Thora: Atomic and Privacy-Preserving Multi-Channel Updates

Lukas Aumayr*
TU Wien
Vienna, Austria
lukas.aumayr@tuwien.ac.at

Kasra Abbaszadeh*
University of Maryland
College Park, USA
kasraz@umd.edu

Matteo Maffei
Christian Doppler Laboratory
Blockchain Technologies for the
Internet of Things, TU Wien
Vienna, Austria
matteo.maffei@tuwien.ac.at

ABSTRACT

Most blockchain-based cryptocurrencies suffer from a heavily limited transaction throughput, which is a barrier to their growing adoption. Payment channel networks (PCNs) are one of the promising solutions to this problem. PCNs reduce the on-chain load of transactions and increase the throughput by processing many payments off-chain. In fact, any two users connected via a path of payment channels (i.e., joint addresses between the two channel end-points) can perform payments, and the underlying blockchain is used only when there is a dispute between users. Unfortunately, payments in PCNs can only be conducted securely along a path, which prevents the design of many interesting applications. Moreover, the most widely used implementation, the Lightning Network in Bitcoin, suffers from a collateral lock time linear in the path length, it is affected by security issues, and it relies on specific scripting features called Hash Timelock Contracts that hinders the applicability of the underlying protocol in other blockchains.

In this work, we present Thora, the first Bitcoin-compatible off-chain protocol that enables the atomic update of arbitrary channels (i.e., not necessarily forming a path). This enables the design of a number of new off-chain applications, such as payments across different PCNs sharing the same blockchain, secure and trustless crowdfunding, and channel rebalancing. Our construction requires no specific scripting functionalities other than digital signatures and timelocks, thereby being applicable to a wider range of blockchains. We formally define security and privacy in the Universal Composability framework and show that our cryptographic protocol is a realization thereof. In our performance evaluation, we show that our construction requires only constant collateral, independently from the number of channels, and has only a moderate off-chain communication as well as computation overhead.

CCS CONCEPTS

• **Security and privacy → Distributed systems security**.

KEYWORDS

cryptocurrencies; payment channels; multi-channel update; atomicity; privacy; blockchain

*These two authors contributed equally to the work.

CCS '22, November 7–11, 2022, Los Angeles, CA, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9450-5/22/11.
https://doi.org/10.1145/3548606.3560556

1 INTRODUCTION

Permissionless cryptocurrencies such as Bitcoin [28] use consensus mechanisms to verify transactions in a decentralized way and record them in a public and distributed ledger. This approach has inherent scalability issues, resulting in a low transaction throughput and a long confirmation latency. These limitations prevent cryptocurrencies from meeting the growing user demands, especially when we compare them with centralized payment networks, like Visa, which handle tens of thousands of transactions per second and confirm transactions usually within seconds.

Off-chain protocols constitute one of the most promising solutions to tackle this scalability issue. Instead of recording every transaction on the public ledger, users exchange and keep their transactions off-chain and use the ledger only as a fallback when there are disputes in order to keep their funds. One of the promising off-chain protocols are Payment Channels (PCs) which are deployed at scale in cryptocurrencies such as Bitcoin and Ethereum [26, 29]. Intuitively, a channel is a shared address that allows two parties to maintain and update a private ledger through off-chain transactions. In a bit more detail, looking at Bitcoin's unspent transaction output (UTXO) model, users first open a PC by locking some coins in a 2-of-2 multi-signature output. Then, they can update the balance in the PC arbitrarily many times by exchanging signed transactions. Each of the users can close the PC by publishing the last state on-chain. This allows them to perform many transactions while burdening the ledger with only two transactions.

1.1 HTLC-based PCNs and their limitations

Payment channel networks (PCNs) like the Lightning Network (LN) [29] and Raiden [1] generalize this approach, by allowing two users to pay each other as long as they are connected by a path of channels with enough capacity. Such a payment in a PCN, also called a multi-hop payment (MHP), requires updating each channel on the path. The challenge here is to ensure atomicity, i.e., either all channels are updated consistently or none, such that no user is at risk of losing money. In the most popular PCN, i.e. the Lightning Network, atomicity is achieved through Hash Timelock Contracts (HTLCs) [29], which make the payments on each channel on the path conditioned on revealing the preimage of a certain hash. The receiver has to reveal that preimage in order to receive the money and then all intermediaries from right to left are incentivized to update their left channel in order to claim the money of the payment. An example of a payment using HTLCs is shown in Figure 1.

HTLC-based PCNs, however, have the following fundamental drawbacks:

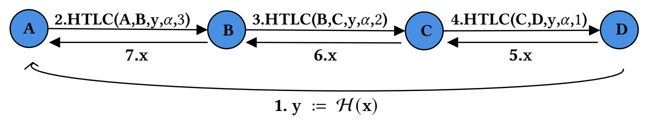

Figure 1: An example of a payment in LN from A to D for a value α using HTLC contracts. An HTLC contract denoted by HTLC(Alice, Bob, x, y, t), shows the following conditions: (i) If timeout t expires, Alice gets back the locked x coins. (ii) If Bob reveals a value r, such that $\mathcal{H}(r) = y$, before timeout t, Alice pays x coins to Bob.

Collateral All parties on the path have to lock the payment amount α up to a period of *locktime*. The payment amount multiplied by the locktime is called *collateral*, a metric that has been used in previous work, e.g., [9, 18, 27]. In addition, parties can impose fees for the service of forwarding payments. In the case of HTLCs, each party has to lock a collateral that is linear in the size of the path n, i.e., $\Theta(\alpha \cdot n \cdot \delta)$, where δ is a security parameter defining the time by which users have to react in case of misbehavior from others (in Lightning, δ is one day).

Due to the linear collateral, the effects of *griefing attacks* [18] on HTLC-based PCNs are particularly severe. In a griefing attack, a malicious user starts a multi-hop payment to itself with the intent to block coins owned by intermediaries. The attacker manages to lock up α coins in $n-1$ honest channels. The fact that the lock duration is also linear in the path length amplifies the effects of this attack further. The malicious user subsequently lets the payment fail to limit the overall network throughput or to lock coins of specific users.

Weak atomicity Lightning guarantees only a weak form of atomicity, that is, only the two adjecent channels of an honest node are updated consistenlty. In particular, Lightning is vulnerable to the *wormhole attack* [24], where two colluding malicious users can skip honest users in the phase where they reveal the preimage. This does not lead to a loss in funds for the honest users, but the malicious users can steal the fees originally intended for the honest users.

Path restriction Since HTLC-based PCN protocols rely on an incentive based forwarding of a preimage via a path to ensure that honest users do not lose funds, these protocols are limited to payments over a path of channels. This rules out other topologies reflecting relevant financial applications (e.g., crowd-funding can be seen as a star topology where all nodes update their channel with the beneficiary).

Value privacy In Lightning, intermediaries implicitly learn the paid amount, as the value has to be the same (except for some fee) over all channels within the path to ensure atomicity of the protocol.

1.2 Related work

Recently, various protocols have been designed to overcome the aforementioned issues, but they all fall short of some property, as summarized in Table 1.

Anonymous Multi-Hop Locks (AMHL) prevent the wormhole attack by dispensing from HTLCs in favor of adaptor signatures, a mechanism in which the secret is somewhat embedded in the randomness of the signature and revealed once that signature is published, but they still suffer from linear collateral and only support path-based payments.

The Atomic Multi-Channel Updates (AMCU) protocol [18] attempts to achieve payments with constant collateral and also to support more generic applications than path-formed payments. Unfortunately, AMCU is not secure: It is vulnerable to *channel closure attacks* [19], where users honestly updating their channels can be victim of double-spending attacks, which can lead to a loss of funds for honest users.

Blitz [9] is a recently proposed payment protocol for multi-hop payments, which in contrast to Lightning requires only one round of communication through the path with constant collateral. However, Blitz supports only path-based payments.

Sprites [27] is the only secure protocol supporting atomic multi-channel updates with constant collateral. In fact, the paper addresses only path-based payments, but we conjecture that the protocol could in principle be modified so as to support arbitrary topologies and also to hide the paid amount. Unfortunately, Sprites inherently requires Turing-complete scripting, which makes it inapplicable to blockchain technologies with limited scripting capabilities, such as Bitcoin itself. A Turing complete scripting language provides more expressiveness, but it also enlarges the trusted computing base, opens the door to programming bugs, and makes computations more expensive (e.g., in terms of gas fees in Ethereum).

Hence, it is both a foundational and practically relevant question whether or not atomic multi-channel updates with constant collateral are possible at all in blockchains with limited scripting languages like Bitcoin. Indeed, it was conjectured in [27] that they are not.

1.3 Our contribution

In this paper, we show that the aforementioned conjecture is incorrect. In particular,

- We introduce Thora, the first secure Bitcoin-compatible protocol with constant collateral for atomic, multi-channel updates. The constant collateral property not only makes the protocol financially sustainable for a large number of channels, but also mitigates the threat of griefing attacks. Thora only requires signatures and timelocks, and it is thus compatible with a number of cryptocurrencies, such as Bitcoin, Stellar, and Ripple. In addition, Thora supports payments over channels with arbitrary topologies, thereby enabling a variety of interesting applications. Finally, Thora achieves value privacy, i.e., the channel owners can synchronize their payments without necessarily disclosing the individual payment amounts.
- We formally model our protocol in the *Global Universal Composability* (GUC) framework [15], analyzing its security and privacy properties. For this, we define an ideal functionality which captures the security and privacy notions of interest and prove that Thora constitutes a GUC-realization thereof.
- We conduct a complexity analysis and performance evaluation, demonstrating the practicality of Thora.
- We instantiate Thora in the context of several applications that go beyond simple path-formed payments, such as mass payments,

Table 1: Comparing different payment methods: Lightning Network, Anonymous Multi-Hop Locks (AMHL), Sprites, Payment Trees, Atomic Multi-Channel Updates(AMCU), Blitz, and our construction. Studied features are: atomicity property, path restriction, need for Turing-complete smart contracts, size of per party collateral, and value privacy. For the latter, note that there are constructions that do not inherently leak the value transferred in individual channels, but they can only be used for applications (i.e., payments) that require the same value in all channels.

	Atomicity	Path restriction	Smart contract	pp Collateral	Value privacy
Lightning Network [29]	No	Yes	No	Linear	application leak
AMHL [24]	Yes	Yes	No	Linear	application leak
AMCU [18]	No	No	No	Constant	No
Payment Trees [19]	Yes	Yes	No	Logarithmic	No
Blitz [9]	Yes	Yes	No	Constant	application leak
Sprites [27]	Yes	No	Yes	Constant	Yes
Thora	Yes	No	No	Constant	Yes

channel rebalancing, and crowd-funding, thereby exemplifying the class of off-chain applications enabled by Thora.

2 BACKGROUND

In this section, we provide an overview on the background and the notations used throughout the paper. For more details, we refer the reader to [7, 9, 23].

2.1 UTXO based transactions

We assume the underlying blockchain to be based on the *unspent transaction output* (UTXO) model, like Bitcoin. In this model, *coins*, or the units of currency, exist in *outputs* of *transactions*. We represent each output as a tuple $\theta := (\mathsf{cash}, \phi)$ where $\theta.\mathsf{cash}$ is the output value, and $\theta.\phi$ is the condition required to spend the output. We encode the condition in the scripting language used by the underlying cryptocurrency. The notation $\mathsf{OneSig}(U)$ denotes the condition that a digital signature w.r.t. U's public key is required for spending an output. If multiple signatures are required, we write $\mathsf{MultiSig}(U_1, U_2, ..., U_n)$.

Users can transfer the ownership of outputs via transactions. A transaction spends a non-empty list of unspent outputs (transaction inputs) and maps them to a list of new unspent outputs (transaction outputs). Formally a transaction is denoted as a tuple $\mathsf{tx} := (\mathsf{id}, \mathsf{input}, \mathsf{output})$. $\mathsf{tx.id} \in \{0,1\}^*$ is the identifier, set to be the hash of inputs and outputs, $\mathsf{tx.id} = \mathcal{H}(\mathsf{tx.input}, \mathsf{tx.output})$, where $\mathcal{H}$ is modeled as a random oracle. tx.input denotes the list of identifiers of the inputs and tx.output denotes the list of new outputs. Also we let $\overline{\mathsf{tx}} := (\mathsf{id}, \mathsf{input}, \mathsf{output}, \mathsf{witness})$ or for convenience also $\overline{\mathsf{tx}} = (\mathsf{tx}, \mathsf{witness})$ denote a full transaction. $\overline{\mathsf{tx}}.\mathsf{witness}$ consists of witnesses for the spending conditions of the transaction's inputs. Only valid transactions can be recorded on the public ledger $\mathcal{L}$ (the blockchain). A transaction is considered valid if (i) its inputs are not spent by other transactions in $\mathcal{L}$, (ii) the sum of its outputs is not greater than the sum of inputs, and (iii) the transaction provides valid witnesses fulfilling the spending conditions of every input. In practice, transactions are not recorded on the ledger and published immediately, but only after the participants in the distributed consensus accept them. We use Δ to denote the upper bound on the time it takes for a valid transaction to be published and accepted to $\mathcal{L}$.

Using the scripting language, we can encode more complex conditions on transaction outputs than simple ownerships. To better visualize transactions, we use charts in which transactions are represented as rounded rectangles and inputs as incoming arrows. Boxes inside transactions represent outputs and the values in these boxes determine the amounts of coins stored in the outputs. Outgoing arrows from an output are used to encode the condition under which said output can be spent. In particular, below an arrow, we identify who can spend an output by listing one or more public keys. A valid transaction must contain signatures that verify under these public keys. Above the arrow, we write additional conditions that are required for spending the output. These conditions can be any script supported by the scripting language of the underlying blockchain, but in this work, we only use time-locks. For denoting relative time-locks, we write $\mathsf{RelTime}(t)$ or $+t$, which means that the output can be spent only if at least t rounds have passed since the transaction holding this output was accepted on $\mathcal{L}$. For denoting absolute time-locks, we use $\mathsf{AbsTime}(t)$ or $\geq t$, which means that the output can be spent only if the round t has already passed. If an output condition is a disjunction of several conditions, i.e., $\phi = \phi_1 \vee \phi_2 \cdots \vee \phi_n$ we draw a diamond in the output box and put each condition ϕ_i below/above its own arrow. For the conjunction of several conditions, we write $\phi = \phi_1 \wedge \phi_2 \ldots \wedge \phi_n$. We illustrate an example of our transaction charts in Figure 2.

2.2 Payment channels

Using payment channels, two users can perform an arbitrary number of payments off-chain by publishing only two transactions on the ledger, one for funding and one for closing. Through the funding transaction tx^{f}, users jointly lock up some coins in a shared multi-signature output, thereby opening a new channel. To avoid having their funds locked, the two users exchange signed transactions spending from tx^{f}, and assigning new balances for users, before posting tx^{f} on-chain. Users can perform payments by exchanging new transactions that reassign their balances. These transactions holding the balances are called *states* of the channel. When the two users are done, they can close the channel by posting the last state to the ledger.

For readability, we omit the implementation details and instead use payment channels in a black-box manner, using the following abstraction: Both users have the same transaction $\mathsf{tx}^{\mathsf{state}}$, which

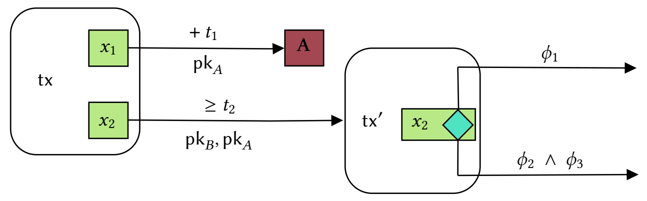

Figure 2: The left transaction tx has two outputs, one of value x_1 that can be spent by A, with a transaction signed w.r.t. pk_A, but only if at least t_1 rounds passed since tx is accepted on the blockchain. The other output of value x_2 can be spent by a transaction signed w.r.t. pk_A and pk_B at or after round t_2. The right transaction tx′ has one input, which is the second output of tx containing x_2 coins, and has only one output, which is of value x_2 and can be spent by a transaction whose witness satisfies the output condition $\phi_1 \vee (\phi_2 \wedge \phi_3)$. The inputs of tx are not shown.

holds the outputs representing the last state of the channel. Furthermore, we assume that the users can only publish the last $\mathsf{tx}^{\mathsf{state}}$ on the ledger. In practice there is a punishment mechanism in place, which gives the total channel capacity to the honest party in case a malicious party publishes an old state. We refer the reader to [7, 23, 24] for more details.

We denote payment channels as $\overline{\gamma} := (\mathsf{id}, \mathsf{users}, \mathsf{cash}, \mathsf{st})$, where $\overline{\gamma}.\mathsf{id} \in \{0,1\}^*$ is the unique identifier of the channel, $\overline{\gamma}.\mathsf{users} \in \mathcal{P}^2$ contains addresses of two involved parties (out of the set of all parties $\mathcal{P}$), $\overline{\gamma}.\mathsf{cash} \in \mathbb{R}_{\geq 0}$ is the total number of coins in the channel, and $\overline{\gamma}.\mathsf{st} := (\mathsf{output}_1, \mathsf{output}_2, ..., \mathsf{output}_n)$ is the last state of the channel and contains a list of outputs. The balance of both users can be inferred from the current state $\overline{\gamma}.\mathsf{st}$, and $\overline{\gamma}.\mathsf{balance}(P)$ returns the amount of coins owned by P for $P \in \overline{\gamma}.\mathsf{users}$. We define a channel skeleton γ for a channel $\overline{\gamma}$, as $\gamma := (\overline{\gamma}.\mathsf{id}; \overline{\gamma}.\mathsf{users})$. Moreover, in the context of our multi-channel updates protocol, based on the direction of the payment in each channel γ, we define one of the involving parties as sender, which is denoted by $\gamma.\mathsf{sender} \in \gamma.\mathsf{users}$, and one as receiver which is denoted by $\gamma.\mathsf{receiver} \in \gamma.\mathsf{users}$.

2.3 Payment channel networks

A payment channel network (PCN) [23] is a graph consisting of vertices, representing the users, and edges, representing the channels between pairs of users. PCNs enable payments between any users connected through a path of open payment channels. This is called a *multi-hop payment*. Assume user U_0 wants to pay user U_n, but there is no direct payment channel between them. Instead, U_0 has an open payment channel γ_0 with U_1, U_1 has an open payment channel γ_1 with U_2 and so on, until the receiver U_n. An MHP allows transferring coins from U_0 to U_n through intermediaries $\{U_i\}_{i\in[1,n-1]}$ atomically in a secure way, which means that no honest user is at the risk of losing money.

HTLC. The Lightning Network (LN) [29] achieves atomicity by using a technique called *Hash Timelock Contract* (HTLC). This contract can be executed by two parties sharing an open payment channel, e.g., Alice and Bob. First, Alice locks some of her coins in an output that is spendable if one of the following conditions is fulfilled. (i) If a specified timeout t expires, Alice gets her money back. (ii) If Bob presents a pre-image r_A for a certain hash value $\mathcal{H}(r_A)$ chosen by Alice, Bob gets the money.

An MHP in LN concatenates several HTLCs aiming for an atomic payment. In a nutshell, suppose again there is a sender U_0 who wants to pay α coins to a receiver U_n through some intermediaries $\{U_i\}_{i\in[1,n-1]}$. The payment receiver U_n chooses a random value r and sends $y = \mathcal{H}(r)$ to the sender. Then the sender sets up an HTLC with U_1 by creating a new state with three outputs $(\mathsf{output}_0, \mathsf{output}_1, \mathsf{output}_2)$ where output_0 contains α coins, output_1 contains U_0's balance minus α, and output_2 contains U_1's balance. The HTLC specifies that output_0 can be spent by U_0 if timeout $n \cdot T$ is expired, or by U_1, if she knows a value x such that $\mathcal{H}(x) = y$. Then U_1 sets up an HTLC with U_2 in a similar manner using the same hash y but a different time, $(n-1) \cdot T$. This step is repeated until the receiver is reached, with a timeout of T. We call this process the *setup phase*. Thereafter, the receiver can reveal r and claim α coins from the left neighbor. Using r, U_{n-1} can claim α coins from U_{n-2} and so on, in a second phase, which is called *open phase*. In this way, all payments can be performed atomically through the path.

Note that in the open phase, each pair of parties can either agree to update their channel to a new state off-chain, where finally U_n has α coins more, or otherwise the receiver can publish the state and a transaction with witness r on-chain. The timelocks of the HTLCs are staggered, i.e., they increase from right to left, because we need to give enough time to an intermediary party to claim her money from the left neighbor, when her right neighbor reveals r and spends the output of the corresponding HTLC. LN payments thus require (i) two rounds of pairwise, sequential communication from sender to receiver and (ii) a linear collateral lock time in terms of the path length. This opens the door to denial-of-service attacks, also called griefing attacks [18] in the literature. Another attack that threatens the security of the HTLC-based protocols is the *wormhole* attack [24]. This attack allows two colluding users to exclude honest intermediaries from the payment and steal their fees.

Blitz. Blitz [9] recently improved on that by requiring only one round of communication through the path, and a constant collateral lock time, while guaranteeing security in the presence of malicious intermediaries. In this protocol, the sender creates a unique transaction *Enable Refund*, which is denoted by $\mathsf{tx}^{\mathsf{er}}$. This transaction acts as a global event and makes the refunds atomic, following a *pay-unless-revoke* paradigm. On a high level, each party U_i for $i \in [0, n-1]$, creates an output of α that is spendable in two ways: (i) U_{i+1} can claim it after some specific time T, or (ii) U_i can refund the coins if $\mathsf{tx}^{\mathsf{er}}$ is on the ledger before that time T. If all channels are updated from sender to receiver in this way, the receiver sends a confirmation to the sender and the payment is considered successful. Otherwise, if any update fails, the sender posts $\mathsf{tx}^{\mathsf{er}}$ before time T to the ledger to trigger all refunds.

