaoluwa Osuntokun. Flare: An approach to routing in lightning network. *White Paper (bitfury. com/content/5-white-papers-research/whitepaper_flare_an_approach_to_routing_in_lightning_n etwork_7_7_2016. pdf)*, 2016.

[114] Paul Resnick and Rahul Sami. Sybilproof transitive trust protocols. In *Proceedings of the 10th ACM Conference on Electronic Commerce*, EC '09, pages 345–354, New York, NY, USA, 2009. ACM.

[115] Ronald L. Rivest. Electronic lottery tickets as micropayments. In Rafael Hirschfeld, editor, *FC'97*, volume 1318 of *LNCS*, pages 307–314. Springer, Heidelberg, February 1997.

[116] Ronald L Rivest. Electronic lottery tickets as micropayments. In *International Conference on Financial Cryptography*, pages 307–314. Springer, 1997.

[117] Stefanie Roos, Martin Beck, and Thorsten Strufe. Anonymous addresses for efficient and resilient routing in f2f overlays. In *Computer Communications, IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on*, pages 1–9. IEEE, 2016.

[118] Stefanie Roos, Pedro Moreno-Sanchez, Aniket Kate, and Ian Goldberg. Settling payments fast and private: Efficient decentralized routing for path-based transactions. *arXiv preprint arXiv:1709.05748*, 2017.

[119] Stefanie Roos, Pedro Moreno-Sanchez, Aniket Kate, and Ian Goldberg. Settling payments fast and private: Efficient decentralized routing for path-based transactions. *arXiv preprint arXiv:1709.05748*, 2017.

[120] Vibhaalakshmi Sivaraman, Shaileshh Bojja Venkatakrishnan, Mohammad Alizadeh, Giulia Fanti, and Pramod Viswanath. Routing cryptocurrency with the spider network. *arXiv preprint arXiv:1809.05088*, 2018.

[121] Yonatan Sompolinsky, Yoad Lewenberg, and Aviv Zohar. SPECTRE: A fast and scalable cryptocurrency protocol. Cryptology ePrint Archive, Report 2016/1159, 2016. `http://eprint.iacr.org/2016/1159`.

[122] Yonatan Sompolinsky and Aviv Zohar. Accelerating Bitcoin's transaction processing. Fast money grows on trees, not chains. Cryptology ePrint Archive, Report 2013/881, 2013. `http://eprint.iacr.org/2013/881`.

[123] Yonatan Sompolinsky and Aviv Zohar. PHANTOM: A scalable BlockDAG protocol. Cryptology ePrint Archive, Report 2018/104, 2018. `https://eprint.iacr.org/2018/104`.

[124] Jeremy Spilman. [bitcoin-development] anti dos for tx replacement. https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2013-April/002433.html, 2013. Accessed: 2018-09-03.

[125] Steemit. Steemit. `https://steemit.com/`, 2018. [Online; accessed 6-May-2018].

[126] Taisei Takahashi and Akira Otsuka. Short paper: Secure offline payments in bitcoin. Third Workshop on Trusted Smart Contracts, 2019.

[127] P. F. Tsuchiya. The landmark hierarchy: A new hierarchy for routing in very large networks. *SIGCOMM Comput. Commun. Rev.*, 18(4):35–42, August 1988.

[128] Nicko van Someren, Andrew M. Odlyzko, Ronald L. Rivest, Tim Jones, and Duncan Goldie-Scot. Does anyone really need micropayments? In Rebecca Wright, editor, *FC 2003*, volume 2742 of *LNCS*, pages 69–76. Springer, Heidelberg, January 2003.

[129] David Wheeler. Transactions using bets. In *International Workshop on Security Protocols*, pages 89–92. Springer, 1996.

[130] David Wheeler. Transactions using bets. In Mark Lomas, editor, *Security Protocols*, pages 89–92, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[131] Bitcoin Wiki. Block size limit controversy. `https://en.bitcoin.it/wiki/Block_size_limit_controversy`, 2018. [Online; accessed 17-October-2018].

[132] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151:1–32, 2014.

[133] Beverly Yang and Héctor García-Molina. PPay: Micropayments for peer-to-peer systems. In Sushil Jajodia, Vijayalakshmi Atluri, and Trent Jaeger, editors, *ACM CCS 2003*, pages 300–310. ACM Press, October 2003.

[134] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. RapidChain: Scaling blockchain via full sharding. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 931–948. ACM Press, October 2018.

[135] Alexei Zamyatin, Dominik Harz, Joshua Lind, Panayiotis Panayiotou, Arthur Gervais, and William J. Knottenbelt. XCLAIM: Interoperability with cryptocurrency-backed tokens. Cryptology ePrint Archive, Report 2018/643, 2018. `https://eprint.iacr.org/2018/643`.

[136] Yuncong Zhang, Yu Long, Zhen Liu, Zhiqiang Liu, and Dawu Gu. Z-channel: Scalable and efficient scheme in zerocash. In Willy Susilo and Guomin Yang, editors, *ACISP 18*, volume 10946 of *LNCS*, pages 687–705. Springer, Heidelberg, July 2018.