Note that in LN, payments in the pessimistic case are performed sequentially. In Blitz, instead, in the case of failure, all refunds can be performed in parallel whenever $\mathsf{tx}^{\mathsf{er}}$ appears on the ledger. Because of that, the collateral lock time in Blitz for each party is constant, thereby significantly reducing the effects of a griefing attack against Blitz compared to protocols with a linear collateral lock time.

3 SOLUTION OVERVIEW

In this work, we present Thora, the first Bitcoin-compatible protocol that enables the atomic update of arbitrary channels, going beyond the path-based topology assumed in HTLC- or Blitz-based payments. In other words, Thora supports multiple senders and receivers, without requiring them to be connected to each other. This feature enables the design of new off-chain applications as well as to perform payments across distinct PCNs sharing the same underlying blockchain. We start by informally presenting the security and privacy goals of interest and then give an intuitive overview of our construction.

3.1 Security and privacy goals

In this work, we focus on two fundamental properties, which we informally define below, referring the reader to Appendix C for the formal definitions.

(S1) Atomicity. The aim of a multi-channel update protocol is to update a set of channels. A multi-channel update protocol achieves atomicity if there are no two channels with at least one honest user each where one update fails and the other one is successful, unless at least one honest user is compensated (i.e., by getting coins she would otherwise not get). In other words, without losing coins (i) a malicious receiver cannot let the update of her channel be successful even though it should fail and (ii) a malicious sender cannot let the update fail, even though it should be successful. Note that a malicious (irrational) user can always forfeit their own coins, e.g., by posting an old channel state, but as this is to the benefit of the honest user, we do not consider it as breaking atomicity.

(P1) Strong value privacy. We say that a multi-channel update protocol achieves value privacy if in the optimistic case (i.e., when the protocol is executed entirely offline), for each channel, no party except for the channel owners can determine the payment value. Note that this property is stronger than value privacy as defined in AMCU [23]. In AMCU, each channel's payment value is known to all parties involved in the protocol, and the privacy of values is preserved only against parties not involved in the protocol.

Assumptions. We assume that there is a secure and authenticated channel between each protocol participant. This can be realized in practice by establishing TLS channels. Also, we do not consider the side channels that can be established by probing the nodes in the network or by observing the opening and closing on-chain operations, as these constitute orthogonal problems that affect all PCNs and can be mitigated with dedicated techniques (e.g. [16]).

3.2 Key idea

The approach we follow to construct our protocol is reminiscent of the *pay-unless-revoke* paradigm adopted in Blitz [9], but it proceeds the other way around and it should thus be seen as a *revoke-unless-pay* paradigm, as discussed below. In particular, for each channel, we aim to design an update contract that simultaneously allows the receiver to claim her coins if all payments are successful and allows the sender to refund her coins if at least one channel fails to perform the payment. We propose our solution in an incremental way. First, we start with a high-level overview of the approach. Then, we discuss the challenges and possible solutions, until reaching the final protocol.

Let $\{\gamma_i\}_{i\in[1,n]}$ be the set of involved payment channels. For each channel γ_i, based on the payment direction, we define one party as the sender, denoted by γ_i.sender, and one as the receiver, denoted by γ_i.receiver. We call the payment value for this channel α_i. As a high-level abstraction, γ_i.sender splits α_i coins from her balance in the channel's current state, and generates a new output. This output can be spent by the receiver if all payments are successful, or can be refunded to the sender if at least one payment fails. In other words, we need to overcome two challenges. First, the design should be such that *if a sender refunds her coins, then all other senders can also do that.* Second, *if the payment in a channel is successful or a receiver is able to claim her coins, then payments in all other channels are forced, and senders cannot refund.*

For the first challenge, we make all refunds possible only if a timeout T expires, so after this time, all senders can refund their coins if the coins have not been spent by the receivers. In other words, we give all users time T to finalize the payments in their channels. If the payment in a channel has not been finalized until this time, the sender can use a refund transaction and get back her coins. T is a protocol parameter, independent of the number of channels, and the same for all channels.

For the second challenge, we make payments atomic using a global event. For each channel, the sender updates the channel and creates a payment transaction, which transfers coins to the receiver only after a global event occurs before time T. When all channels are updated correctly, senders are expected to finalize their channels, transferring coins to their receiver neighbor. In this case, if at least one receiver does not receive coins, the global event will be triggered before time T, and all payment transactions will become valid. Then, receivers can claim their cash. This global event is the appearance of a specific transaction on the ledger, which we call *Enable Payment* transaction, and denote it by $\mathsf{tx}^{\mathsf{ep}}$. This transaction is similar to *Enable Refund* transaction in the Blitz protocol, but the logic is reversed. Instead of refunds, we make payments dependent to a global event.

Update contract. For easing the presentation, let us assume first that there is a trusted user, who creates $\mathsf{tx}^{\mathsf{ep}}$ and is responsible for posting it to the ledger. $\mathsf{tx}^{\mathsf{ep}}$ contains outputs to all receivers, which is the key to achieve atomicity. We discuss the structure of the update contract below, which makes both the payment and the refund available to the channel owners. In more detail, for each channel γ_i, the sender γ_i.sender creates three transactions: $\mathsf{tx}^{\mathsf{state}}$, tx^{r}, and tx^{p}. $\mathsf{tx}^{\mathsf{state}}$ is a new state transaction, where α_i coins from the sender are put in a contract which can be spent by the other two transactions. Transaction tx^{r} refunds back the α_i coins to the sender if a timeout T expires. Transaction tx^{p} has inputs from $\mathsf{tx}^{\mathsf{ep}}$ and $\mathsf{tx}^{\mathsf{state}}$ and transfers the coins to the receiver, if $\mathsf{tx}^{\mathsf{ep}}$ is on the ledger before time T. The design of these transactions is shown in Figure 3. The sender sends $\mathsf{tx}^{\mathsf{state}}$ and the signed tx^{p} to the receiver, who verifies the messages and updates the channel to the new state $\mathsf{tx}^{\mathsf{state}}$ together with the sender. In the case of success, the receiver sends an endorsement to the trusted user.

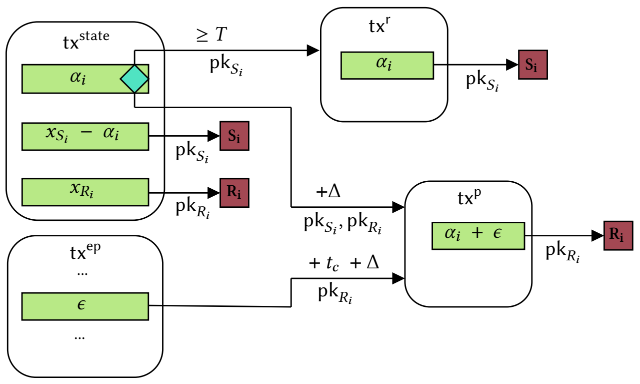

Figure 3: Update contract for the channel γ_i between two neighboring users γ_i.sender and γ_i.receiver with the new state $\mathsf{tx}^{\mathsf{state}}$. x_{S_i} is the amount that $S_i = \gamma_i$.sender owns and x_{R_i} is the amount that $R_i = \gamma_i$.receiver owns in the state before $\mathsf{tx}^{\mathsf{state}}$.

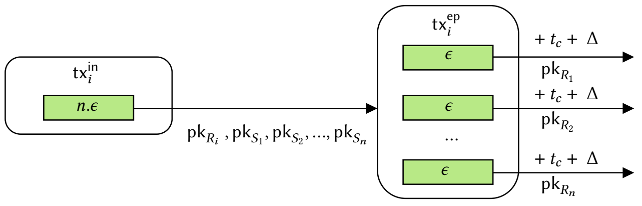

Figure 4: Transaction $\mathsf{tx}_i^{\mathsf{ep}}$ created by receiver R_i for a payment with n channels, where the set of all senders is $\{S_j\}_{j\in[1,n]}$ and the set of all receivers is $\{R_j\}_{j\in[1,n]}$. This transaction enables all payments and spends the output of transaction $\mathsf{tx}_i^{\mathsf{in}}$.

Atomic payments. If the trusted user receives endorsements from all receivers, she informs all parties to finalize their channels and to transfer coins to receivers safely. There are two error cases. (i) The trusted user does not receive the endorsement from every receiver. In this case, no party will get a message from the trusted user to finalize the channel, so all channels are safe, and after time T they can be restored to the initial state based on refund transactions. (ii) If a sender gets the *finalize* message from the trusted user but does not finalize her channel, the corresponding receiver informs the trusted user to put $\mathsf{tx}^{\mathsf{ep}}$ on-chain before time T in order to force all payments.

At this point, our goal is to eliminate the trusted user assumption. Indeed, if we elected one of the parties for creating and publishing $\mathsf{tx}^{\mathsf{ep}}$, that party might act maliciously and break atomicity. For instance, by not posting $\mathsf{tx}^{\mathsf{ep}}$ to the ledger when some senders do not finalize their channel, or by posting $\mathsf{tx}^{\mathsf{ep}}$ when some channels have been updated with $\mathsf{tx}^{\mathsf{state}}$ and some not, payments would no longer be atomic. Our strategy is thus to enable all receivers to publish $\mathsf{tx}^{\mathsf{ep}}$, but only after every channel updated already to $\mathsf{tx}^{\mathsf{state}}$. For this, each receiver creates her own $\mathsf{tx}^{\mathsf{ep}}$. Each $\mathsf{tx}^{\mathsf{ep}}$ has an input conditioned on the public keys of the creator and of all senders, and it has outputs to all receivers. An example of this transaction is shown in Figure 4.

All receivers send their $\mathsf{tx}^{\mathsf{ep}}$ to all other parties, and this time each sender creates one tx^{p} per $\mathsf{tx}^{\mathsf{ep}}$. Then, for each channel, the sender and the receiver jointly update the channel using $\mathsf{tx}^{\mathsf{state}}$ as we discussed earlier. If no error occurs, the receiver sends a first endorsement to all parties instead of the trusted user. Each sender waits until receiving all endorsements to make sure that all channels are updated using $\mathsf{tx}^{\mathsf{state}}$. After that, the sender sends her signature to each $\mathsf{tx}^{\mathsf{ep}}$ to the creator. Eventually, when all receivers get complete signatures to their $\mathsf{tx}^{\mathsf{ep}}$, they send their second endorsement and the senders are safe to start finalizing channels and transfer coins to the receivers, because all channels have been updated with $\mathsf{tx}^{\mathsf{state}}$. If some transfer fails, the receivers can post $\mathsf{tx}^{\mathsf{ep}}$ on the ledger and force all payments.

We now intuitively argue why atomicity and strong privacy hold. For atomicity, an honest sender will only update the channel with her receiver neighbor, if she receives the second endorsement from all receivers, which means that every receiver is able to force payments via $\mathsf{tx}^{\mathsf{ep}}$. Similarly, honest receivers will only give their second endorsement if they received all the signatures from $\mathsf{tx}^{\mathsf{ep}}$. This means that if a malicious user does not send her signature or endorsement to any or some of the users, this will not break atomicity but potentially only prevent updates from taking place or force the updates via some $\mathsf{tx}^{\mathsf{ep}}$. Moreover, if a malicious receiver sends either endorsement prematurely, she will only potentially lose money without side effect to other channels, i.e., the adversary will donate money to the sender without affecting the payments in the other channels. Finally, malicious users are rational, which means they will either refund their money or claim the money from a forced update, if possible.

With regards to privacy, the payment value is only known to the sender and the receiver, and in particular it is not disclosed to the other parties involved in the protocol.

Timelocks. tx^{p} should be valid until time T, and tx^{r} should be valid after that time. The latter can easily be handled by using an absolute timelock of T, which is supported by the underlying scripting language of most cryptocurrencies, including Bitcoin. However, we do not have access to scripting functionalities to define outputs that are valid before time T.

We can solve this problem by applying relative timelocks. In particular, we add a relative timelock of Δ for the transaction tx^{p}, where Δ is the blockchain delay. According to this timelock, if $\mathsf{tx}^{\mathsf{state}}$ appears on the ledger after time T, users have enough time to post tx^{r} before the relative timelock of tx^{p} expires. In other words, tx^{r} is always accepted over tx^{p}, in the case that both are published after time T. On the other hand, if $\mathsf{tx}^{\mathsf{state}}$ appears before time $T - \Delta$, users have enough time to post tx^{p} and force the payment.

One other issue we should consider is the unfair advantage of a receiver who closes her channel in advance and puts her $\mathsf{tx}^{\mathsf{ep}}$ on the ledger just before time $T - \Delta$. In this case, the receiver can post tx^{p} and force the payment in her channel, but other receivers, who have not closed their channels, do not have enough time to react to $\mathsf{tx}^{\mathsf{ep}}$. To prevent this issue and give enough time to all users to close their channels and post tx^{p} to the ledger, we add a relative time of $t_c + \Delta$ to the outputs of $\mathsf{tx}^{\mathsf{ep}}$, where t_c is an upper bound on the time a user needs to close a channel (Figure 3). For more detail on how we prevent race conditions, we refer the reader to Section 8.

We point out that, as in the Lightning Network, honest users are assumed to be online and to monitor the ledger. This assumption is orthogonal to our construction and can be removed using the techniques proposed in the literature for this purpose, e.g., Watchtowers [11, 25].

Protocol overview. To wrap up, our protocol proceeds in four main phases, as described below and visualized in Figure 5.

(1) **Pre-Setup**: Each receiver creates her own $\mathsf{tx}^{\mathsf{ep}}$, and sends it to all other parties. Each $\mathsf{tx}^{\mathsf{ep}}$, in addition to the creator's signature, requires signatures from all senders, and has one output for each receiver.
(2) **Setup**: The senders create $\mathsf{tx}^{\mathsf{state}}$ and tx^{r}, and also one tx^{p} per $\mathsf{tx}^{\mathsf{ep}}$. They send $\mathsf{tx}^{\mathsf{state}}$ and all tx^{p} to their receiver neighbor. Also, they include their signatures for every tx^{p} in the message to their receiver neighbor. This ensures that receivers can post tx^{p} on the ledger regardless of which $\mathsf{tx}^{\mathsf{ep}}$ is posted in the end. Eventually, the receivers verify the messages and send their first endorsement to all parties.
(3) **Confirmation**: When a sender gets all such endorsements, she is sure that all channels have been updated by $\mathsf{tx}^{\mathsf{state}}$. Then, the sender signs each $\mathsf{tx}^{\mathsf{ep}}$ and sends it to the corresponding receiver. When a receiver gets the signatures from all senders, she is able to post her $\mathsf{tx}^{\mathsf{ep}}$ on the ledger, so she sends a second endorsement to all parties.
(4) **Finalizing**: When the senders get the second endorsement from all receivers, they know that all receivers are able to put their $\mathsf{tx}^{\mathsf{ep}}$ on the ledger, so they can start updating their channels safely. When one update fails and the corresponding receiver does not get the coins, she checks if a $\mathsf{tx}^{\mathsf{ep}}$ is on the ledger or else posts her own $\mathsf{tx}^{\mathsf{ep}}$. Either way, she claims her coins via some tx^{p}.

Fast payments. Similar to the Lightning Network, in the case that all users are honest, updates can be carried out almost instantaneously, i.e., the channels are updated as soon as the second endorsements are received from receivers. When the senders are ensured that each receiver has all signatures required for spending her $\mathsf{tx}^{\mathsf{ep}}$, they can safely update their channels and pay coins to their right neighbors.

Honest update. The update contract and the corresponding transactions $\mathsf{tx}^{\mathsf{state}}$, tx^{r}, and tx^{p} are exchanged between two parties sharing a channel to guarantee that honest users do not lose their coins and atomicity holds during the protocol execution. However, when one of the two channel owners is able to convince the other one that she is able to force the payment (or refund) by posting tx^{p} (or tx^{r}) to the ledger, the two parties can update the channel honestly to a state on which both agree. In other words, when both parties of a channel are honest, no on-chain transaction is required.

4 CONSTRUCTION

4.1 Building blocks

Digital signatures. A digital signature scheme consists of three algorithms: KeyGen, Sign, Vrfy.
$(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$ is a PPT algorithm, taking the security parameter 1^λ as input and returning a public key pk and the corresponding secret key sk.
$\sigma \leftarrow \mathsf{Sign}(\mathsf{sk}, m)$ is a PPT algorithm, taking a secret key sk and a message m as inputs and returning a signature σ.
$\{0, 1\} \leftarrow \mathsf{Vrfy}(\sigma, m, \mathsf{pk})$ is a DPT algorithm, taking signature σ, a message m, and a public key pk as inputs, and returning 1 if σ is a valid signature on message m and created by the secret key corresponding to pk. Otherwise it returns 0.

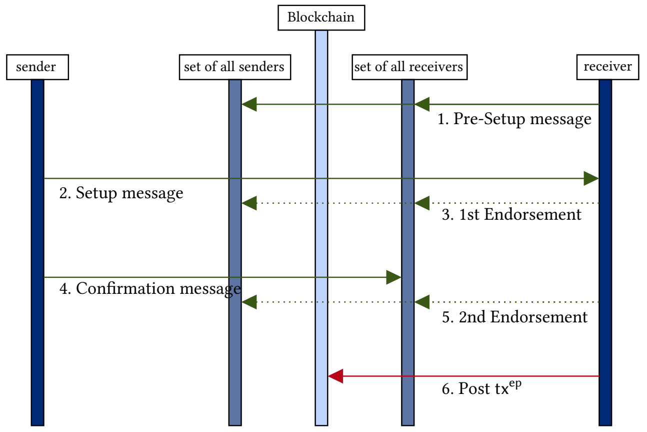

Figure 5: For each channel, first, the receiver sends her own $\mathsf{tx}^{\mathsf{ep}}$ to all other parties (the Pre-Setup message). The sender creates $\mathsf{tx}^{\mathsf{state}}$ and one tx^{p} for each $\mathsf{tx}^{\mathsf{ep}}$, then sends all these transactions to the receiver (Setup message). After verifying the message, the receiver sends her first endorsement to all other parties. When the sender gets all endorsements, she sends her signature to each $\mathsf{tx}^{\mathsf{ep}}$ to its creator (Confirmation message). After getting all signatures and verifying them, the receiver sends the second endorsement to all other parties. Finally, when the receiver has enough signatures as her $\mathsf{tx}^{\mathsf{ep}}$ witnesses, and the payment is not received, she will post her $\mathsf{tx}^{\mathsf{ep}}$ to the ledger.

Ledger and payment channels. In this work, we use a ledger and a PCN as black-boxes. The ledger keeps a record of balances of users and all transactions. The PCN supports the operations *open*, *close*, and *update*. For simplicity, we assume the payment channels involved in the multi-channel updates protocol to be already open. We assume that ledger and PCN expose the following API to the users:

- getBalance(U): Returns the sum of all coins in the UTXOs owned by user U on the ledger.
- splitCoins(U, v, ϕ): Aggregates all UTXOs owned by U and returns a transaction with an output containing v coins, which is conditioned on ϕ. If the balance of U is greater than v, the rest is sent to an address controlled by U. If the balance of U is less than v, the procedure returns $\perp$.
- publishTx($\overline{\mathsf{tx}}$): Appends the transaction $\overline{\mathsf{tx}}$ to the ledger after at most Δ rounds, if witnesses are valid, inputs exist and are unspent, and the sum of coins in their outputs is less than or equal to the sum of coins in the inputs.

- updateChannel($\overline{\gamma}$, tx^{state}): Initiates an update in the channel $\overline{\gamma}$ to the state defined by tx^{state}, when called by a user $\in \overline{\gamma}$.users. The update is performed after at most t_u rounds. Upon the termination, the procedure returns `UPDATE-OK` in the case of success, and `UPDATE-FAIL` in the case of failure to both users.
- closeChannel($\overline{\gamma}$): Closes the channel $\overline{\gamma}$ when called by a user $\in \overline{\gamma}$.users. The latest state $\overline{\gamma}$.st appears on the ledger after at most t_c rounds.

4.2 Protocol description

Let $U := \{(\gamma_i, \alpha_i)\}_{i\in[1,n]}$ be the set of all updates, where $\{\gamma_i\}_{i\in[1,n]}$ denotes the involved payment channels and α_i denotes the payments value through the channel γ_i. Let dealer be the trigger party, $\mathcal{S} := \{\gamma_i.\text{sender}\}_{i\in[1,n]}$ and $\mathcal{R} := \{\gamma_i.\text{receiver}\}_{i\in[1,n]}$ the set of all senders and all receivers respectively. $\mathcal{S}$ and $\mathcal{R}$ are known to all parties. A simplified version of the Thora protocol and the used macros are shown below. We refer the reader to Appendix B.5 for a full description of the protocol. The main phases of the protocol are as follows.

Initialization. First, we make sure that all parties are aware of every channel who is participating in the update. The protocol then starts from the *Pre-Setup* phase. The protocol execution is triggered by a party denoted by dealer. Note that the triggering party has no security or privacy advantages over the others.

Pre-Setup. Each user γ_i.receiver creates tx_i^{in}, which has an output conditioned on the public keys of γ_i.receiver and all senders in $\mathcal{S}$. The value of the output is $n \cdot \varepsilon$, where ε is the smallest possible amount of cash. tx_i^{in} is created by calling the procedure `GenTxIn`. Then, γ_i.receiver calls `GenTxEp`, which takes tx_i^{in} and $\mathcal{R}$ as inputs, and returns a transaction tx_i^{ep} with outputs to all users in $\mathcal{R}$, each containing ε coins. γ_i.receiver sends tx_i^{ep} to all users. The structure of tx_i^{in} and tx_i^{ep} can be viewed in Figure 4.

Setup. γ_i.sender, upon receiving $\{\text{tx}_j^{\text{ep}}\}_{j\in[1,n]}$ from all receivers, verifies the correctness of these transactions. Then, γ_i.sender creates $\text{tx}_i^{\text{state}}$, tx_i^{r}, and $\{\text{tx}_{i,j}^{\text{p}}\}_{j\in[1,n]}$. $\text{tx}_i^{\text{state}}$ splits α_i coins from the sender's current balance in γ.st, which is spendable by payment or refund transactions. tx_i^{r} returns the coins back to γ_i.sender only if the time T elapses. $\text{tx}_{i,j}^{\text{p}}$ has an input from tx_j^{ep} and sends the split coins to γ_i.receiver. The sender creates $\text{tx}_i^{\text{state}}$ by the procedure `GenState`, tx_i^{r} by the procedure `GenRef`, and $\text{tx}_{i,j}^{\text{p}}$ by the procedure `GenPay`. γ_i.sender sends $\text{tx}_i^{\text{state}}$ and all signed $\text{tx}_{i,j}^{\text{p}}$ to the receiver neighbor. We refer the reader to Figure 3 for the structure of these transactions. γ_i.receiver checks the correctness of the transactions and signatures, then sends the first endorsement to all parties.

Confirmation. When a sender γ_i.sender gets first endorsements from all parties in $\mathcal{R}$, it updates γ_i using $\text{tx}_i^{\text{state}}$. If the update is performed successfully, γ_i.sender sends a signature on each tx_j^{ep} to the receiver γ_j.receiver. Each receiver γ_i.receiver waits for all signatures on tx_i^{ep} and then sends the second endorsement to all parties if γ_i has been updated successfully.

Finalizing. Upon receiving the second endorsements from all parties in $\mathcal{R}$, a sender can safely update the channel to its final state with the receiver neighbor. When updating a channel fails in this phase, and no tx^{ep} is on the ledger, the receiver can post her tx^{ep} and force the payment.

Respond. This phase is executed in every round by all users. Each sender γ_i.sender checks whether the current round is greater than T, γ_i has been closed, and at least one tx^{ep} is on the ledger. If so, γ_i.sender posts tx_i^{r} to the ledger before γ_i.receiver force the payment by posting a payment transaction. On the other side, each receiver γ_i.receiver checks whether one tx_j^{ep} has appeared on the ledger. If so, she closes the channel γ_i. After the appearance of $\text{tx}_i^{\text{state}}$ on the ledger, she posts $\text{tx}_{i,j}^{\text{p}}$ to the ledger and force the payment through the channel γ_i.

The Thora multi-channel updates protocol

- Let dealer be a selected user as the trigger party, T the upper bound on the time we expect the updates to be performed, and Δ the blockchain delay.
- Let $U := \{(\gamma_i, \alpha_i)\}_{i\in[1,n]}$ be the set of all ongoing updates. Each α_i is known only for parties in γ_i.users.

Initialization

dealer

(1) Send message (`init`, $\{\gamma_i\}_{i\in[1,n]}$) to all parties in $\{\gamma_i.\text{sender}\}_{i\in[1,n]} \cup \{\gamma_i.\text{receiver}\}_{i\in[1,n]}$.

All parties upon receiving (`init`, $\{\gamma_i\}_{i\in[1,n]}$) from dealer

(1) Verify the channels set. If decision is not participating in the protocol, return `abort`.
(2) Set $\mathcal{S} := \{\gamma_i.\text{sender}\}_{i\in[1,n]}$, $\mathcal{R} := \{\gamma_i.\text{receiver}\}_{i\in[1,n]}$, and $\mathcal{P} := \mathcal{S} \cup \mathcal{R}$.
(3) Go to the *Pre-Setup* phase.

Pre-Setup

γ_i.receiver

(1) Set tx_i^{in} := `GenTxIn`(γ_i.receiver, $\{\gamma_k\}_{k\in[1,n]}$).
(2) Set tx_i^{ep} := `GenTxEp`($\{\gamma_k\}_{k\in[1,n]}$, tx_i^{in}).
(3) Send tx_i^{ep} to all parties in $\mathcal{R} \cup \mathcal{S}$.

All users upon receiving $\{\text{tx}_j^{\text{ep}}\}_{j\in[1,n]}$ from all parties in $\mathcal{R}$

(1) For all $j \in [1, n]$, If `CheckTxEp`(tx_j^{ep},γ_j.receiver, $\{\gamma_k\}_{k\in[1,n]}$) = $\perp$, return `abort`.
(2) Go to the *Setup* phase.

Setup

γ_i.sender

(1) Set $\text{tx}_i^{\text{state}}$ = `GenState`(α_i, T, $\overline{\gamma}_i$).
(2) Set tx_i^{r} = `GenRef`($\text{tx}_i^{\text{state}}$, γ_i.sender).
(3) For all $j \in [1, n]$, let $\theta_{i,j}$ be the output of tx_j^{ep} which corresponds to γ_i.receiver, then create $\text{tx}_{i,j}^{\text{p}}$:= `GenPay`($\text{tx}_i^{\text{state}}$, γ_i.receiver, $\theta_{i,j}$) and the corresponding signature $\sigma_{\gamma_i.\text{sender}}(\text{tx}_{i,j}^{\text{p}})$.
(4) Send ($\text{tx}_i^{\text{state}}$, $\{\text{tx}_{i,j}^{\text{p}}, \sigma_{\gamma_i.\text{sender}}(\text{tx}_{i,j}^{\text{p}}))\}_{j\in[1,n]}$) to γ_i.receiver.

γ_i.receiver upon receiving

($\text{tx}_i^{\text{state}}$, $\{(\text{tx}_{i,j}^{\text{p}}, \sigma_{\gamma_i.\text{sender}}(\text{tx}_{i,j}^{\text{p}}))\}_{j\in[1,n]}$) from γ_i.sender

(1) If $\text{tx}_i^{\text{state}} \neq$ `GenState`(α_i, T, $\overline{\gamma}_i$), return `abort`.
(2) If any signature $\sigma_{\gamma_i.\text{sender}}(\text{tx}_{i,j}^{\text{p}})$ is not correct, return `abort`.
(3) For all $j \in [1, n]$, let $\theta_{i,j}$ be the output of tx_j^{ep} owned by γ_i.receiver. if $\text{tx}_{i,j}^{\text{p}} \neq$ `GenPay`($\text{tx}_i^{\text{state}}$, γ_i.receiver, $\theta_{i,j}$), return `abort`.
(4) Send message (`setup-ok`$_i$) to all parties in $\mathcal{P}$.

```
All users upon receiving { (setup-ok$_j$ }$_{j\in[1,n]}$)
from all parties in $\mathcal{R}$
(1) Go to the *Confirmation* phase.
                    Confirmation
$\gamma_i$.sender
(1) updateChannel($\overline{\gamma}_i$, tx$_i^{\text{state}}$).
(2) If time $t_u$ has expired and the message (UPDATE-OK) has not been
    returned, return abort.
(3) For all $j \in [1,n]$, send $\sigma(\text{tx}_j^{\text{ep}})$ to $\gamma_j$.receiver.
$\gamma_i$.receiver upon receiving $\{\sigma(\text{tx}_i^{\text{ep}})\}_{j\in[1,n]}$ from all parties in $\mathcal{S}$
(1) If (UPDATE-OK) has been returned and for all $j \in [1,n]$, $\sigma(\text{tx}_j^{\text{ep}})$
    is a valid signatures, send message (confirmation-ok$_i$) to all
    parties in $\mathcal{P}$, otherwise return abort.
All users upon receiving { (confirmation-ok$_j$ }$_{j\in[1,n]}$)
from all parties in $\mathcal{R}$
(1) Go to the *Finalizing* phase.
                    Finalizing
$\gamma_i$.sender
(1) Set tx$_i^{\text{trans}}$ = GenTrans($\alpha_i$, $\overline{\gamma}_i$).
(2) updateChannel($\overline{\gamma}_i$, tx$_i^{\text{trans}}$).
$\gamma_i$.receiver
(1) If the message (UPDATE-OK) has not been received for the final
    transfer, and no tx$^{\text{ep}}$ is on the ledger, before time $T - t_c - 3\Delta$, combine
    received signatures from senders for tx$_i^{\text{ep}}$ with own signature inside
    $\sigma(\text{tx}_i^{\text{ep}})$ and calls publishTx(tx$_i^{\text{ep}}$, $\sigma(\text{tx}_i^{\text{ep}})$).
              Respond(Executed in every round $\tau_x$)
$\gamma_i$.receiver
(1) If $\tau_x < T - t_c - 2\Delta$ and at least one tx$^{\text{ep}}$ is on-chain, closeChannel($\overline{\gamma}_i$).
(2) After tx$_i^{\text{state}}$ is accepted on the blockchain within at most $t_c$ rounds,
    wait $\Delta$ rounds. Let $\sigma(\text{tx}_i^{\text{p}})$ be a signature using the secret key
    $sk_{\gamma_i.\text{receiver}}$ in addition to received signature from $\gamma_i$.sender for
    tx$_i^{\text{p}}$. publishTx(tx$_i^{\text{p}}$, $\sigma(\text{tx}_i^{\text{p}})$).
$\gamma_i$.sender
(1) If $\tau_x > T$, $\overline{\gamma}_i$ is closed and tx$_i^{\text{state}}$ and at least one tx$^{\text{ep}}$ is on the ledger,
    but not tx$_i^{\text{p}}$, publishTx(tx$_i^{\text{r}}$, $\sigma_{\gamma_i.\text{sender}}(\text{tx}_i^{\text{r}})$).
```

Subprocedures used in the multi-channel updates protocol

```
GenTxIn($R$, $\{\gamma_k\}_{k\in[1,n]}$):
(1) $n := |\{\gamma_k\}_{k\in[1,n]}|$
(2) $\phi$ := MultiSig($R$, $\gamma_1$.sender, $\gamma_2$.sender, ..., $\gamma_n$.sender).
(3) Return tx$^{\text{in}}$ := splitCoins($R$, $n \cdot \varepsilon$, $\phi$).
GenTxEp($\{\gamma_k\}_{k\in[1,n]}$, tx$^{\text{in}}$):
(1) $n := |\{\gamma_k\}_{k\in[1,n]}|$
(2) If tx$^{\text{in}}$.output[0].cash $\leq n \cdot \varepsilon$, return $\bot$.
(3) outputList := $\emptyset$.
(4) For each $R_i := \gamma_i$.receiver for all $i \in [1,n]$:
    • outputList = outputList $\cup$ ($\varepsilon$, OneSig($R_i$) $\wedge$ RelTime($t_c + \Delta$))
(5) $id := \mathcal{H}$(tx$^{\text{in}}$.output[0], outputList).
(6) Return tx$^{\text{ep}}$ := ($id$, tx$^{\text{in}}$.output[0], outputList).
CheckTxEp(tx$^{\text{ep}}$, $R$, $\{\gamma_k\}_{k\in[1,n]}$):
(1) $n := |\{\gamma_k\}_{k\in[1,n]}|$
(2) If tx$^{\text{ep}}$.input.cash $\leq n \cdot \varepsilon$ or
    tx$^{\text{ep}}$.input.$\phi \neq$ MultiSig($R$, $\gamma_1$.sender, $\gamma_2$.sender, ..., $\gamma_n$.sender),
    return $\bot$.
(3) If |tx$^{\text{ep}}$.output| $\neq n$, return $\bot$.
(4) For all outputs (cash, $\phi$) $\in$ tx$^{\text{ep}}$.output if cash $\neq \varepsilon$ or
    $\phi \neq$ (OneSig($x$), RelTime($t_c + \Delta$)), where $x$ is one of the receivers,
    return $\bot$.
(5) Return $\top$.
GenState($\alpha$, $T$, $\overline{\gamma}$):
(1) Let $\theta' := \overline{\gamma}$.st be the current state of channel $\overline{\gamma}$ and contains two
    outputs $\theta'_s = (x_s$, OneSig($\overline{\gamma}$.sender)) and
    $\theta'_r = (x_r$, OneSig($\overline{\gamma}$.receiver)).
(2) If $x_s < \alpha$ return $\bot$.
(3) Return $\theta := (\theta_0, \theta_1, \theta_2)$ such that:
    • $\theta_0 := (\alpha$, (OneSig($\overline{\gamma}$.sender) $\wedge$ AbsTime($T$))$\vee$
      (MultiSig($\overline{\gamma}$.sender, $\overline{\gamma}$.receiver) $\wedge$ RelTime($t_c + \Delta$)))
    • $\theta_1 := (x_s - \alpha$, OneSig($\overline{\gamma}$.sender))
    • $\theta_2 := (x_r$, OneSig($\overline{\gamma}$.receiver))
GenRef(tx$^{\text{state}}$, $\gamma_i$.sender):
(1) Return a transaction tx$^{\text{r}}$ such that tx$^{\text{r}}$.input := tx$^{\text{state}}$.output[0] and
    tx$^{\text{r}}$.output := (tx$^{\text{state}}$.output[0].cash, OneSig($\gamma_i$.sender).
GenPay(tx$^{\text{state}}$, $\gamma$.receiver, $\theta$):
(1) Return a transaction tx$^{\text{p}}$ such that tx$^{\text{p}}$.input := (tx$^{\text{state}}$.output[0], $\theta$)
    and
    tx$^{\text{p}}$.output := (tx$^{\text{state}}$.output[0].cash + $\theta$.cash, OneSig($\gamma$.receiver).
GenTrans($\alpha$, $\overline{\gamma}$):
(1) Let $\theta' := \overline{\gamma}$.st = $(\theta'_0, \theta'_1, \theta'_2)$ be the current state of channel $\overline{\gamma}$.
(2) Return $\theta := (\theta_0, \theta_1)$ such that:
    • $\theta_0 := (\theta'_1$.cash, OneSig($\overline{\gamma}$.sender))
    • $\theta_1 := (\theta'_2$.cash + $\alpha$, OneSig($\overline{\gamma}$.receiver))
```

5 SECURITY ANALYSIS

5.1 Security model

We model the security of our multi-channel updates protocol in the synchronous setting and global universal composability (GUC) framework [15]. Our security model is similar to the one adopted in prior work [7, 9, 17]. In particular, the global ledger $\mathcal{L}$ is modeled by the functionality $\mathcal{G}_{ledger}$, which is parameterized by a signature scheme Σ and a blockchain delay Δ. We model the notion of communication by the ideal functionality $\mathcal{F}_{GDC}$ and the time by $\mathcal{G}_{clock}$. Moreover, we define an ideal functionality $\mathcal{G}_{channel}$, which provides *open*, *update*, and *close* operations for payment channels.

The formal security analysis is detailed in Appendix B. In this section, we briefly present a high-level overview of the security model. First, we provide an ideal functionality $\mathcal{F}_{update}$, which describes an ideal multi-channel update protocol with atomicity and strong value privacy properties. $\mathcal{F}_{update}$ is parameterized by a blockchain delay Δ and a time T, which determine an upper bound on the expected time for a successful Thora payment. The ideal functionality describes input/output behaviors of the payment protocol users, and their impacts on the global ledger.

We then describe the Thora protocol Π formally, and show that Π GUC-realizes $\mathcal{F}_{update}$. Intuitively, this means that we design a simulator $\mathcal{X}$, which translates any attack on the protocol Π on the ideal functionality $\mathcal{F}_{update}$. We then show that no PPT environment can distinguish between interacting with the real world and interacting with the ideal world. Thus, Π provides both atomicity and strong value privacy. This is stated by Theorem 1 and formally proven in Appendix B.

Theorem 1. *For any $\Delta, T \in \mathbb{N}$, the protocol Π GUC-realizes the ideal functionality $\mathcal{F}_{update}$.*

5.2 High level functionality description

We give a high level description of our channel update ideal functionality $\mathcal{F}_{update}$ and refer to Appendix B for the formal UC description. $\mathcal{F}_{update}$ can be called for a set of channels to be updated, essentially with the goal of atomically performing payments in each channel from sender to receiver. Similar to the protocol, the ideal functionality proceeds in the following phases.

In the *initialization* phase, the set of channels to be updated is registered with $\mathcal{F}_{update}$. This phase is initiated by a dealer, which can be any party that is part of the set of channels to be updated. Following this, in phase *pre-setup*, $\mathcal{F}_{update}$ prepares all channels for update by creating a synchronizing transaction tx^{ep} per channel that can later be used to force all payments. In phase *setup*, $\mathcal{F}_{update}$ proceeds with preparing an intermediary state update for each channel. In this intermediary state, the payment can be enforced if any of the synchronizing transactions gets posted to $\mathcal{G}_{ledger}$ and reverted after timeout T. Then, in phase *confirmation*, the updates to the intermediary states are performed via $\mathcal{G}_{channel}$.

The functionality $\mathcal{F}_{update}$ proceeds to the *finalizing* phase iff all updates are successful and either the set of senders are honest or the simulator provided a valid signature from all dishonest senders for the synchronizing transactions. This is crucial, because at this point $\mathcal{F}_{update}$ can enforce the payment for honest receivers and only then it is safe to start finalizing. In the *finalizing* phase, all channels are finalized, i.e., updated to the state where the payment went through. If an update fails, $\mathcal{F}_{update}$ can utilize the synchronizing transaction to ensure that the payment is forced for honest receivers.

Further, the functionality checks each round if a synchronizing transaction tx^{ep} was posted to $\mathcal{G}_{ledger}$. This can be achieved by expecting the environment to pass the execution token to $\mathcal{F}_{update}$ each round. If it does not, $\mathcal{F}_{update}$ outputs an error the next time it gets the execution token. In case that a synchronizing transaction is posted, $\mathcal{F}_{update}$ can force the payment on $\mathcal{G}_{ledger}$. Similarly, a refund can be forced after T.

5.3 Informal security analysis

Here, we informally argue why the Thora protocol description shown in Section 4.2 achieves atomicity and strong value privacy as defined in Section 3.1.

Atomicity. We want to show that if there exist two channels with different update statuses, where each has at least one honest user, then the party deviating from the protocol loses the payment value in favour of the other (honest) channel end-point.

Assume that for two channels γ_i, γ_j, each with at least one honest user and with payment values α_i and α_j, γ_i is updated successfully, but γ_j is reverted. There are two possible cases as follows.

(1) The final update in γ_i is done by γ_i.sender using $\text{tx}_i^{\text{trans}}$. If γ_i.sender has followed the protocol correctly, she should receive `confirmation-ok` message from all receivers, including γ_j.receiver. So, γ_j.receiver has enough signatures to put tx_j^{ep} on the ledger and force the payment. If γ_i.sender has finalized γ_i without receiving all `confirmation-ok` messages, she is deviating from the protocol at the cost of losing her funds to γ_i.receiver. Also, if γ_j.receiver has sent `confirmation-ok` without having enough signatures or refuses to force the payment using tx_j^{ep}, she is deviating from the protocol at the cost of losing her funds to γ_j.sender. None of the cases would affect others' security.

(2) The payment in γ_i is forced via posting an enable payment transaction tx_k^{ep} and $\text{tx}_{i,k}^{\text{p}}$ on the ledger. Thus, all other receivers, including γ_j.receiver, can force the payment in their channels using tx_k^{ep}. Note that tx_k^{ep} contains an output owned by γ_j.receiver, otherwise this user would not send `setup-ok` to other parties, including γ_i.sender. If γ_i.sender continued the protocol without receiving all `setup-ok` messages, she is deviating from protocol at the cost of losing her funds. Also, if γ_j.receiver has sent `setup-ok` having incorrect tx_k^{ep} or refuses to force the payment using tx_k^{ep}, she is deviating from the protocol at the cost of losing her funds to γ_j.sender. None of the cases would affect others' security.

Strong value privacy. For an optimistic execution of the protocol, the value of payment α_i through each channel γ_i is only known to the sender and the receiver of this channel. α_i is used only in $\text{tx}_i^{\text{state}}$, tx_i^{r}, and $\{\text{tx}_{i,j}^{\text{p}}\}_{j\in[1,n]}$. These transactions are exchanged between γ_i.sender and γ_i.receiver through secure and authenticated channels. If both parties are honest, the payment value is not visible to an adversary.

6 EVALUATION

In this section, we analyze the performance of our construction. We conducted an asymptotic analysis to determine the number of transactions required on-chain and off-chain. We also built an implementation to evaluate the size of these transactions and to check the compatibility of the construction with Bitcoin's scripting functionalities. The implementation is open-source and the code is publicly available [6]. Let n be the number of payment channels to be updated, which means that there are n possibly non-distinct senders and n possibly non-distinct receivers, and $m \in [0, n]$ be the number of channels in which parties do not agree to update off-chain, and therefore on-chain transactions are required to settle the dispute.

Number and size of transactions. In the honest case, Thora happens completely off-chain, requiring no on-chain cost. The (worst-case) on-chain overhead of the scheme is linear, requiring $2m + 1$ transactions to be posted on-chain. As shown in Table 2 and discussed below, this in line with the state-of-the-art Bitcoin-compatible PCN protocols (e.g., Lightning Network and Blitz). In Thora, however, users are required to store a linear number of off-chain transactions per channel (which results in a quadratic number of total off-chain transactions), whereas the off-chain overhead for the existing Bitcoin-compatible PCN protocols is only constant per channel (or linear in total). We argue that this is a reasonable price to pay for supporting a larger class of off-chain applications, as (i) this increase does not lead to any extra on-chain fees and (ii) the size is small enough in practice to be easily handled even on mobile devices, as we show now.

The transaction tx^{ep} is $141n + 160$ bytes large, since it requires an output and a signature for each channel. Making use of Taproot's aggregated Schnorr signatures [2], one can reduce the size of this transaction to $38n + 256$ bytes. This is achieved by eliminating n public keys (32 bytes) and signatures (70-72 bytes) from the redeem script in $\mathsf{tx}^{\mathsf{ep}}$, adding instead one Schnorr public key (32 bytes), which is the aggregation of public keys of one receiver and n senders, and one Schnorr signature (64 bytes).

Moreover, each channel requires n transactions tx^{p} (501 bytes each), one transaction tx^{r} (272 bytes), an input transaction to $\mathsf{tx}^{\mathsf{ep}}$ (224 bytes), a channel update of size 380 bytes for initiating the update, and another one of size 337 bytes for finalizing the update. For the whole protocol execution, this leads to an off-chain storage overhead of $539n + 1469$ bytes per channel as we plot in Figure 6. For example, even when updating $n = 100$ channels, the off-chain transaction overhead is only around 55KB per channel, or around 5.5MB are exchanged in total.

Collateral. Because the success of the update depends on the global event $\mathsf{tx}^{\mathsf{ep}}$, Thora manages a constant collateral lock time. For the payment protocols LN [29] and AMHL [24], this collateral is instead linear in the number of channels, as they require a growing timelock for each channel to propagate the preimage required for unlocking. In PT [19], the time is logarithmic due to the underlying tree-based structure. Finally, Blitz [9], Sprites [27], and AMCU [18] achieve also constant collateral, at the price of various security, expressiveness, and compatibility trade-offs (cf. Tables 1 and 2).

Computational overhead. Computationally, the protocol needs to create and verify transactions (mostly string operations) and handle signatures. In particular, the computational overhead is dominated by computing and verifying signatures. Each sender needs to sign up to $2n+2$ transactions, more specifically the channel update transaction $\mathsf{tx}^{\mathsf{state}}$, one force refund transaction tx^{r} which they need only in case of dispute, n force payment transactions tx^{p} for their receiver neighbors, and n transactions $\mathsf{tx}^{\mathsf{ep}}$, one for each receiver. Each receiver signs up to $n+2$ transactions, i.e., the channel update transaction $\mathsf{tx}^{\mathsf{state}}$, one force payment transaction tx^{p} which they need only in case of dispute, and their own transaction $\mathsf{tx}^{\mathsf{ep}}$. In our implementation, the time required for creating and verifying one signature is about 30ms on average.

On-chain comparison with LN and Blitz. In Table 3, we compare the on-chain costs of Thora with LN and Blitz, the two state-of-the-art solutions for path-based payments. We assume that Thora is used to conduct such a payment and focus on the on-chain load on the blockchain together with the associated fees, which we calculate using the current price of Bitcoin in USD [4] and the current average fee per bytes [5] (February 2022). When all parties are honest, both protocols are executed completely off-chain, and no transaction is required to appear on the ledger, thus here we are interested in the case where parties need to force either the payment or the refund.

Thora and Blitz have similar message costs, just the cost for the payment and refund transactions are inverted, which corresponds to the fact that one adopts the pay-unless-revoke paradigm and the other one the revoke-unless-pay paradigm. The size of the channel state transaction holding the update contract (370 bytes) is the same in all three constructions, due to our usage of P2SH addresses. The size of the payment transaction in LN is 451 bytes, the size of the refund is 302 bytes. The main difference between the on-chain overhead of these two protocols is $\mathsf{tx}^{\mathsf{ep}}$ in Thora. In the case of

Table 2: Asymptotic comparison of current solutions, with n being the number of channels.

	Collateral	# tx (on-chain)	# tx (off-chain)
LN [29]	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
AMHL [24]	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
AMCU [18]	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
PT [19]	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
Blitz [9]	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Sprites [27]	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Thora	$\Theta(1)$	$\Theta(n)$	$\Theta(n^2)$

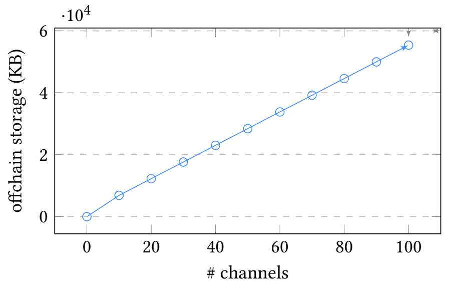

Figure 6: Per-channel off-chain storage overhead for varying number of synchronized channels.

forced payments, in addition to one tx^{p} per channel, also one $\mathsf{tx}^{\mathsf{ep}}$ in total has to be posted to the ledger to enable payments in all channels. This overhead is present in the Blitz refund case. Aside from this, the on-chain fees of Thora are similar to those for LN (the payment transaction is 6% more expensive, while the refund transaction is 6% cheaper). A difference to LN and similarity to Blitz is, that the user posting $\mathsf{tx}^{\mathsf{ep}}$ in Thora (or the equivalent transaction in Blitz) loses $(n-1)\cdot\epsilon$ coins. In Bitcoin, outputs cannot hold 0 coins, therefore ϵ is chosen to be the smallest possible value, e.g., for P2WPKH outputs this is currently 294 satoshis (roughly 0.06 USD). This cost is not present in LN.

7 APPLICATIONS

Most of the existing PCN solutions only support payments from one sender to one receiver and these are to be connected by a path of open channels. This limitation prevents the design of applications with multiple senders or multiple receivers, or those involving payments through two or more distinct PCNs sharing the same blockchain. We show below how Thora overcomes these limitations.

Mass payments. Mass payments can be used by entities that need to perform a high volume of payments. Suppose that a single entity S wants to pay multiple recipients $R_1, R_2, ..., R_n$ simultaneously, with corresponding values $\alpha_1, \alpha_2, ..., \alpha_n$. Here, atomicity can be highly desirable as it guarantees that either all payments are performed correctly or the sender is refunded. For simplicity, we assume that S has a direct channel γ_i to each receiver R_i. The sender S can use Thora with the input of the update set $U := \{(\gamma_1, \alpha_1), (\gamma_2, \alpha_2), ..., (\gamma_n, \alpha_n)\}$ to perform a mass payment in an atomic and off-chain way. Going one step further, the sender does not need to be directly connected to all receivers, but instead can set up updates via some intermediaries. A special case of this is when one sender wants to atomically

Table 3: On-chain overhead and cost comparison of LN, Blitz and Thora. n is the number of channels and $m \in [0, n]$ is the number of disputed channels.

Overhead	LN (Bytes || USD)	Blitz (Bytes || USD)	Thora (Bytes || USD)
Payment transaction	$821m$ || $1.50m$	$642m$ || $1.17m$	$871m$ || $1.59m$
Refund transaction	$672m$ || $1.23m$	$871m$ || $1.59m$	$642m$ || $1.17m$
Cost of enforcing pay/refund	0	$257 + 35n$ || $0.47 + 0.06n$	$256 + 36n$ || $0.47 + 0.06n$

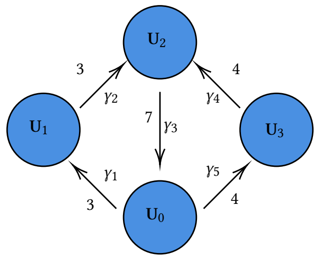

Figure 7: An example of rebalancing with 4 users and 5 channels. Each user holds the same coins after the rebalancing as before, but distribution of coins through channels is changed in order to refund depleted channels. In this case, rebalancing cannot be conducted using a single path-formed payment without using a channel more than once.

pay one receiver over multiple paths at once, e.g., when the balance of one path is not sufficient. This is known as *atomic multi-path payment* [3] and can be achieved with Thora.

Rebalancing. In a bidirectional channel, when payments in one direction are more frequent than in the other direction, the channel becomes skewed and is eventually reduced to a unidirectional channel. Users can close the channel and create a new channel with fresh balances, but for that they need to post some transactions to the blockchain. Alternatively, if there exists a path of channels between the two users that wish to rebalance their channel, they can leverage a payment through this path to replenishing the depleted channel. This can be more efficient if there are multiple users on the path that wish to rebalance their channels. However, as the length of the path grows, refunding becomes more expensive in terms of fees and collateral [18, 21].

Moreover, in some cases, rebalancing is performed through more complex topologies, where (i) a single path payment does not suffice without using certain channels more than once, see Figure 7, or (ii) rebalancing can be made more efficient by making use of the *cancelling out effect*, as shown in [12]. In the example of Figure 7, users hold the same amount of coins after the payments as before, but the distribution of coins in the channels is changed. We can perform rebalancing in this case by initiating Thora with the input of the update set $\{(\gamma_1, 3), (\gamma_2, 3), (\gamma_3, 7), (\gamma_4, 4), (\gamma_5, 4)\}$. The set of senders and receivers are defined based on the direction of the payment in each channel.

Transaction aggregation. Suppose S_0 wants to pay 5 coins to R_1 and S_1 wants to pay 5 coins to S_0, however, there are only channels between S_0 and R_0 and between S_1 and R_1. A more generalized version of this problem was introduced as *transaction aggregation* in [30] along with a construction that uses Thora as a building block, which solves this problem.

Crowdfunding. This application is similar to mass payments, but reversed. We have multiple senders $S_0, S_1, ..., S_n$ who want to fund one single receiver R in an atomic way. In such a case, each sender S_i may want to pay α_i coins to the receiver only when there is a guarantee that all other senders will pay their funds in the same way. Analogous to previous cases, we can use Thora to perform trustless and off-chain crowdfunding by including all involved channels and corresponding payments values in the update set.

8 DISCUSSION

Enhancing privacy. In the case of a dispute when one tx^{ep} appears on the ledger, users can decide to perform honest updates (Section 3) and to post no transaction to the ledger. In this way, they can still preserve the privacy of payment values and save the cost of transaction fees. However, because tx^{ep} includes outputs to all receivers, receivers' identities are revealed publicly when tx^{ep} is posted.

To enhance privacy, we can use stealth addresses [31]. On a high level, instead of existing addresses, receivers can generate fresh addresses for other receivers, and create tx^{ep} using new addresses. Thus, if any tx^{ep} is posted to the ledger and the two channel users decide to update the channel honestly, their identities will stay private from all parties not involved in the protocol. For more details on stealth addresses, we refer the reader to Appendix A.

Accountability. Thora guarantees strong value privacy for off-chain payments. However, in some applications, users may have an interest in accounting payments instead of privacy. For instance, in the crowdfunding application, suppose that all senders have planned to fund the receiver entity with an identical value. Here, the users want to be sure all updates are consistent with the agreed payment value. In this case, the senders can use signed versions of tx^{state} and the set of tx^{p} as receipts and prove their correct behavior.

Communication and computation complexity. As previously discussed, parties have to exchange off-chain messages with each other (i.e., tx^{ep} and signatures), which leads to quadratic communication overhead. By extending the role of dealer to a user whom all parties send these messages and who aggregates the signatures, one could asymptotically reduce the number of signatures that each party has to handle from linear to constant, since only the aggregated signature is sent instead of every individual one. Note that, despite the resulting gain, the size of the transactions is, technically speaking, still quadratic from an asymptotical point of view, because $\mathsf{tx}^{\mathsf{ep}}$ has a linear number of outputs and there is one for every channel.

Race condition. When a receiver posts $\mathsf{tx}^{\mathsf{ep}}$, it will appear in the ledger after at most Δ rounds. According to Section 3, we put a timelock of $t_c + \Delta$ on outputs of a $\mathsf{tx}^{\mathsf{ep}}$ to give enough time to users to close their channels and post tx^{p}. Thus, for a rational receiver, the latest possible time to publish $\mathsf{tx}^{\mathsf{ep}}$ is $T - 3\Delta - t_c$, so that it is accepted at $T - 2\Delta - t_c$ and the timelock of the outputs runs out at $T - \Delta$. This ensures that the payment tx^{p} has precedence over the refund tx^{r}. However, if a receiver posts $\mathsf{tx}^{\mathsf{ep}}$ after $T - 3\Delta - t_c$ and before $T - 2\Delta - t_c$, the timelock on the outputs of $\mathsf{tx}^{\mathsf{ep}}$ could run out just before T, at which point the refunds tx^{r} become possible. Now, there is a potential race between the payments and the refunds. In particular, there is a chance that one receiver can post tx^{p} just before T, and in a another channel, a sender might post a refund.

Of course, this behaviour is irrational since the receiver puts her balance and possibly the one of other malicious receivers at risk, as other channels with honest receivers will have already either updated honestly or posted their $\mathsf{tx}^{\mathsf{ep}}$ before $T - 3\Delta - t_c$. If interested, we can anyway prevent this race condition caused by irrational receivers by changing the spending condition of $\mathsf{tx}^{\mathsf{in}}$. In more detail, each receiver R sets the condition of her $\mathsf{tx}^{\mathsf{in}}$ as follows: $(\mathsf{MultiSig}(R, S_1, S_2, ..., S_n) \wedge \mathsf{RelTime}(\Delta)) \vee (\mathsf{AbsTime}(T - 3\Delta - t_c))$, where S_i is the sender of channel γ_i. According to the new condition, the receiver is forced to post $\mathsf{tx}^{\mathsf{ep}}$ before $T - 5\Delta - t_c$, because otherwise, any party, e.g., also miners, can spend $\mathsf{tx}^{\mathsf{in}}$ and prevent forced payments. This mechanism is similar to the one adopted in Blitz [9].

9 CONCLUSION

In this work, we presented Thora, the first Bitcoin-compatible multi-channel update protocol that guarantees atomicity of payments without restrictions on the channel topology. Moreover, Thora enables channel owners to keep their payment value private.

We defined an ideal functionality to model the security and privacy notations of interest, and showed that Thora is a secure realization thereof within the *Global Universal Composability* framework. Further, we evaluated the performance and showed that the collateral is constant and independent of the number of channels. Our construction does not require Turing-complete smart contracts and can be implemented on top of any blockchain that supports time-locks and signatures in its scripting language.

An interesting direction of future work is exploring the possibility to extend Thora to achieve a threshold atomicity property in generic channel networks. For instance a k-threshold atomicity holds, if at least k channels are updated successfully or else, all channels are reverted to the initial state. This extension can further widen the range of practical applications of Thora payments. Other venues of future research are interoperability, exploring how to refine Thora in order to support atomic channel updates over different blockchains, and optimizing Thora in terms of storage and communication for more specific network topologies.

Acknowledgements

The work was partially supported by CoBloX Labs, by the European Research Council (ERC) under the European Union's Horizon 2020 research (grant agreement 771527-BROWSEC), by the Austrian Science Fund (FWF) through the projects PROFET (grant agreement P31621) and the project W1255-N23, by the Austrian Research Promotion Agency (FFG) through the Bridge-1 project PR4DLT (grant agreement 13808694), the COMET K1 SBA and COMET K1 ABC, by the Vienna Business Agency through the project Vienna Cybersecurity and Privacy Research Center (VISP), by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development and the Christian Doppler Research Association through the Christian Doppler Laboratory Blockchain Technologies for the Internet of Things (CDL-BOT).

REFERENCES

[1] 2017. Raiden network. https://raiden.network/.
[2] 2021. Taproot (BIP 341). https://github.com/bitcoin/bips/blob/master/bip-0341.mediawiki.
[3] 2022. Atomic Multi-path Payments (AMP). https://docs.lightning.engineering/lightning-network-tools/lnd/amp.
[4] 2022. Bitcoin price in USD. https://coinmarketcap.com/.
[5] 2022. Bitcoin transaction fee estimator, average fee per byte. https://privacypros.io/tools/bitcoin-fee-estimator/.
[6] 2022. Thora Payments overhead. https://github.com/Thora-Payments/overhead.
[7] Lukas Aumayr, Oguzhan Ersoy, Andreas Erwig, Sebastian Faust, Kristina Hostakova, Matteo Maffei, Pedro Moreno-Sanchez, and Siavash Riahi. 2021. Generalized Channels from Limited Blockchain Scripts and Adaptor Signatures. In *Advances in Cryptology – ASIACRYPT 2021.*
[8] L. Aumayr, M. Maffei, O. Ersoy, A. Erwig, S. Faust, S. Riahi, K. Hostakova, and P. Moreno-Sanchez. 2021. Bitcoin-Compatible Virtual Channels. In *2021 2021 IEEE Symposium on Security and Privacy (SP).* IEEE Computer Society, Los Alamitos, CA, USA, 901–918. https://doi.org/10.1109/SP40001.2021.00097
[9] Lukas Aumayr, Pedro Moreno-Sanchez, Aniket Kate, and Matteo Maffei. 2021. Blitz: Secure Multi-Hop Payments Without Two-Phase Commits. In *30th USENIX Security Symposium (USENIX Security 21).* USENIX Association, 4043–4060.
[10] Lukas Aumayr, Pedro Moreno-Sanchez, Aniket Kate, and Matteo Maffei. 2021. Donner: UTXO-Based Virtual Channels Across Multiple Hops. Cryptology ePrint Archive, Report 2021/855.
[11] Zeta Avarikioti, Orfeas Stefanos Thyfronitis Litos, and Roger Wattenhofer. 2020. Cerberus Channels: Incentivizing Watchtowers for Bitcoin. Springer International Publishing.
[12] Zeta Avarikioti, Krzysztof Pietrzak, Iosif Salem, Stefan Schmid, Samarth Tiwari, and Michelle Yeo. 2022. HIDE & SEEK: Privacy-Preserving Rebalancing on Payment Channel Networks. In *Financial Cryptography and Data Security.*
[13] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. 2017. Bitcoin as a Transaction Ledger: A Composable Treatment. In *Advances in Cryptology CRYPTO*, Vol. 10401. Springer International Publishing, Cham, 324–356.
[14] R. Canetti. 2001. Universally composable security: a new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science.* 136–145. https://doi.org/10.1109/SFCS.2001.959888
[15] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. 2007. Universally Composable Security with Global Setup. In *Theory of Cryptography TCC*, Vol. 4392. Springer Berlin Heidelberg, Berlin, Heidelberg, 61–85.
[16] Maya Dotan, Saar Tochner, Aviv Zohar, and Yossi Gilad. 2022. Twilight: A Differentially Private Payment Channel Network. https://eprint.iacr.org/2022/136.pdf.
[17] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, Julia Hesse, and Kristina Hostáková. 2019. Multi-party Virtual State Channels. In *Advances in Cryptology – EUROCRYPT 2019.* Springer International Publishing, Cham, 625–656.
[18] Christoph Egger, Pedro Moreno-Sanchez, and Matteo Maffei. 2019. Atomic Multi-Channel Updates with Constant Collateral in Bitcoin-Compatible Payment-Channel Networks. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19).* Association for Computing Machinery, New York, NY, USA, 801–815.
[19] Maxim Jourenko, Mario Larangeira, and Keisuke Tanaka. 2021. Payment Trees: Low Collateral Payments for Payment Channel Networks. In *Financial Cryptography and Data Security1.*
[20] Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. 2013. Universally Composable Synchronous Computation. In *Theory of Cryptography*, Vol. 7785. Springer Berlin Heidelberg, Berlin, Heidelberg, 477–498.
[21] Rami Khalil and Arthur Gervais. 2017. Revive: Rebalancing Off-Blockchain Payment Networks. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17).* Association for

Computing Machinery, New York, NY, USA, 439–453.
[22] Neal Koblitz. 1987. Elliptic curve cryptosystems. *Math. Comp.* 48 (1987), 203–209.
[23] Giulio Malavolta, Pedro Moreno-Sanchez, Aniket Kate, Matteo Maffei, and Srivatsan Ravi. 2017. Concurrency and Privacy with Payment-Channel Networks. Cryptology ePrint Archive, Report 2017/820.
[24] Giulio Malavolta, Pedro Moreno-Sanchez, Clara Schneidewind, Aniket Kate, and Matteo Maffei. 2019. Anonymous Multi-Hop Locks for Blockchain Scalability and Interoperability. In *Network and Distributed System Security Symposium, NDSS*.
[25] Patrick McCorry, Surya Bakshi, Iddo Bentov, Sarah Meiklejohn, and Andrew Miller. 2019. Pisa: Arbitration Outsourcing for State Channels. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*. Association for Computing Machinery.
[26] Patrick Mccorry, Malte Möser, Siamak F. Shahandasti, and Feng Hao. 2016. Towards Bitcoin Payment Networks. Springer-Verlag, Berlin, Heidelberg, 57–76.
[27] Andrew Miller, Iddo Bentov, Surya Bakshi, Ranjit Kumaresan, and Patrick McCorry. 2019. Sprites and State Channels: Payment Networks that Go Faster Than Lightning. In *Financial Cryptography and Data Security*. Springer International Publishing, Cham, 508–526.
[28] Satoshi Nakamoto. 2009. Bitcoin: A Peer-to-Peer Electronic Cash System.
[29] Joseph Poon and Thaddeus Dryja. 2016. The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments.
[30] Samarth Tiwari, Michelle Yeo, Zeta Avarikioti, Iosif Salem, Krzysztof Pietrzak, and Stefan Schmid. 2022. Wiser: Increasing Throughput in Payment Channel Networks with Transaction Aggregation. arXiv. https://arxiv.org/abs/2205.11597
[31] Nicolas Van Saberhagen. 2018. Cryptonote v 2.0 (2013). *URL: https://bytecoin.org/old/whitepaper.pdf. Accessed* (2018), 04–13.

A STEALTH ADDRESSES

The stealth addresses scheme allows us to derive one-time and fresh public keys in a digital signature scheme for a specific user. Here, we briefly describe a basic dual-key stealth addresses protocol (DKSAP). Assume that G is a base point of an elliptic curve, in which the difficulty of the elliptic curve discrete logarithm problem (ECDLP) [22] holds. Moreover, assume that there is a user (say Alice) with two pairs of private/public keys (a, A), (b, B) such that $A = a \cdot G$ and $B = b \cdot G$. We want to derive fresh public keys for Alice. A DKSAP is a tuple of two algorithms DKSAP := (GenPk, GenSk) defined as follows.

- $(P, R) \leftarrow \mathsf{GenPk}(A, B)$: A PPT algorithm takes two Alice's public keys A, B as inputs and returns a fresh public key for Alice P along with an additional value R, which is required for deriving the secret key for P. For that, a random $r \xleftarrow{\$} [0, l-1]$ is sampled uniformly, where l is the prime order of the underlying elliptic curve. Then, P is computed as $P := \mathcal{H}(r \cdot A) \cdot G + B$, $\mathcal{H}$ is a hash function modelled as a random oracle. Moreover, R is computed as $R := r \cdot G$.
- $p \leftarrow \mathsf{GenSk}(a, b, P, R)$: A DPT algorithm takes two Alice's secret keys a, b and P, R generated by GenPk algorithm as inputs and returns the secret key corresponding to P. For that p is computed as $p := \mathcal{H}(a \cdot R) + b$.

Correctness of algorithms follows directly: $p \cdot G = (\mathcal{H}(a \cdot R) + b) \cdot G = \mathcal{H}(a \cdot r \cdot G) \cdot G + b \cdot G = \mathcal{H}(r \cdot A) \cdot G + B = P$. In [31] it is argued that the new address P is unlikable for a spectator, even when observing R.

B UC MODELING

In this section, we formalize our construction in the global UC framework (GUC) [15], which is an extension of the standard UC framework [14] that allows for a global setup. We use this global step for modelling the ledger. Through this section, first, we provide some preliminaries. Then, we define an ideal functionality for the multi-channel updates protocol. Our model follows closely the model in [7, 9, 10].

B.1 Preliminaries, communication model and threat model

In the real world, a protocol Π is executed by a set of parties $\mathcal{P}$ and in the presence of an adversary $\mathcal{A}$. A security parameter $\lambda \in \mathbb{N}$ and an auxiliary input $z \in \{0, 1\}^*$ are given to the adversary as inputs. We consider a static corruption model, which means that $\mathcal{A}$ can corrupt any party $P_i \in \mathcal{P}$ at the beginning of the protocol execution. $\mathcal{A}$ controls corrupted parties and learns their internal states. All parties in $\mathcal{P}$ and $\mathcal{A}$ take their input from a special entity called environment $\mathcal{E}$, which represents everything external to the protocol. This entity observes all output messages from participants. We assume that the communication network is synchronous, and the protocol execution takes place in rounds. The global ideal functionality $\mathcal{G}_{clock}$ [20] represents a global clock that proceeds to the next round if all honest parties indicate that they are ready to do so. Every entity always knows the current round. Communications between parties in $\mathcal{P}$ are through authenticated channels with guaranteed delivery after exactly one round. If a party P sends a message to party Q in round t, then Q receives that message in the beginning of round $t+1$ and knows that P has sent the message. We model authenticated channels by an ideal functionality $\mathcal{F}_{GDC}$ [17]. The adversary can read and reorder the messages sent in the same round, but can not modify or delay messages. Communications involving $\mathcal{A}$, $\mathcal{E}$ or the simulator $\mathcal{X}$ and every computation that a party executes locally take zero rounds.

B.2 Ledger and channels

We model a UTXO based blockchain in the ideal functionality $\mathcal{G}_{ledger}$. We denote the blockchain delay as Δ, and the blockchain's signature scheme by Σ. $\mathcal{G}_{ledger}$ communicates with a fixed set of parties $\mathcal{P}$.

Initially, the environment $\mathcal{E}$ chooses a key pair (sk_P, pk_P) for each $P \in \mathcal{P}$ and registers it to the ledger by sending $(\mathtt{sid}, \mathtt{register}, pk_P)$ to $\mathcal{G}_{ledger}$. Also, $\mathcal{E}$ sets the initial state of $\mathcal{L}$, which is a publicly accessible set of all published transactions. A party $P \in \mathcal{P}$ can post a transaction $\overline{\mathtt{tx}}$ via message $(\mathtt{sid}, \mathtt{POST}, \overline{\mathtt{tx}})$ to $\mathcal{G}_{ledger}$. The transaction will be added to the ledger after at most Δ rounds, if it is valid. The exact number of delay rounds is chosen by the adversary. In this work, we consider a simplified model for the underlying blockchain and assume that the set of users is fixed instead of allowing them to join or leave dynamically. For a more precise model, we refer the reader to works [13]. We define an ideal functionality $\mathcal{G}_{channel}$ [8], which is built on top of $\mathcal{G}_{ledger}$ and provides *open*, *update*, and *close* procedures related to payment channels. We assume that closing a channel takes at most t_c rounds and updating a channel takes at most t_u rounds. For simplicity, we assume that channels involved in the multi-channel updates protocol have already been registered and opened with the ledger functionality.

The complete API of $\mathcal{G}_{channel}$ and $\mathcal{G}_{ledger}$ are shown below. We hide the calls to $\mathcal{G}_{clock}$ and $\mathcal{F}_{GDC}$ in our notation. Instead of explicitly calling these functionalities, we write msg $\overset{t}{\hookrightarrow} X$ to denote sending message msg to party X in round t and also msg $\overset{t}{\hookleftarrow} X$ to denote receiving message msg from party X in round t.

Interface of $\mathcal{G}_{ledger}(\Delta, \Sigma)$ [7, 9]

This functionality keeps a record of the public keys of parties. Also, all transactions that are posted (and accepted, see below) are stored in the publicly accessible set $\mathcal{L}$ containing tuples of all accepted transactions .

Parameters:

Δ: upper bound on the number of rounds it takes a valid transaction to be published on $\mathcal{L}$

Σ: a digital signature scheme

API:

Messages from $\mathcal{E}$ via a dummy user $P \in \mathcal{P}$:

- $(\mathtt{sid}, \mathtt{REGISTER}, \mathrm{pk}_P) \overset{\tau}{\hookleftarrow} P$:
  This function adds an entry (pk_P, P) to PKI consisting of the public key pk_P and the user P, if it does not already exist.
- $(\mathtt{sid}, \mathtt{POST}, \overline{\mathrm{tx}}) \overset{\tau}{\hookleftarrow} P$:
  This function checks if $\overline{\mathrm{tx}}$ is a valid transaction and if yes, accepts it on $\mathcal{L}$ after at most Δ rounds.

Interface of $\mathcal{G}_{channel}(T, k)$ [7, 9]

Parameters:

T: upper bound on the maximum number of consecutive off-chain communication rounds between channel users

k: number of ways the channel state can be published on the ledger

API:

Messages from $\mathcal{E}$ via a dummy user P:

- $(\mathtt{sid}, \mathtt{CREATE}, \overline{\gamma}, \mathrm{tid}_P) \overset{\tau}{\hookleftarrow} P$:
  Let $\overline{\gamma}$ be the attribute tuple $(\overline{\gamma}.\mathrm{id}, \overline{\gamma}.\mathrm{users}, \overline{\gamma}.\mathrm{cash}, \overline{\gamma}.\mathrm{st})$, where $\gamma.\mathrm{id} \in \{0,1\}^*$ is the identifier of the channel, $\overline{\gamma}.\mathrm{users} \subset \mathcal{P}$ are the users of the channel (and $P \in \overline{\gamma}.\mathrm{users}$), $\overline{\gamma}.\mathrm{cash} \in \mathbb{R}^{\geq 0}$ is the total money in the channel and $\overline{\gamma}.\mathrm{st}$ is the initial state of the channel. tid_P defines P's input for the funding transaction of the channel. When invoked, this function asks $\overline{\gamma}$.otherParty to create a new channel.
- $(\mathtt{sid}, \mathtt{UPDATE}, \mathrm{id}, \vec{\theta}) \overset{\tau}{\hookleftarrow} P$:
  Let $\overline{\gamma}$ be the channel where $\overline{\gamma}.\mathrm{id} = \mathrm{id}$. When invoked by $P \in \overline{\gamma}.\mathrm{users}$ and both parties agree, the channel $\overline{\gamma}$ (if it exists) is updated to the new state $\vec{\theta}$. If the parties disagree or at least one party is dishonest, the update can fail or the channel can be forcefully closed to either the old or the new state. Regardless of the outcome, we say that t_u is the upper bound that an update takes. In the successful case, $(\mathtt{sid}, \mathtt{UPDATED}, \mathrm{id}, \vec{\theta}) \xhookrightarrow{\leq \tau + t_u} \overline{\gamma}.\mathrm{users}$ is output.
- $(\mathtt{sid}, \mathtt{CLOSE}, \mathrm{id}) \overset{\tau}{\hookleftarrow} P$:
  Will close the channel $\overline{\gamma}$, where $\overline{\gamma}.\mathrm{id} = \mathrm{id}$, either peacefully or forcefully. After at most t_c in round $\leq \tau + t_c$, a transaction tx with the current state $\overline{\gamma}.\mathrm{st}$ as output (tx.output := $\overline{\gamma}.\mathrm{st}$) appears on $\mathcal{L}$ (the public ledger of $\mathcal{G}_{ledger}$).

B.3 The UC-security definition

Closely following [9, 10], we define Π as a *hybrid* protocol that accesses to ideal functionality $\mathcal{F}_{prelim}$ consisting of $\mathcal{F}_{GDC}$, $\mathcal{G}_{ledger}$, $\mathcal{G}_{channel}$, and $\mathcal{G}_{clock}$. In the beginning, the environment $\mathcal{E}$ supplies inputs to the parties in $\mathcal{P}$ and the adversary $\mathcal{A}$ with a security parameter λ and auxiliary input z. We denote the output that $\mathcal{E}$ observes as the ensemble $\mathsf{EXEC}^{\mathcal{F}_{prelim}}_{\Pi,\mathcal{A},\mathcal{E}}(\lambda, z)$. $\breve{}_{\mathcal{F}_{update}}$ denotes the ideal protocol of the ideal functionality $\mathcal{F}_{update}$, where the dummy users simply forward their input to $\mathcal{F}_{update}$. With access to functionalities $\mathcal{F}_{prelim}$, we denote the output of this idealized protocol as $\mathsf{EXEC}^{\mathcal{F}_{prelim}}_{\breve{}_{\mathcal{F}_{update}},\mathcal{X},\mathcal{E}}(\lambda, z)$.

If a protocol Π GUC-realizes an ideal functionality $\mathcal{F}_{update}$, then any attack that is possible on the real world protocol Π can be carried out against the ideal protocol $\breve{}_{\mathcal{F}_{update}}$ and vice versa.

Definition 1. A protocol Π GUC-realizes an ideal functionality $\mathcal{F}_{update}$, w.r.t. $\mathcal{F}_{prelim}$, if for every adversary $\mathcal{A}$ there exists a simulator $\mathcal{X}$ such that for any $z \in \{0,1\}^*$ and $\lambda \in \mathbb{N}$, we have

$$\mathsf{EXEC}^{\mathcal{F}_{prelim}}_{\Pi,\mathcal{A},\mathcal{E}}(\lambda, z) \approx_c \mathsf{EXEC}^{\mathcal{F}_{prelim}}_{\breve{}_{\mathcal{F}_{update}},\mathcal{X},\mathcal{E}}(\lambda, z) \quad (1)$$

where $\approx_c$ denotes computational indistinguishability.

B.4 Ideal functionality

Here, we define our the ideal functionality $\mathcal{F}_{update}$. This functionality can output an ERROR message, e.g., when a transaction does not appear on the ledger as it should. When $\mathcal{F}_{update}$ outputs ERROR, any guarantees are lost. Hence, we are only interested in protocols that realize $\mathcal{F}_{update}$ and never output an ERROR. The subprocedures used in $\mathcal{F}_{update}$, Π, and $\mathcal{X}$ follow the same logic as the macros defined in Section 4.2.

Note that in $\mathcal{F}_{update}$ and Π, for better readability, we use the set $\mathcal{P}$ to store all parties, the set $\mathcal{S}$ to store all senders, and the set $\mathcal{R}$ to store all receivers. We know that two different channels may have a common user. Thus, for handling duplicated identifiers in the aforementioned sets, we implicitly assign different identifiers for users of different channels. Consequently, the size of each set is equal to the number of channels.

Ideal Functionality $\mathcal{F}_{update}(\Delta, T)$

Parameters:

Δ : Upper bound on the time it takes a transaction to appear on $\mathcal{L}$.

T : Upper bound on the time expected for successful payments.

Local variables:

idSet : A set of tuples containing pairs of ids and channels $(\mathtt{pid}, \gamma_i)$ to avoid duplicated channels.

Γ : A set of tuples $(\mathtt{pid}, \overline{\gamma}_i, \mathrm{tx}^{\mathrm{state}}_i, \mathrm{tx}^{\mathrm{r}}_i, \{\mathrm{tx}^{\mathrm{p}}_{i,j}, \theta_{i,j}\}_{j\in[1,n]})$ that for each payment id pid and channel $\overline{\gamma}_i$, store the state transaction $\mathrm{tx}^{\mathrm{state}}_i$, refund transaction $\mathrm{tx}^{\mathrm{r}}_i$ and a set of tuples for payment transactions $(\mathrm{tx}^{\mathrm{p}}_{i,j}, \theta_{i,j})$ where $\theta_{i,j}$ is the output of $\mathrm{tx}^{\mathrm{ep}}_j$ used in $\mathrm{tx}^{\mathrm{p}}_{i,j}$.

Ψ : A map, storing for a given pid a copy of all $\mathrm{tx}^{\mathrm{ep}}$ in a set $\{\mathrm{tx}^{\mathrm{ep}}_j\}_{j\in[1,n]}$.

t_u : Time required to perform a ledger channel update honestly.

t_c : Time it takes at most to close a channel.

Start (executed in the beginning in round t_{start})

Send $(\mathtt{sid}, \mathtt{start}) \xhookrightarrow{t_{\mathrm{start}}} \mathcal{X}$ and upon $(\mathtt{sid}, \mathtt{start\text{-}ok}, t_u, t_c) \xhookleftarrow{t_{\mathrm{start}}} \mathcal{X}$ set t_u and t_c accordingly.

Initialization

Let τ be the current round, and $\mathcal{S}$, $\mathcal{R}$, and $\mathcal{P}$ be initially empty sets.

(1) If $(\mathtt{sid}, \mathtt{pid}, \mathtt{CHANNELS\text{-}SET}, \{\gamma_i\}_{i\in[1,n]}) \overset{\tau}{\hookleftarrow}$ dealer where the dealer is honest, do the following.

(a) Send $(\texttt{sid}, \texttt{pid}, \texttt{send-init}, \{\gamma_j\}_{j\in[1,n]}, \text{dealer}) \overset{\tau}{\hookrightarrow} \mathcal{X}$.
(b) For all honest $P_i \in \{\gamma_i.\text{sender}\}_{i\in[1,n]} \cup \{\gamma_i.\text{receiver}\}_{i\in[1,n]}$, send $(\texttt{sid}, \texttt{pid}, \texttt{INIT-CHECK}, \{\gamma_j\}_{j\in[1,n]}) \xrightarrow{\tau+1} P_i$.

(2) Upon each message $(\texttt{sid}, \texttt{pid}, \texttt{send-check}, \{\gamma_i\}_{i\in[1,n]}, P_i) \xleftrightarrow{\tau+1} \mathcal{X}$, send $(\texttt{sid}, \texttt{pid}, \texttt{INIT-CHECK}, \{\gamma_j\}_{j\in[1,n]}) \xrightarrow{\tau+1} P_i$.
(3) Upon $(\texttt{sid}, \texttt{pid}, \texttt{INIT-CHECKED}, \{\gamma_j\}_{j\in[1,n]}) \xleftrightarrow{\tau+1} P_i$ for each honest P_i, do following.
(a) Send $(\texttt{sid}, \texttt{pid}, \texttt{send-init-ok}, \{\gamma_j\}_{j\in[1,n]}, P_i) \xrightarrow{\tau+1} \mathcal{X}$.
(b) If this is the first INIT-CHECKED message from an honest party, for each γ_i the tuple $(\texttt{pid}, \gamma_i) \notin \text{idSet}$, set $\text{idSet} = \text{idSet} \cup \{(\texttt{pid}, \gamma_i)\}$, add γ_i.sender to $\mathcal{S}$ and $\mathcal{P}$, and add γ_i.receiver to $\mathcal{R}$ and $\mathcal{P}$.
(4) If there is an honest $P_i \in \mathcal{P}$, where the message $(\texttt{sid}, \texttt{pid}, \texttt{INIT-CHECKED}, \{\gamma_j\}_{j\in[1,n]}) \xleftrightarrow{\tau+1} P_i$ is not received, go idle.
(5) If there is an honest $P_i \in \mathcal{P}$ and a corrupted $P_j \in \mathcal{P}$, where the message $(\texttt{sid}, \texttt{pid}, \texttt{init-acc}, P_i, P_j) \xleftrightarrow{\tau+2} \mathcal{X}$ is not received, remove P_i from $\mathcal{P}$ and $\mathcal{S}$ or $\mathcal{R}$.
(6) Go to the *Pre-Setup* phase, and pass the set of channels with the receiver in $\mathcal{P}$ to the next phase.

<u>Pre-Setup</u>

Let τ be the current round.
(1) For each channels γ_i do following.
(a) Let $\text{tx}_i^{\text{in}} := \texttt{GenTxIn}(\gamma_i.\text{receiver}, \{\gamma_k\}_{k\in[1,n]})$.
(b) Let $\text{tx}_i^{\text{ep}} := \texttt{GenTxEp}(\{\gamma_k\}_{k\in[1,n]}, \text{tx}_i^{\text{in}})$, and add tx_i^{ep} to $\Psi(\texttt{pid})$
(c) If γ_i.receiver is corrupted, send $(\texttt{sid}, \texttt{pid}, \texttt{presetup-req}, \gamma_i, \text{tx}_i^{\text{ep}}) \overset{\tau}{\hookrightarrow} \mathcal{X}$.
(d) Else if γ_i.receiver is honest, for all corrupted $P_j \in \mathcal{P}$ send $(\texttt{sid}, \texttt{pid}, \texttt{send-presetup}, \text{tx}_i^{\text{ep}}, \gamma_i.\text{receiver}, P_j) \overset{\tau}{\hookrightarrow} \mathcal{X}$.
(2) If there is an honest $P_i \in \mathcal{P}$ and a corrupted $P_j \in \mathcal{R}$, where the message $(\texttt{sid}, \texttt{pid}, \texttt{presetup-acc}, P_i, P_j) \xleftrightarrow{\tau+1} \mathcal{X}$ is not received, remove P_i from $\mathcal{P}$ and $\mathcal{S}$ or $\mathcal{R}$.
(3) Go to the *Setup* phase, and pass the set of channels with at least one user in $\mathcal{P}$ to the next phase.

<u>Setup</u>

Let τ be the current round.
(1) For each channel γ_i if both γ_i.sender and γ_i.receiver are honest, do the following.
(a) If $\gamma_i.\text{sender} \in \mathcal{P}$, $(\texttt{sid}, \texttt{pid}, \texttt{REQ-VALUE}, \gamma_i) \overset{\tau}{\hookrightarrow} \gamma_i.\text{sender}$.
(b) Upon $(\texttt{sid}, \texttt{pid}, \texttt{VALUE}, \overline{\gamma}_i, \alpha_i) \overset{\tau}{\hookleftarrow} \gamma_i.\text{sender}$, continue. Otherwise skip the steps (c) to (g).
(c) Let $\text{tx}_i^{\text{state}} := \texttt{GenState}(\alpha_i, T, \overline{\gamma}_i)$, and $\text{tx}_i^{\text{r}} := \texttt{GenRef}(\text{tx}_i^{\text{state}}, \gamma_i.\text{sender})$.
(d) For all $j \in [1, n]$, let $\theta_{i,j}$ be the output of tx_j^{ep} which corresponds to γ_i.receiver, then create $\text{tx}_{i,j}^{\text{p}} = \texttt{GenPay}(\text{tx}_i^{\text{state}}, \gamma_i.\text{receiver}, \theta_{i,j})$.
(e) If $\gamma_i.\text{receiver} \in \mathcal{P}$, send $(\texttt{sid}, \texttt{pid}, \texttt{REQ-VALUE}, \gamma_i) \xrightarrow{\tau+1} \gamma_i.\text{receiver}$.
(f) Upon $(\texttt{sid}, \texttt{pid}, \texttt{VALUE}, \overline{\gamma}_i, \alpha_i) \xleftrightarrow{\tau+1} \gamma_i.\text{receiver}$, continue. Otherwise skip the step (g).
(g) For all corrupted $P_j \in \mathcal{P}$, send $(\texttt{sid}, \texttt{pid}, \texttt{send-setup-ok}, \gamma_i.\text{receiver}, P_j) \xrightarrow{\tau+1} \mathcal{X}$.
(2) Else If γ_i.sender is corrupted and γ_i.receiver is honest, do the following.
(a) If $(\texttt{sid}, \texttt{pid}, \texttt{setup-acc}, \overline{\gamma}_i, \text{tx}_i^{\text{state}}, \{\text{tx}_{i,j}^{\text{p}}\}\}_{j\in[1,n]}) \xleftrightarrow{\tau+1} \mathcal{X}$, set $\alpha_i := \text{tx}_i^{\text{state}}.\text{output}[0].\text{cash}$. Otherwise, skip the steps (b) to (d).
(b) If $\gamma_i.\text{receiver} \in \mathcal{P}$, send $(\texttt{sid}, \texttt{pid}, \texttt{REQ-VALUE}, \gamma_i) \xrightarrow{\tau+1} \gamma_i.\text{receiver}$.
(c) Upon $(\texttt{sid}, \texttt{pid}, \texttt{VALUE}, \overline{\gamma}_i, \alpha_i) \xleftrightarrow{\tau+1} \gamma_i$.receiver with a same α_i as the step(b) and $\text{tx}_i^{\text{state}} = \texttt{GenState}(\alpha_i, T, \overline{\gamma}_i)$, continue. Otherwise skip the step (e).
(d) For all corrupted $P_j \in \mathcal{P}$, send $(\texttt{sid}, \texttt{pid}, \texttt{send-setup-ok}, P_i, P_j) \xrightarrow{\tau+1} \mathcal{X}$.
(3) Else If γ_i.sender is honest and γ_i.receiver is corrupted, do the following.
(a) If $\gamma_i.\text{sender} \in \mathcal{P}$, $(\texttt{sid}, \texttt{pid}, \texttt{REQ-VALUE}, \gamma_i) \overset{\tau}{\hookrightarrow} \gamma_i.\text{sender}$.
(b) Upon $(\texttt{sid}, \texttt{pid}, \texttt{VALUE}, \overline{\gamma}_i, \alpha_i) \overset{\tau}{\hookleftarrow} \gamma_i.\text{sender}$, continue. Otherwise skip the steps (c) to (e).
(c) Let $\text{tx}_i^{\text{state}} := \texttt{GenState}(\alpha_i, T, \overline{\gamma}_i)$, and $\text{tx}_i^{\text{r}} := \texttt{GenRef}(\text{tx}_i^{\text{state}}, \gamma_i.\text{sender})$.
(d) For all $j \in [1, n]$, let $\theta_{i,j}$ be the output of tx_j^{ep} which corresponds to γ_i.receiver, then create $\text{tx}_{i,j}^{\text{p}} = \texttt{GenPay}(\text{tx}_i^{\text{state}}, \gamma_i.\text{receiver}, \theta_{i,j})$.
(e) Send $(\texttt{sid}, \texttt{pid}, \texttt{send-setup}, \overline{\gamma}_i, \text{tx}_i^{\text{state}}, \{(\text{tx}_{i,j}^{\text{p}}, \sigma_{\gamma_i.\text{sender}}(\text{tx}_{i,j}^{\text{p}}))\}_{j\in[1,n]}) \xrightarrow{\tau+1} \mathcal{X}$.
(4) If there is an honest receiver $P_i \in \mathcal{R}$, where the message $(\texttt{sid}, \texttt{pid}, \texttt{VALUE}, \overline{\gamma}_i, \alpha_i) \xleftrightarrow{\tau+1} P_i$ is not received, go idle.
(5) If there is an honest $P_i \in \mathcal{P}$ and a corrupted $P_j \in \mathcal{R}$, where the message $(\texttt{sid}, \texttt{pid}, \texttt{setup-finalized}, P_i, P_j) \xleftrightarrow{\tau+1} \mathcal{X}$ is not received, remove P_i from $\mathcal{P}$ and $\mathcal{S}$ or $\mathcal{R}$.
(6) Go to the *Confirmation* phase. Pass the set of channels with at least one user in $\mathcal{P}$ to the next phase.

<u>Confirmation</u>

- Let τ be the current round.
(1) For each honest sender $\gamma_i.\text{sender} \in \mathcal{S}$, do the following.
(a) Send $(\texttt{ssid}_C, \texttt{UPDATE}, \overline{\gamma}_i.\text{id}, \text{tx}_i^{\text{state}}.\text{output}) \overset{\tau}{\hookrightarrow} \mathcal{G}_{channel}$.
(b) If not $(\texttt{ssid}_C, \texttt{UPDATED}, \overline{\gamma}_i.\text{id}, \text{tx}_i^{\text{state}}.\text{output}) \xleftrightarrow{\tau+t_u} \mathcal{G}_{channel}$, skip the step (c).
(c) For each corrupted $\gamma_j.\text{receiver} \in \mathcal{R}$, send $(\texttt{sid}, \texttt{pid}, \texttt{send-sig}, \gamma_i.\text{sender}, \gamma_j.\text{receiver}, \text{tx}_j^{\text{ep}}) \xrightarrow{\tau+t_u} \mathcal{X}$.
(2) For each honest receiver $\gamma_i.\text{receiver} \in \mathcal{R}$, if
(i) $(\texttt{sid}, \texttt{pid}, \texttt{confirmation-acc}, \gamma_i.\text{receiver}, \gamma_j.\text{sender}) \xleftrightarrow{\tau+t_u+1} \mathcal{X}$ is received for all corrupted $\gamma_j.\text{sender} \in \mathcal{S}$, and
(ii) $(\texttt{ssid}_C, \texttt{UPDATED}, \overline{\gamma}_i.\text{id}, \text{tx}_i^{\text{state}}.\text{output}) \xleftrightarrow{\tau+t_u} \mathcal{G}_{channel}$ on behalf of γ_i.receiver, do the following.
(a) Send $(\texttt{sid}, \texttt{pid}, \texttt{OPENED}, \overline{\gamma}_i) \xrightarrow{\tau+t_u+1} \gamma_i$.receiver
(b) For all corrupted $P_j \in \mathcal{P}$, $(\texttt{sid}, \texttt{pid}, \texttt{send-confirmation-ok}, \gamma_i.\text{receiver}, P_j) \xrightarrow{\tau+t_u} \mathcal{X}$.
(3) If there is an honest receiver γ_i.receiver, where $(\texttt{sid}, \texttt{pid}, \texttt{confirmation-acc}, \gamma_i.\text{receiver}, \gamma_j.\text{sender}) \xleftrightarrow{\tau+t_u+1} \mathcal{X}$ is not received for at least one corrupted $\gamma_j.\text{sender} \in \mathcal{S}$, or $(\texttt{ssid}_C, \texttt{UPDATED}, \overline{\gamma}_i.\text{id}, \text{tx}_i^{\text{state}}.\text{output}) \xleftrightarrow{\tau+t_u} \mathcal{G}_{channel}$ on behalf of γ_i.receiver, go idle.
(4) If there is an honest $P_i \in \mathcal{P}$ and a corrupted $P_j \in \mathcal{R}$, where the message $(\texttt{sid}, \texttt{pid}, \texttt{confirmation-finalized}, P_i, P_j) \xleftrightarrow{\tau+t_u+1} \mathcal{X}$ is not received, remove P_i from $\mathcal{P}$ and $\mathcal{S}$ or $\mathcal{R}$.
(5) Send $(\texttt{sid}, \texttt{pid}, \texttt{agg-sig}, \{\text{tx}_j^{\text{ep}}\}_{j\in[1,n]}, \mathcal{S}) \xrightarrow{\tau+t_u+1} \mathcal{X}$.
(6) Go to the *Finalizing* phase. Pass the set of channels with at least one user in $\mathcal{P}$ to the next phase.

<u>Finalizing</u>

- Let τ be the current round.
(1) For each channel γ_i, let $\text{tx}_i^{\text{trans}} := \texttt{GenTrans}(\alpha_i, \overline{\gamma}_i)$.

(2) For each honest sender γ_i.sender, send $(\mathtt{ssid}_C, \mathtt{UPDATE}, \gamma_i.\mathrm{id}, \mathrm{tx}_i^{\mathrm{trans}}.\mathrm{output}) \overset{\tau}{\hookrightarrow} \mathcal{G}_{channel}$.

(3) For each channels γ_i, If γ_i.receiver is honest, do the following.

(a) If not $(\mathtt{ssid}_C, \mathtt{UPDATED}, \overline{\gamma}_i.\mathrm{id}, \mathrm{tx}_i^{\mathrm{trans}}.\mathrm{output}) \xleftarrow{\tau+t_u} \mathcal{G}_{channel}$, $(\mathtt{sid}, \mathtt{pid}, \mathtt{post\text{-}txep}, \overline{\gamma}_i, \mathrm{tx}_i^{\mathrm{ep}}) \xhookrightarrow{\tau+t_u} \mathcal{X}$.

(b) Send $(\mathtt{sid}, \mathtt{pid}, \mathtt{FINALIZED}, \overline{\gamma}_i) \xhookrightarrow{\tau+t_u} \gamma_i$.receiver.

<u>Respond (executed at the end of every round)</u>

Let t be the starting round. For every element $(\mathtt{pid}, \overline{\gamma}_i, \mathrm{tx}_i^{\mathrm{state}}, \mathrm{tx}_i^{\mathrm{r}}, \{\mathrm{tx}_{i,j}^{\mathrm{p}}, \theta_{i,j}\}_{j\in[1,n]}) \in \Gamma$, if $\overline{\gamma}_i.\mathrm{st} = \mathrm{tx}_i^{\mathrm{state}}.\mathrm{output}$, and one $\mathrm{tx}_j^{\mathrm{ep}} \in \Psi(\mathtt{pid})$ is on $\mathcal{L}$, do the Pay step as follows.

Pay: If γ_i.receiver is honest and $t < T - t_c - 2\Delta$ do the following.

(1) $(\mathtt{ssid}_C, \mathtt{CLOSE}, \overline{\gamma_i}.\mathrm{id}) \overset{t}{\hookrightarrow} \mathcal{G}_{channel}$

(2) At time $t + t_c$, if a transaction tx with tx.output $= \overline{\gamma}_i$.st appears on $\mathcal{L}$, Wait for Δ rounds and send $(\mathtt{sid}, \mathtt{pid}, \mathtt{post\text{-}pay}, \overline{\gamma}_i, \mathrm{tx}_{i,j}^{\mathrm{p}}) \xhookrightarrow{t' < T - \Delta} \mathcal{X}$.

(3) At time $t'' < T$, if a transaction tx′ appears on $\mathcal{L}$ with tx′.input $= [\theta_{i,j}, \mathrm{tx.output}[0]]$ and tx′.output $= [(\mathrm{tx.output}[0].\mathrm{cash} + \theta_{i,j}.\mathrm{cash}, \mathrm{OneSig}(\gamma_i.\mathrm{receiver}))]$, send $(\mathtt{sid}, \mathtt{pid}, \mathtt{PAID}) \overset{t''}{\hookrightarrow} \gamma_i$.receiver. Otherwise return ERROR to all parties.

Force-Refund: Else, if a transaction tx with tx.output $= \overline{\gamma}_i$.st is on-chain and tx.output[0] is unspent, $t \geq T$, and γ_i.sender is honest, do the following.

(1) Send $(\mathtt{sid}, \mathtt{pid}, \mathtt{post\text{-}refund}, \overline{\gamma}_i, \mathrm{tx}_i^{\mathrm{r}}) \overset{t}{\hookrightarrow} \mathcal{X}$

(2) If transaction tx′ with tx′.input $= [\mathrm{tx.output}[0]]$ and tx′.output $= (\mathrm{tx.output}[0].\mathrm{cash}, \mathrm{OneSig}(\gamma_i.\mathrm{sender}))$ appears on the $\mathcal{L}$ in round $t_1 < t + \Delta$, send $(\mathtt{sid}, \mathtt{pid}, \mathtt{FORCE\text{-}REFUND}) \overset{t_1}{\hookrightarrow} \gamma_i$.sender. Otherwise, return ERROR to all parties.

B.5 Protocol

In this section, we present the formal protocol Π. The protocol is similar to what is presented in Figure 5, but extended with payment ids and UC formalism. We add the environment $\mathcal{E}$ and model communication in rounds. The protocol is divided into six phases. In *Initialization*, a user dealer receives the ongoing updates from $\mathcal{E}$ and sends them to every user to check whether all participants agree with that. In *Pre-Setup*, each receiver generates $\mathrm{tx}^{\mathrm{ep}}$ and sends it to all parties. In *Setup*, senders generate and send $\mathrm{tx}^{\mathrm{state}}$, tx^{p}, and tx^{r} to their neighbors. Receivers verify the messages and inform all parties when everything is OK. In *Confirmation*, senders update their channels, and then send their signature to each $\mathrm{tx}^{\mathrm{ep}}$ to the corresponding receivers. When a receiver gets all signatures, sends an endorsement to all parties. In *Finalizing*, the senders after receiving all endorsements update their channel to the final state. If a receiver does not get UPDATED from $\mathcal{G}_{channel}$, puts $\mathrm{tx}^{\mathrm{ep}}$ on-chain. In *Respond* users will react to $\mathrm{tx}^{\mathrm{ep}}$ being published and, either force payments or refunds.

Protocol Π

Local variables:

pidSet :	A set storing every payment id pid that a user has participated in, to prevent duplicates.
paySet :	A map storing for a given pid a tuple $(\{\gamma_i\}_{i\in[1,n]}, \mathcal{S}, \mathcal{R})$ where U is the set of containing channels and payment values, $\mathcal{S}$ is the set of all senders and $\mathcal{R}$ is the set of all receivers.
local :	A map storing for a given pid a copy of all $\mathrm{tx}^{\mathrm{ep}}$ in a set $\{\mathrm{tx}_j^{\mathrm{ep}}\}_{j\in[1,n]}$.
left :	For each sender γ_i.sender, a map storing for a given pid a tuple $(\overline{\gamma_i}, \mathrm{tx}_i^{\mathrm{state}}, \mathrm{tx}_i^{\mathrm{r}})$ which contains the channel $\overline{\gamma}_i$ and corresponding state and refund transactions.
right :	For each receiver γ_i.receiver, a map storing for a given pid a tuple $(\overline{\gamma_i}, \mathrm{tx}_i^{\mathrm{state}}, \{(\mathrm{tx}_{i,j}^{\mathrm{p}}, \sigma_{\gamma_i.\mathrm{sender}}(\mathrm{tx}_{i,j}^{\mathrm{p}}), \theta_{i,j})\}_{j\in[1,n]})$ which contains a channel and corresponding state transaction and the set of payment transactions. Along with each $\mathrm{tx}_{i,j}^{\mathrm{p}}$, a signature from the sender of the channel and the input of this transaction that comes from $\mathrm{tx}_j^{\mathrm{ep}}$ are saved.
sigSet :	For each receiver γ_i.receiver, a map, storing for a given pid the signatures for $\mathrm{tx}_i^{\mathrm{ep}}$ of all senders $\{\sigma_{\gamma_i.\mathrm{sender}}(\mathrm{tx}_i^{\mathrm{ep}})\}_{j\in[1,n]}$.

<u>Initialization</u>

- Let τ be the current round.

<u>dealer upon $(\mathtt{sid}, \mathtt{pid}, \mathtt{CHANNELS\text{-}SET}, \{\gamma_i\}_{i\in[1,n]}) \overset{\tau}{\hookleftarrow} \mathcal{E}$</u>

(1) For all parties P_i in $\{\gamma_i.\mathrm{sender}\}_{i\in[1,n]} \cup \{\gamma_i.\mathrm{receiver}\}_{i\in[1,n]}$, send $(\mathtt{sid}, \mathtt{pid}, \mathtt{init}, \{\gamma_i\}_{i\in[1,n]}) \overset{\tau}{\hookrightarrow} P_i$.

<u>Each γ_i.sender and γ_i.receiver</u>

<u>upon $(\mathtt{sid}, \mathtt{pid}, \mathtt{init}, \{\gamma_j\}_{j\in[1,n]}) \xhookleftarrow{\tau+1}$ dealer</u>

(1) If pid ∈ pidSet, abort. Add pid to pidSet, and let $\mathcal{S}$, $\mathcal{R}$ and $\mathcal{P}$ be initially empty sets.

(2) Send $(\mathtt{sid}, \mathtt{pid}, \mathtt{INIT\text{-}CHECK}, \{\gamma_j\}_{j\in[1,n]}) \xhookrightarrow{\tau+1} \mathcal{E}$.

(3) If $(\mathtt{sid}, \mathtt{pid}, \mathtt{INIT\text{-}CHECKED}, \{\gamma_j\}_{j\in[1,n]}) \xhookleftarrow{\tau+1} \mathcal{E}$, for each channel γ_j add γ_j.sender to $\mathcal{S}$ and γ_j.receiver to $\mathcal{R}$. Then set $\mathrm{paySet}(\mathtt{pid}) := (\{\gamma_j\}_{j\in[1,n]}, \mathcal{S}, \mathcal{R})$ and $\mathcal{P} := \mathcal{R} \cup \mathcal{S}$. Otherwise abort.

(4) Send $(\mathtt{sid}, \mathtt{pid}, \mathtt{init\text{-}ok}) \xhookrightarrow{\tau+1} P_i$ to all $P_i \in \mathcal{P}$.

(5) If $(\mathtt{sid}, \mathtt{pid}, \mathtt{init\text{-}ok}) \xhookleftarrow{\tau+2} P_i$ from all parties in $\mathcal{P}$, go to the *Pre-Setup* phase. Otherwise abort.

<u>Pre-Setup</u>

- Let τ be the current round.

<u>γ_i.receiver</u>

(1) Let $\mathrm{tx}_i^{\mathrm{in}} := \mathtt{GenTxIn}(\gamma_i.\mathrm{receiver}, \{\gamma_k\}_{k\in[1,n]})$.

(2) Let $\mathrm{tx}_i^{\mathrm{ep}} := \mathtt{GenTxEp}(\{\gamma_k\}_{k\in[1,n]}, \mathrm{tx}_i^{\mathrm{in}})$.

(3) Send $(\mathtt{sid}, \mathtt{pid}, \mathtt{pre\text{-}setup}, \mathrm{tx}_i^{\mathrm{ep}}) \overset{\tau}{\hookrightarrow} P_i$ for all $P_i \in \mathcal{P}$.

<u>All users upon</u>

<u>$(\mathtt{sid}, \mathtt{pid}, \mathtt{pre\text{-}setup}, \mathrm{tx}_i^{\mathrm{ep}}) \xhookleftarrow{\tau+1} \gamma_i$.receiver for all $i \in [1, n]$</u>

(1) For all $j \in [1, n]$, if $\texttt{CheckTxEp}(\mathsf{tx}^{\mathsf{ep}}_j, \gamma_j.\text{receiver}, \{\gamma_k\}_{k\in[1,n]}) = \bot$, abort. otherwise set $\text{local}(\texttt{pid}) = \{\mathsf{tx}^{\mathsf{ep}}_j\}_{j\in[1,n]}$ and go to the *Setup* phase.

Setup

- Let τ be the current round.

γ_i.sender

(1) Send $(\texttt{sid}, \texttt{pid}, \texttt{REQ-VALUE}, \gamma_i) \overset{\tau}{\hookrightarrow} \mathcal{E}$. If this message is replied by $(\texttt{sid}, \texttt{pid}, \texttt{VALUE}, \overline{\gamma}_i, \alpha_i) \overset{\tau}{\hookleftarrow} \mathcal{E}$, continue. Otherwise go idle.
(2) Let $\mathsf{tx}^{\mathsf{state}}_i := \texttt{GenState}(\alpha_i, T, \overline{\gamma}_i)$.
(3) Let $\mathsf{tx}^{\mathsf{r}}_i := \texttt{GenRef}(\mathsf{tx}^{\mathsf{state}}_i, \gamma_i.\text{sender})$.
(4) For all $j \in [1, n]$, let $\theta_{i,j}$ be the output of $\mathsf{tx}^{\mathsf{ep}}_j$ which corresponds to γ_i.receiver, then create $\mathsf{tx}^{\mathsf{p}}_{i,j} := \texttt{GenPay}(\mathsf{tx}^{\mathsf{state}}_i, \gamma_i.\text{receiver}, \theta_{i,j})$.
(5) Set $\text{left}(\texttt{pid}) := (\overline{\gamma}_i, \mathsf{tx}^{\mathsf{state}}_i, \mathsf{tx}^{\mathsf{r}}_i, \{\mathsf{tx}^{\mathsf{p}}_{i,j}\}_{j\in[1,n]})$.
(6) Generate the set $\{\sigma_{\gamma_i.\text{sender}}(\mathsf{tx}^{\mathsf{p}}_{i,j})\}_{j\in[1,n]}$.
(7) Send $(\texttt{sid}, \texttt{pid}, \texttt{setup}, \overline{\gamma}_i, \mathsf{tx}^{\mathsf{state}}_i, \{(\mathsf{tx}^{\mathsf{p}}_{i,j}, \sigma_{\gamma_i.\text{sender}}(\mathsf{tx}^{\mathsf{p}}_{i,j}))\}_{j\in[1,n]}) \overset{\tau}{\hookrightarrow} \gamma_i$.receiver.

γ_i.receiver upon $(\texttt{sid}, \texttt{pid}, \texttt{setup}, \overline{\gamma}_i, \mathsf{tx}^{\mathsf{state}}_i, \{(\mathsf{tx}^{\mathsf{p}}_{i,j}, \sigma_{\gamma_i.\text{sender}}(\mathsf{tx}^{\mathsf{p}}_{i,j}))\}_{j\in[1,n]}) \xleftarrow{\tau+1} \gamma_i$.sender

(1) Send $(\texttt{sid}, \texttt{pid}, \texttt{REQ-VALUE}, \gamma_i) \xhookrightarrow{\tau+1} \mathcal{E}$. If this message is replied by $(\texttt{sid}, \texttt{pid}, \texttt{VALUE}, \overline{\gamma}_i, \alpha_i) \xhookleftarrow{\tau+1} \mathcal{E}$, continue. Otherwise go idle.
(2) If $\mathsf{tx}^{\mathsf{state}}_i \neq \texttt{GenState}(\alpha_i, T, \overline{\gamma}_i)$, abort.
(3) For each element in $\{(\mathsf{tx}^{\mathsf{p}}_{i,j}, \sigma_{\gamma_i.\text{sender}}(\mathsf{tx}^{\mathsf{p}}_{i,j}))\}_{j\in[1,n]}$, If $\sigma_{\gamma_i.\text{sender}}(\mathsf{tx}^{\mathsf{p}}_{i,j})$ is not a correct signature, abort.
(4) For all $j \in [1, n]$, let $\theta_{i,j}$ be the output of $\mathsf{tx}^{\mathsf{ep}}_j$ which corresponds to γ_i.receiver. If $\mathsf{tx}^{\mathsf{p}}_{i,j} \neq \texttt{GenPay}(\mathsf{tx}^{\mathsf{state}}_i, \gamma_i.\text{receiver}, \theta_{i,j})$, abort.
(5) Set $\text{right}(\texttt{pid}) = (\overline{\gamma_i}, \mathsf{tx}^{\mathsf{state}}_i, \{\mathsf{tx}^{\mathsf{p}}_{i,j}, \sigma_{\overline{\gamma}_i.\text{sender}}(\mathsf{tx}^{\mathsf{p}}_{i,j}, \theta_{i,j})\}_{j\in[1,n]})$
(6) Send $(\texttt{sid}, \texttt{pid}, \texttt{setup-ok}) \xhookrightarrow{\tau+1} P_i$ for all $P_i \in \mathcal{P}$.

All users

(1) If $(\texttt{sid}, \texttt{pid}, \texttt{setup-ok}) \xhookleftarrow{\tau+2} P_i$ for all $P_i \in \mathcal{R}$, go to the *Confirmation* phase. Otherwise abort.

Confirmation

- Let τ be the current round.

γ_i.sender

(1) Send $(\texttt{ssid}_C, \texttt{UPDATE}, \overline{\gamma}_i.\text{id}, \mathsf{tx}^{\mathsf{state}}_i.\text{output}) \overset{\tau}{\hookrightarrow} \mathcal{G}_{channel}$.
(2) If $(\texttt{ssid}_C, \texttt{UPDATED}, \overline{\gamma}_i.\text{id}, \mathsf{tx}^{\mathsf{state}}_i.\text{output}) \xhookleftarrow{\tau+t_u} \mathcal{G}_{channel}$, for all $j \in [1, n]$, create signature $\sigma_{\gamma_i.\text{sender}}(\mathsf{tx}^{\mathsf{ep}}_j)$ and send $(\texttt{sid}, \texttt{pid}, \texttt{confirmation}, \sigma_{\gamma_i.\text{sender}}(\mathsf{tx}^{\mathsf{ep}}_j)) \xhookrightarrow{\tau+t_u} \gamma_j$.receiver.

γ_i.receiver upon $(\texttt{sid}, \texttt{pid}, \texttt{confirmation}, \sigma_{\gamma_j.\text{sender}}(\mathsf{tx}^{\mathsf{ep}}_i)) \xhookleftarrow{\tau+t_u+1} \gamma_j$.sender for all $j \in [1, n]$

(1) If $(\texttt{ssid}_C, \texttt{UPDATED}, \overline{\gamma}_i.\text{id}, \mathsf{tx}^{\mathsf{state}}_i.\text{output}) \xhookleftarrow{\tau+t_u} \mathcal{G}_{channel}$, send $(\texttt{sid}, \texttt{pid}, \texttt{OPENED}, \overline{\gamma}_i) \xhookrightarrow{\tau+t_u+1} \mathcal{E}$. Otherwise abort.
(2) If for all $j \in [1, n]$, $\sigma_{\gamma_j.\text{sender}}(\mathsf{tx}^{\mathsf{ep}}_i)$ are valid signatures, let $\text{sigSet} := \{(\sigma_{\gamma_j.\text{sender}}(\mathsf{tx}^{\mathsf{ep}}_i))\}_{j\in[1,n]}$. Otherwise abort.
(3) Send $(\texttt{sid}, \texttt{pid}, \texttt{confirmation-ok}) \xhookrightarrow{\tau+t_u+1} P_i$ for all $P_i \in \mathcal{P}$.

All users

(1) If $(\texttt{sid}, \texttt{pid}, \texttt{confirmation-ok}) \xhookleftarrow{\tau+t_u+2} P_i$ for all $P_i \in \mathcal{R}$, go to the *Finalizing* phase. Otherwise abort.

Finalizing

- Let τ be the starting round.

γ_i.sender

(1) Let $\mathsf{tx}^{\mathsf{trans}}_i := \texttt{GenTrans}(\alpha_i, \overline{\gamma}_i)$.
(2) Send $(\texttt{ssid}_C, \texttt{UPDATE}, \overline{\gamma}_i.\text{id}, \mathsf{tx}^{\mathsf{trans}}_i.\text{output}) \overset{\tau}{\hookrightarrow} \mathcal{G}_{channel}$.

γ_i.receiver

(1) If not $(\texttt{ssid}_C, \texttt{UPDATED}, \overline{\gamma}_i.\text{id}, \mathsf{tx}^{\mathsf{trans}}_i.\text{output}) \xhookleftarrow{\tau+t_u} \mathcal{G}_{channel}$, sign $\mathsf{tx}^{\mathsf{ep}}_i$ and add the signature to sigSet. $(\texttt{ssid}_L, \texttt{POST}, (\mathsf{tx}^{\mathsf{ep}}_i, \text{sigSet})) \xhookrightarrow{\tau+t_u} \mathcal{G}_{ledger}$.
(2) Send $(\texttt{sid}, \texttt{pid}, \texttt{FINALIZED}, \overline{\gamma}_i) \xhookrightarrow{\tau+t_u} \mathcal{E}$.

Respond

Let t be the current round. Do the following:

γ_i.receiver at the end of every round t

(1) For every pid in right.keyList(), let $(\overline{\gamma_i}, \mathsf{tx}^{\mathsf{state}}_i, \{\mathsf{tx}^{\mathsf{p}}_{i,j}, \sigma_{\gamma_i.\text{sender}}(\mathsf{tx}^{\mathsf{p}}_{i,j}, \theta_{i,j})\}_{j\in[1,n]}) := \text{right}(\texttt{pid})$ and let $\{\mathsf{tx}^{\mathsf{ep}}_j\}_{j\in[1,n]} := \text{local}(\texttt{pid})$.
(2) If $t < T - t_c - 2\Delta$, one $\mathsf{tx}^{\mathsf{ep}}_j$ is on the ledger $\mathcal{L}$, and $\overline{\gamma}_i.\text{st} = \mathsf{tx}^{\mathsf{state}}_i.\text{output}$, do the following:
    (a) Send $(\texttt{ssid}_C, \texttt{CLOSE}, \overline{\gamma_i}.\text{id}) \overset{t}{\hookrightarrow} \mathcal{G}_{channel}$.
    (b) If a transaction tx with $\text{tx.output} = \mathsf{tx}^{\mathsf{state}}_i.\text{output}$ is on $\mathcal{L}$ in round $t + t_c$, wait Δ rounds.
    (c) Sign $\mathsf{tx}^{\mathsf{p}}_{i,j}$ and set $\overline{\mathsf{tx}^{\mathsf{p}}_{i,j}} := (\mathsf{tx}^{\mathsf{p}}_{i,j}, \{\sigma_{\gamma_i.\text{receiver}}(\mathsf{tx}^{\mathsf{p}}_{i,j}), \sigma_{\gamma_i.\text{sender}}(\mathsf{tx}^{\mathsf{p}}_{i,j})\})$.
    (d) Send $(\texttt{ssid}_L, \texttt{POST}, \overline{\mathsf{tx}^{\mathsf{p}}_{i,j}}) \xhookrightarrow{t+t_c+\Delta} \mathcal{G}_{ledger}$.
    (e) When $\mathsf{tx}^{\mathsf{p}}_{i,j}$ appears on $\mathcal{L}$ in round $t_1 < T$, send $(\texttt{sid}, \texttt{pid}, \texttt{PAID}, \overline{\gamma}_i) \overset{t_1}{\hookrightarrow} \mathcal{E}$

γ_i.sender at the end of every round t

(1) For every pid in left.keyList(), let $(\overline{\gamma_i}, \mathsf{tx}^{\mathsf{state}}_i, \mathsf{tx}^{\mathsf{r}}_i, \{\mathsf{tx}^{\mathsf{p}}_{i,j}\}_{j\in[1,n]}) := \text{left}(\texttt{pid})$.
(2) If $t > T$ and a transaction tx with $\text{tx.output} = \mathsf{tx}^{\mathsf{state}}_i$ is on the ledger $\mathcal{L}$, but not any transaction in $\{\mathsf{tx}^{\mathsf{p}}_{i,j}\}_{j\in[1,n]}$, do the following:
    (a) Sign $\mathsf{tx}^{\mathsf{r}}_i$ and set $\overline{\mathsf{tx}^{\mathsf{r}}_i} := (\mathsf{tx}^{\mathsf{r}}_i, \sigma_{\gamma_i.\text{sender}}(\mathsf{tx}^{\mathsf{r}}_i))$.
    (b) Send $(\texttt{ssid}_L, \texttt{POST}, \overline{\mathsf{tx}^{\mathsf{r}}_i}) \overset{t}{\hookrightarrow} \mathcal{G}_{ledger}$.
    (c) When $\mathsf{tx}^{\mathsf{r}}_i$ appears on $\mathcal{L}$ in round $t_1 < t + \Delta$, send $(\texttt{sid}, \texttt{pid}, \texttt{FORCE-REFUND}, \overline{\gamma}_i) \overset{t_1}{\hookrightarrow} \mathcal{E}$

B.6 Proof

In this section, we present the simulator and formal proof that our multi-channel updates protocol Appendix B.5 UC-realizes the ideal functionality $\mathcal{F}_{update}$ Appendix B.4.

Simulator

Local variables:

enableSig :	A map, sorting for a given $(\texttt{pid}, \mathsf{tx}^{\mathsf{ep}}_i)$ the set of signatures $\{\sigma_{\gamma_j.\text{sender}}(\mathsf{tx}^{\mathsf{ep}}_i)\}$ from all senders.
paySig :	A map, sorting for a given $(\texttt{pid}, \mathsf{tx}^{\mathsf{p}}_{i,j})$ the signature $\sigma_{\gamma_i.\text{sender}}(\mathsf{tx}^{\mathsf{p}}_{i,j})$.

Start phase

- Upon $(\mathtt{sid}, \mathtt{start}) \xleftarrow{t_{\text{start}}} \mathcal{F}_{update}$, Send $(\mathtt{sid}, \mathtt{start\text{-}ok}, t_{\text{u}}, t_{\text{c}}) \xrightarrow{t_{\text{start}}} \mathcal{F}_{update}$ and go to the *Initialization* phase.

Initialization phase

- Upon $(\mathtt{sid}, \mathtt{pid}, \mathtt{send\text{-}init}, \{\gamma_j\}_{j\in[1,n]}, \text{dealer}) \overset{\tau}{\hookleftarrow} \mathcal{F}_{update}$, for all corrupted $P_i \in \{\gamma_i.\text{sender}\}_{i\in[1,n]} \cup \{\gamma_i.\text{receiver}\}_{i\in[1,n]}$, send $(\mathtt{sid}, \mathtt{pid}, \mathtt{init}, \{\gamma_i\}_{i\in[1,n]}) \overset{\tau}{\hookrightarrow} P_i$ on behalf of dealer.
- If the trigger party dealer is corrupted, upon $(\mathtt{sid}, \mathtt{pid}, \mathtt{init}, \{\gamma_i\}_{i\in[1,n]}) \overset{\tau}{\hookleftarrow}$ dealer on behalf on each honest party P_i, send $(\mathtt{sid}, \mathtt{pid}, \mathtt{send\text{-}check}, \{\gamma_i\}_{i\in[1,n]}, P_i) \overset{\tau}{\hookrightarrow} \mathcal{F}_{update}$.
- Upon $(\mathtt{sid}, \mathtt{pid}, \mathtt{send\text{-}init\text{-}ok}, \{\gamma_j\}_{j\in[1,n]}, P_i) \overset{\tau}{\hookrightarrow} \mathcal{X}$, for corrupted $P_j \in \{\gamma_i.\text{sender}\}_{i\in[1,n]} \cup \{\gamma_i.\text{receiver}\}_{i\in[1,n]}$, send $(\mathtt{sid}, \mathtt{pid}, \mathtt{init\text{-}ok}) \overset{\tau}{\hookrightarrow} P_j$ on behalf of P_i.
- Upon $(\mathtt{sid}, \mathtt{pid}, \mathtt{init\text{-}ok}) \xleftarrow{\tau+2} P_j$ on behalf of P_i, where P_i is honest and P_j is corrupted, send $(\mathtt{sid}, \mathtt{pid}, \mathtt{init\text{-}acc}, P_i, P_j) \xrightarrow{\tau+2} \mathcal{F}_{update}$.

Pre-Setup phase

- Upon $(\mathtt{sid}, \mathtt{pid}, \mathtt{presetup\text{-}req}, \gamma_i, \text{tx}_x^{\text{ep}}) \overset{\tau}{\hookleftarrow} \mathcal{F}_{update}$, where γ_i.receiver is a corrupted party, do the following.
  1. Upon $(\mathtt{sid}, \mathtt{pid}, \mathtt{pre\text{-}setup}, \text{tx}_i^{\text{ep}}) \xleftarrow{\tau+1} \gamma_j$.receiver of behalf of P_i, where γ_i.receiver is corrupted, and P_i is honest, check if $\text{tx}_i^{\text{ep}} = \text{tx}_x^{\text{ep}}$, $(\mathtt{sid}, \mathtt{pid}, \mathtt{presetup\text{-}acc}, P_i, \gamma_j.\text{receiver}) \xrightarrow{\tau+1} \mathcal{F}_{update}$.
- Upon $(\mathtt{sid}, \mathtt{pid}, \mathtt{send\text{-}presetup}, \text{tx}_i^{\text{ep}}, \gamma_i.\text{receiver}, P_j) \overset{\tau}{\hookleftarrow} \mathcal{F}_{update}$, where γ_i.receiver is honest and P_j is corrupted, send $(\mathtt{sid}, \mathtt{pid}, \mathtt{pre\text{-}setup}, \text{tx}_i^{\text{ep}}) \overset{\tau}{\hookrightarrow} P_j$ on behalf of γ_i.receiver.

Setup phase

- Upon, $(\mathtt{sid}, \mathtt{pid}, \mathtt{send\text{-}setup\text{-}ok}, P_i, P_j) \overset{\tau}{\hookleftarrow} \mathcal{F}_{update}$, where P_i is honest and P_j is corrupted, send $(\mathtt{sid}, \mathtt{pid}, \mathtt{setup\text{-}ok}) \overset{\tau}{\hookrightarrow} P_j$ on behalf of P_i.
- Upon $(\mathtt{sid}, \mathtt{pid}, \mathtt{setup}, \overline{\gamma}_i, \text{tx}_i^{\text{state}}, \{(\text{tx}_{i,j}^{\text{p}}, \sigma_{\gamma_i.\text{sender}}(\text{tx}_{i,j}^{\text{p}}))\}_{j\in[1,n]}) \xleftarrow{\tau+1} \gamma_i$.sender, where γ_i.sender is corrupted, do the following.
  1. Check if any signature $\sigma_{\gamma_i.\text{sender}}(\text{tx}_{i,j}^{\text{p}})$ is not valid, abort.
  2. For all $j \in [1,n]$, let $\theta_{i,j}$ be the output of tx_j^{ep} which corresponds to γ_i.receiver. If $\text{tx}_{i,j}^{\text{p}} \neq \mathtt{GenPay}(\text{tx}_i^{\text{state}}, \gamma_i.\text{receiver}, \theta_{i,j})$, abort.
  3. Add the signature for each $\text{tx}_{i,j}^{\text{p}}$ to $\text{paySig}(\text{pid}, \text{tx}_{i,j}^{\text{p}})$.
  4. $(\mathtt{sid}, \mathtt{pid}, \mathtt{setup\text{-}acc}, \overline{\gamma}_i, \text{tx}_i^{\text{state}}, \{\text{tx}_{i,j}^{\text{p}}\}_{j\in[1,n]}) \xrightarrow{\tau+1} \mathcal{F}_{update}$.
- Upon $(\mathtt{sid}, \mathtt{pid}, \mathtt{send\text{-}setup}, \text{tx}_i^{\text{state}}, \{\text{tx}_{i,j}^{\text{p}}\}_{j\in[1,n]}, \gamma_i) \overset{\tau}{\hookleftarrow} \mathcal{F}_{update}$ where γ_i.sender is honest but γ_i.receiver is corrupted, do the following.
  1. sign $\text{tx}_{i,j}^{\text{p}}$ on behalf of γ_i.sender and add it to $\text{paySig}(\text{pid}, \text{tx}_{i,j}^{\text{p}})$.
  2. send $(\mathtt{sid}, \mathtt{pid}, \mathtt{setup}, \overline{\gamma}_i, \text{tx}_i^{\text{state}}, \{(\text{tx}_{i,j}^{\text{p}}, \sigma_{\gamma_i.\text{sender}}(\text{tx}_{i,j}^{\text{p}}))\}_{j\in[1,n]}) \overset{\tau}{\hookrightarrow} \gamma_i$.receiver on behalf of γ_i.sender.
- Upon $(\mathtt{sid}, \mathtt{pid}, \mathtt{setup\text{-}ok}) \xleftarrow{\tau+1} \gamma_j$.receiver on behalf of P_i, where P_i is honest and γ_j.receiver is corrupted, send $(\mathtt{sid}, \mathtt{pid}, \mathtt{setup\text{-}finalized}, P_i, \gamma_j.\text{receiver}) \xrightarrow{\tau+1} \mathcal{F}_{update}$

Confirmation phase

- Upon $(\mathtt{sid}, \mathtt{pid}, \mathtt{send\text{-}sig}, \gamma_i.\text{sender}, \gamma_j.\text{receiver}, \text{tx}_j^{\text{ep}}) \overset{\tau}{\hookleftarrow} \mathcal{F}_{update}$, where γ_i.sender is honest but γ_j.receiver is corrupted, sign tx_j^{ep} on behalf of γ_i.sender and send $(\mathtt{sid}, \mathtt{pid}, \mathtt{confirmation}, \sigma_{\gamma_i.\text{sender}}(\text{tx}_j^{\text{ep}})) \overset{\tau}{\hookrightarrow} \gamma_j$.receiver.
- Upon $(\mathtt{sid}, \mathtt{pid}, \mathtt{confirmation}, \sigma_{\gamma_j.\text{sender}}(\text{tx}_i^{\text{ep}})) \overset{\tau}{\hookleftarrow} \gamma_j$.sender is received on behalf of γ_i.receiver, where γ_i.receiver is honest and γ_j.sender is corrupted, check if all signatures are valid, send $(\mathtt{sid}, \mathtt{pid}, \mathtt{confirmation\text{-}acc}, \gamma_i.\text{receiver}, \gamma_j.\text{sender}) \overset{\tau}{\hookrightarrow} \mathcal{F}_{update}$.
- Upon, $(\mathtt{sid}, \mathtt{pid}, \mathtt{send\text{-}confirmation\text{-}ok}, P_i, P_j) \overset{\tau}{\hookleftarrow} \mathcal{F}_{update}$, where P_i is honest and P_j is corrupted, $(\mathtt{sid}, \mathtt{pid}, \mathtt{confirmation\text{-}ok}) \xrightarrow{\tau+1} P_j$ on behalf of P_i.
- Upon $(\mathtt{sid}, \mathtt{pid}, \mathtt{confirmation\text{-}ok}) \overset{\tau}{\hookleftarrow} \gamma_j$.receiver is received on behalf of an honest party P_i, where γ_j.receiver is corrupted, send $(\mathtt{sid}, \mathtt{pid}, \mathtt{confirmation\text{-}finalized}, P_i, \gamma_i.\text{receiver}) \overset{\tau}{\hookrightarrow} \mathcal{F}_{update}$.
- Upon $(\mathtt{sid}, \mathtt{pid}, \mathtt{agg\text{-}sig}, \{\text{tx}_j^{\text{ep}}\}_{j\in[1,n]}, \mathcal{S}) \overset{\tau}{\hookleftarrow} \mathcal{X}$, for each tx_j^{ep}, sign the transaction on behalf of all honest $P_i \in \mathcal{S}$ and add $\sigma_{P_i}(\text{tx}_j^{\text{ep}})$ to $\text{enableSig}(\text{pid}, \text{tx}_j^{\text{ep}})$

Finalizing phase

- Upon $(\mathtt{sid}, \mathtt{pid}, \mathtt{post\text{-}txep}, \overline{\gamma}_i, \text{tx}_i^{\text{ep}}) \overset{\tau}{\hookleftarrow} \mathcal{F}_{update}$ where γ_i.receiver is a honest:
  1. Sign tx_i^{ep} on behalf of γ_i.receiver and add the signature to $\text{enableSig}(\text{pid}, \text{tx}_i^{\text{ep}})$
  2. Set $\overline{\text{tx}_i^{\text{ep}}} := (\text{tx}_i^{\text{ep}}, \text{enableSig}(\text{pid}, \text{tx}_i^{\text{ep}}))$.
  3. Send $(\mathtt{ssid}_L, \mathtt{POST}, \overline{\text{tx}_i^{\text{ep}}}) \overset{\tau}{\hookrightarrow} \mathcal{G}_{ledger}$.

Respond phase

- Upon $(\mathtt{sid}, \mathtt{pid}, \mathtt{post\text{-}pay}, \overline{\gamma}_i, \text{tx}_{i,j}^{\text{p}}) \overset{\tau}{\hookleftarrow} \mathcal{F}_{update}$, where γ_i.receiver is honest:
  1. Sign $\text{tx}_{i,j}^{\text{p}}$ on behalf of γ_i.receiver and add the signature to $\text{paySig}(\text{pid}, \text{tx}_{i,j}^{\text{p}})$.
  2. Set $\overline{\text{tx}_{i,j}^{\text{p}}} := (\text{tx}_{i,j}^{\text{p}}, \text{paySig}(\text{pid}, \text{tx}_{i,j}^{\text{p}})$.
  3. Send $(\mathtt{ssid}_L, \mathtt{POST}, \overline{\text{tx}_{i,j}^{\text{p}}}) \xrightarrow{\tau+t_c} \mathcal{G}_{ledger}$.
- Upon $(\mathtt{sid}, \mathtt{pid}, \mathtt{post\text{-}refund}, \overline{\gamma}_i, \text{tx}_i^{\text{r}}) \overset{\tau}{\hookleftarrow} \mathcal{F}_{update}$ where γ_i.sender is honest:
  1. Sign tx_i^{r} on behalf of γ_i.sender and set $\overline{\text{tx}_i^{\text{r}}} := (\text{tx}_i^{\text{r}}, \sigma_{\gamma_i.\text{sender}}(\text{tx}_i^{\text{r}})$.
  2. Send $(\mathtt{ssid}_L, \mathtt{POST}, \overline{\text{tx}_i^{\text{r}}}) \xrightarrow{\tau+t_c} \mathcal{G}_{ledger}$.

Now, we show that in the view of the environment $\mathcal{E}$, a transcript resulted from interactions between the simulator $\mathcal{X}$ and the ideal functionality $\mathcal{F}_{update}$ is indistinguishable from a transcript resulted from a execution of the protocol Π in the presence of the adversary $\mathcal{A}$. Formally, we want to show that $\mathsf{EXEC}_{\Pi,\mathcal{A},\mathcal{E}}$ and $\mathsf{EXEC}_{\mathcal{F}_{update},\mathcal{X},\mathcal{E}}$ are indistinguishable.

Our protocol Π and ideal functionality $\mathcal{F}_{update}$ both are executed in six phases: *Initialization*, *Pre-Setup*, *Setup*, *Confirmation*, *Finalize*, and *Respond*. For each phase separately, we show how the ideal world and the real world are indistinguishable for the environment.

In our description, we write $m[\tau]$ to denote that message m is observed at round τ. In other meaning, τ is the receiving round for message m (not the round it is sent). Moreover, sometimes we interact with ideal functionalities such as $\mathcal{G}_{channel}$ and $\mathcal{G}_{ledger}$. These functionalities in turn interact with either the environment $\mathcal{E}$ or other parties, who are possibly under adversarial, either by sending messages or additional impacts on publicly observable variables, i.e., the ledger $\mathcal{L}$. To capture this, we define $\text{obsSet}(m, \mathcal{F}, \tau)$ as the set of all observable messages which are triggered by calling $\mathcal{F}$ with message m in round τ.

Lemma 1. *The initialization phase of protocol Π GUC-emulates the initialization phase of the functionality $\mathcal{F}_{update}$.*

Proof. Let τ be the starting round. Note that in the real world environment controls $\mathcal{A}$, and therefore, all corrupted parties. For better readability we define following messages that are used for *Initialization* phase in $\mathcal{F}_{update}$ and Π.

- $m_0 := (\mathsf{sid}, \mathsf{pid}, \mathtt{INIT\text{-}CHECK}, \{\gamma_i\}_{i\in[1,n]})$
- $m_1 := (\mathsf{sid}, \mathsf{pid}, \mathtt{INIT\text{-}CHECKED}, \{\gamma_j\}_{j\in[1,n]})$
- $m_2 := (\mathsf{sid}, \mathsf{pid}, \mathtt{CHANNELS\text{-}SET}, \{\gamma_i\}_{i\in[1,n]})$
- $m_3 := (\mathsf{sid}, \mathsf{pid}, \mathtt{init}, \{\gamma_i\}_{i\in[1,n]})$
- $m_4 := (\mathsf{sid}, \mathsf{pid}, \mathtt{init\text{-}ok})$
- $m_5 := (\mathsf{sid}, \mathsf{pid}, \mathsf{send\text{-}init}, \{\gamma_j\}_{j\in[1,n]}, \mathsf{dealer})$
- $m_6 := (\mathsf{sid}, \mathsf{pid}, \mathsf{send\text{-}check}, \{\gamma_i\}_{i\in[1,n]}, P_i)$
- $m_7 := (\mathsf{sid}, \mathsf{pid}, \mathsf{send\text{-}init\text{-}ok}, \{\gamma_j\}_{j\in[1,n]}, P_i)$
- $m_8 := (\mathsf{sid}, \mathsf{pid}, \mathsf{init\text{-}acc}, P_i, P_j)$

For each participant P_i, we compare messages that $\mathcal{E}$ receives from this party and the trigger party dealer in the ideal world and the real world. The types of the messages depends on corruption cases for P_i and dealer. Note that messages from corrupted parties to $\mathcal{E}$ are not considered, because the environment is commuincating with itself, which is trivially the same in the ideal and the real world.

Case 1: P_i **honest,** dealer **honest**

Real world: $\mathcal{E}$ receives m_3 from dealer in round $\tau + 1$ on behalf of all corrupted parties. Moreover, $\mathcal{E}$ receives m_0 from P_i, which contains the set of all channels in round $\tau+1$. If P_i gets m_1 from $\mathcal{E}$ in the response, then $\mathcal{E}$ receives m_4 from P_i on behalf of all corrupted parties in round $\tau + 2$.

$\mathsf{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_3[\tau + 1], m_0[\tau + 1], m_4[\tau + 2]\}$

Ideal world: $\mathcal{F}_{update}$ sends m_5 to the simulator, which in turn, $\mathcal{X}$ sends m_3 on behalf on dealer to all corrupted parties in round τ. Moreover, $\mathcal{F}_{update}$ sends m_0 on behalf of P_i to $\mathcal{E}$ in round τ. Upon this message is replied by m_1 from $\mathcal{E}$, $\mathcal{F}_{update}$ sends m_7 to the simulator. After receiving this message, $\mathcal{X}$ sends m_4 to all corrupted parties on behalf of P_i in round $\tau + 1$, which is received by $\mathcal{E}$.

$\mathsf{EXEC}_{\mathcal{F}_{update},\mathcal{X},\mathcal{E}} := \{m_3[\tau + 1], m_0[\tau + 1], m_4[\tau + 2]\}$

Case 2: P_i **honest,** dealer **corrupted**

Real world: Because dealer is corrupted, we do not need to consider messages from dealer to $\mathcal{E}$. Other received message are similar to the previous case.

$\mathsf{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_0[\tau + 1], m_4[\tau + 2]\}$

Ideal world: No longer $\mathcal{X}$ is required to send m_3 on behalf of dealer to $\mathcal{E}$. Simulation of the behavior of P_i is done same as the previous case.

$\mathsf{EXEC}_{\mathcal{F}_{update},\mathcal{X},\mathcal{E}} := \{m_0[\tau + 1], m_4[\tau + 2]\}$

Case 3: P_i **corrupted,** dealer **honest**

Real world: We do not to consider messages sent from P_i. $\mathcal{E}$ receives m_3 From dealer on behalf of all corrupted parties.

$\mathsf{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_3[\tau + 1]\}$

Ideal world: $\mathcal{F}_{update}$ sends m_5 to the simulator, which in turn, $\mathcal{X}$ sends m_3 to all corrupted parties who are under the control of $\mathcal{E}$.

$\mathsf{EXEC}_{\mathcal{F}_{update},\mathcal{X},\mathcal{E}} := \{m_3[\tau + 1]\}$

Lemma 2. *The pre-setup phase of protocol* Π *GUC-emulates the pre-setup phase of the functionality* $\mathcal{F}_{update}$.

Proof. Again we compare observed messages by $\mathcal{E}$ in the ideal world and the real world. Let τ be the starting round, and consider the following definitions for all messages that are used for *Pre-Setup* phase in $\mathcal{F}_{update}$ and Π.

- $m_9 := (\mathsf{sid}, \mathsf{pid}, \mathtt{pre\text{-}setup}, \mathsf{tx}_i^{\mathsf{ep}})$
- $m_{10} := (\mathsf{sid}, \mathsf{pid}, \mathsf{presetup\text{-}req}, \gamma_i, \mathsf{tx}_i^{\mathsf{ep}})$
- $m_{11} := (\mathsf{sid}, \mathsf{pid}, \mathsf{send\text{-}presetup}, \mathsf{tx}_i^{\mathsf{ep}}, \gamma_i.\mathsf{receiver}, P_j)$
- $m_{12} := (\mathsf{sid}, \mathsf{pid}, \mathsf{presetup\text{-}acc}, P_i, P_j)$

In this phase, for each channel γ_i, $\mathcal{E}$ receives message only from $\gamma_i.\mathsf{receiver}$, so we should consider only one case. The case that $\gamma_i.\mathsf{receiver}$ is honest.

Real world: $\gamma_i.\mathsf{receiver}$ creates $\mathsf{tx}_i^{\mathsf{in}}$ and $\mathsf{tx}_i^{\mathsf{ep}}$ and sends m_9 to all other parties, so this message is received by $\mathcal{E}$ on behalf of all corrupted parties in round $\tau + 1$.

$\mathsf{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_9[\tau + 1]\}$

Ideal world: $\mathcal{F}_{update}$ first creates $\mathsf{tx}_i^{\mathsf{in}}$ and $\mathsf{tx}_i^{\mathsf{ep}}$ transactions for each channel γ_i. Then, $\mathcal{F}_{update}$ sends m_{11} to the simulator for all corrupted parties P_j. When $\mathcal{X}$ receives this massage, sends m_9 to the all corrupted parties on behalf of $\gamma_i.\mathsf{receiver}$. The messages are received by $\mathcal{E}$ in round $\tau + 1$.

$\mathsf{EXEC}_{\mathcal{F}_{update},\mathcal{X},\mathcal{E}} := \{m_9[\tau + 1]\}$

Lemma 3. *The setup phase of protocol* Π *GUC-emulates the setup phase of the functionality* $\mathcal{F}_{update}$.

Proof. Again we compare observed messages by $\mathcal{E}$ in the ideal world and the real world. Let τ be the starting round, and consider the following definitions for all messages that are used for *Setup* phase in $\mathcal{F}_{update}$ and Π.

- $m_{13} := (\mathsf{sid}, \mathsf{pid}, \mathtt{REQ\text{-}VALUE}, \gamma_i)$
- $m_{14} := (\mathsf{sid}, \mathsf{pid}, \mathtt{VALUE}, \overline{\gamma}_i, \alpha_i)$
- $m_{15} := (\mathsf{sid}, \mathsf{pid}, \mathtt{setup}, \overline{\gamma}_i, \mathsf{tx}_i^{\mathsf{state}}, \{(\mathsf{tx}_{i,j}^{\mathsf{p}}, \sigma_{\gamma_i.\mathsf{sender}}(\mathsf{tx}_{i,j}^{\mathsf{p}}))\}_{j\in[1,n]})$
- $m_{16} := (\mathsf{sid}, \mathsf{pid}, \mathtt{setup\text{-}ok})$
- $m_{17} := (\mathsf{sid}, \mathsf{pid}, \mathsf{send\text{-}setup}, \overline{\gamma}_i, \mathsf{tx}_i^{\mathsf{state}}, \{(\mathsf{tx}_{i,j}^{\mathsf{p}}, \sigma_{\gamma_i.\mathsf{sender}}(\mathsf{tx}_{i,j}^{\mathsf{p}}))\}_{j\in[1,n]})$
- $m_{18} := (\mathsf{sid}, \mathsf{pid}, \mathsf{setup\text{-}acc}, \overline{\gamma}_i, \mathsf{tx}_i^{\mathsf{state}}, \{\mathsf{tx}_{i,j}^{\mathsf{p}}\}\}_{j\in[1,n]})$
- $m_{19} := (\mathsf{sid}, \mathsf{pid}, \mathsf{send\text{-}setup\text{-}ok}, \gamma_i.\mathsf{receiver}, P_j)$
- $m_{20} := (\mathsf{sid}, \mathsf{pid}, \mathtt{setup\text{-}finalized}, P_i, P_j)$

In this phase, for each channel γ_i, both the sender and the receiver have interactions with the environment. We need to consider different corruption cases for these parties except the case that both of them are corrupted.

Case 1: $\gamma_i.\mathsf{sender}$ **honest,** $\gamma_i.\mathsf{receiver}$ **honest**

Real world: $\gamma_i.\mathsf{sender}$ sends m_{13} to $\mathcal{E}$ in round τ. Upon this message is replied by m_{14}, $\gamma_i.\mathsf{sender}$ generates $\mathsf{tx}_i^{\mathsf{state}}$, $\mathsf{tx}_i^{\mathsf{r}}$, and the set $\{\mathsf{tx}_{i,j}^{\mathsf{p}}\}_{j\in[1,n]}$. Then she sends m_{15} to $\gamma_i.\mathsf{receiver}$. When $\gamma_i.\mathsf{receiver}$ gets this message, first asks $\mathcal{E}$ about the payment value via message m_{13} in round $\tau + 1$. Upon this message is replied by m_{14}, $\gamma_i.\mathsf{receiver}$ checks validity of the transactions inside received m_{15}, and then sends m_{16} to all other parties, which is received by $\mathcal{E}$ on behalf of corrupted parties in round $\tau + 2$. Note that two m_{13} messages are received by $\mathcal{E}$ in different rounds. One from the sender and one from the receiver.

$\mathsf{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_{13}[\tau], m_{13}[\tau + 1], m_{16}[\tau + 2]\}$

Ideal world: $\mathcal{F}_{update}$ sends m_{13} to $\mathcal{E}$ on behalf of $\gamma_i.\mathsf{sender}$ in round τ. After receiving the response m_{14}, $\mathcal{F}_{update}$ creates $\mathsf{tx}_i^{\mathsf{state}}$,

$\mathsf{tx}^{\mathsf{r}}_i$, and the set $\{\mathsf{tx}^{\mathsf{p}}_{i,j}\}_{j\in[1,n]}$. Again, $\mathcal{F}_{update}$ sends m_{13} to $\mathcal{E}$ this time on behalf of γ_i.receiver in round $\tau+1$. After receiving the response, $\mathcal{F}_{update}$ sends m_{19} to the simulator, which in turn, $\mathcal{X}$ sends m_{16} to all corrupted parties, which is received in round $\tau+2$.

$\mathsf{EXEC}_{\mathcal{F}_{update},\mathcal{X},\mathcal{E}} := \{m_{13}[\tau], m_{13}[\tau+1], m_{16}[\tau+2]\}$

Case 2: γ_i.sender **honest,** γ_i.receiver **corrupted**

Real world: In this case, we only consider messages that are sent from the sender. Similar to the previous case, γ_i.sender sends m_{13} to $\mathcal{E}$ in round τ, and waits for the response m_{14}. Then she generates $\mathsf{tx}^{\mathsf{state}}_i$, $\mathsf{tx}^{\mathsf{r}}_i$, and the set $\{\mathsf{tx}^{\mathsf{p}}_{i,j}\}_{j\in[1,n]}$ and sends m_{15} to γ_i.receiver. This time the message m_{15} is observed by $\mathcal{E}$ in round $\tau+1$. because the receiver is corrupted.

$\mathsf{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_{13}[\tau], m_{15}[\tau+1]\}$

Ideal world: Similar to the previous case, $\mathcal{F}_{update}$ sends m_{13} to $\mathcal{E}$ on behalf of γ_i.sender in round τ. After receiving the response m_{14}, $\mathcal{F}_{update}$ creates $\mathsf{tx}^{\mathsf{state}}_i$, $\mathsf{tx}^{\mathsf{r}}_i$, and the set $\{\mathsf{tx}^{\mathsf{p}}_{i,j}\}_{j\in[1,n]}$. This time $\mathcal{F}_{update}$ sends m_{17} to the simulator, which in turn, $\mathcal{X}$ sends m_{15} to the corrupted receiver in round τ.

$\mathsf{EXEC}_{\mathcal{F}_{update},\mathcal{X},\mathcal{E}} := \{m_{13}[\tau], m_{15}[\tau+1]\}$

Case 3: γ_i.sender **corrupted,** γ_i.receiver **honest**

Real world: In this case, we only consider messages that are sent from the receiver. At first, When γ_i.receiver gets m_{15} message from the sender, sends m_{13} to $\mathcal{E}$ to get the payment value in round $\tau+1$. Then, this party after receiving the response from $\mathcal{E}$, checks the validity of the transactions inside m_{15}. Finally, she sends m_{16} to all other parties, which received by $\mathcal{E}$ on behalf of corrupted parties in round $\tau+2$.

$\mathsf{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_{13}[\tau+1], m_{16}[\tau+2]\}$

Ideal world: $\mathcal{X}$ gets transactions $\mathsf{tx}^{\mathsf{state}}_i$, and the set $\{(\mathsf{tx}^{\mathsf{p}}_{i,j}, \sigma_{\gamma_i.\mathsf{sender}}(\mathsf{tx}^{\mathsf{p}}_{i,j}))\}_{j\in[1,n]}$ from $\mathcal{A}$ and sends them to $\mathcal{F}_{update}$ via m_{18} if they are correct. $\mathcal{F}_{update}$ sends m_{13} to $\mathcal{E}$ this time on behalf of γ_i.receiver in round $\tau+1$. If this message is reponsed by $\mathcal{E}$ with m_{14}, $\mathcal{F}_{update}$ checks correctness of $\mathsf{tx}^{\mathsf{state}}_i$ received from the simulator. $\mathcal{F}_{update}$ sends m_{19} to the simulator, which in turn, $\mathcal{X}$ sends m_{16} to all corrupted parties in round $\tau+1$.

$\mathsf{EXEC}_{\mathcal{F}_{update},\mathcal{X},\mathcal{E}} := \{m_{13}[\tau+1], m_{16}[\tau+2]\}$

Lemma 4. *The confirmation phase of protocol* Π *GUC-emulates the confirmation phase of the functionality* $\mathcal{F}_{update}$.

Proof. Again we compare observed messages by $\mathcal{E}$ in the ideal world and the real world. Let τ be the starting round, and consider the following definitions for all messages that are used for *Confirmation* phase in $\mathcal{F}_{update}$ and Π.

- $m_{21} := (\mathsf{ssid}_C, \mathsf{UPDATE}, \overline{\gamma}_i.\mathsf{id}, \mathsf{tx}^{\mathsf{state}}_i.\mathsf{output})$
- $m_{22} := (\mathsf{ssid}_C, \mathsf{UPDATED}, \overline{\gamma}_i.\mathsf{id}, \mathsf{tx}^{\mathsf{state}}_i.\mathsf{output})$
- $m_{23} := (\mathsf{sid}, \mathsf{pid}, \mathtt{confirmation}, \sigma_{\gamma_i.\mathsf{sender}}(\mathsf{tx}^{\mathsf{ep}}_j))$
- $m_{24} := (\mathsf{sid}, \mathsf{pid}, \mathtt{OPENED}, \overline{\gamma}_i)$
- $m_{25} := (\mathsf{sid}, \mathsf{pid}, \mathtt{confirmation\text{-}ok})$
- $m_{26} := (\mathsf{sid}, \mathsf{pid}, \mathtt{send\text{-}sig}, \gamma_i.\mathsf{sender}, \gamma_j.\mathsf{receiver}, \mathsf{tx}^{\mathsf{ep}}_j)$
- $m_{27} := (\mathsf{sid}, \mathsf{pid}, \mathtt{confirmation\text{-}acc}, \gamma_i.\mathsf{receiver}, \gamma_j.\mathsf{sender})$
- $m_{28} := (\mathsf{sid}, \mathsf{pid}, \mathtt{send\text{-}confirmation\text{-}ok}, \gamma_i.\mathsf{receiver}, P_j)$
- $m_{29} := (\mathsf{sid}, \mathsf{pid}, \mathtt{confirmation\text{-}finalized}, P_i, P_j)$
- $m_{30} := (\mathsf{sid}, \mathsf{pid}, \mathtt{agg\text{-}sig}, \{\mathsf{tx}^{\mathsf{ep}}_j\}_{j\in[1,n]}, \mathcal{S})$

For each channel γ_i, both the sender and the receiver send messages to $\mathcal{E}$. We need to consider different corruption cases for these parties except the case that both of them are corrupted.

Case 1: γ_i.sender **honest,** γ_i.receiver **honest**

Real world: γ_i.sender sends m_{21} to $\mathcal{G}_{channel}$ in round τ to update the state of $\overline{\gamma}_i$ using $\mathsf{tx}^{\mathsf{state}}_i$. If the update is executed correctly, γ_i.sender sends m_{23} to each receiver. This message is received by $\mathcal{E}$ in behalf of each corrupted receiver in round $\tau+t_u+1$. Again, if the update is executed correctly, γ_i.receiver waits until receiving signatures to $\mathsf{tx}^{\mathsf{ep}}_i$ from all senders. Then, she sends m_{24} to $\mathcal{E}$ in round $\tau+t_u+1$. Also, after verifying all signatures, she sends m_{25} messages to all parties, which are received by $\mathcal{E}$ on behalf of corrupted parties in round $\tau+t_u+2$.

$\mathsf{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_{23}[\tau+t_u+1], m_{24}[\tau+t_u+1], m_{25}[\tau+t_u+2]\} \cup \mathsf{obsSet}(m_{21}, \mathcal{G}_{channel}, \tau)\}$

Ideal world: $\mathcal{F}_{update}$ sends m_{21} massage to $\mathcal{G}_{channel}$. If the update is executed correctly, $\mathcal{F}_{update}$ via message m_{26}, asks $\mathcal{X}$ to generate a signature to each $\mathsf{tx}^{\mathsf{ep}}_j$ on behalf of γ_i.sender and sends it to the corresponding receiver if the receiver is corrupted. This is done via message m_{23} which is received by $\mathcal{E}$ in round $\tau+t_u+1$. Moreover, $\mathcal{F}_{update}$ sends m_{24} to $\mathcal{E}$ in round $\tau+t_u+1$ and m_{28} to the simulator, which in turn, $\mathcal{X}$ sends m_{25} on behalf of γ_i.receiver to all corrupted parties, which is received by $\mathcal{E}$ in round $\tau+t_u+2$.

$\mathsf{EXEC}_{\mathcal{F}_{update},\mathcal{X},\mathcal{E}} := \{m_{23}[\tau+t_u+1], m_{24}[\tau+t_u+1], m_{25}[\tau+t_u+2]\} \cup \mathsf{obsSet}(m_{21}, \mathcal{G}_{channel}, \tau)\}$

Case 2: γ_i.sender **honest,** γ_i.receiver **corrupted**

Real world: In this case, we only consider messages that are sent from the sender. γ_i.sender sends m_{21} to $\mathcal{G}_{channel}$ in round τ. If the update is executed correctly, she sends m_{23} to each receiver. This message is received by $\mathcal{E}$ in behalf of each corrupted receiver in round $\tau+t_u+1$.

$\mathsf{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_{23}[\tau+t_u+1]\} \cup \mathsf{obsSet}(m_{21}, \mathcal{G}_{channel}, \tau)$

Ideal world: Again, $\mathcal{F}_{update}$ sends m_{21} massage to $\mathcal{G}_{channel}$ and if the update is executed correctly, $\mathcal{F}_{update}$ sends m_{26} to $\mathcal{X}$ to generate a signature to each $\mathsf{tx}^{\mathsf{ep}}_j$ on behalf of γ_i.sender. Then $\mathcal{X}$ sends it to the corresponding receiver if she is corrupted via message m_{23} in round $\tau+t_u$.

$\mathsf{EXEC}_{\mathcal{F}_{update},\mathcal{X},\mathcal{E}} := \{m_{23}[\tau+t_u+1]\} \cup \mathsf{obsSet}(m_{21}, \mathcal{G}_{channel}, \tau)$

Case 3: γ_i.sender **corrupted,** γ_i.receiver **honest**

Real world: In this case, we only consider messages that are sent from the receiver. If the update is executed correctly, γ_i.receiver verifies received signatures to $\mathsf{tx}^{\mathsf{ep}}_i$ from all senders, sends m_{24} to $\mathcal{E}$ in round $\tau+t_u+1$, and sends m_{25} messages to all parties in round $\tau+t_u+2$.

$\mathsf{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_{24}[\tau+t_u+1], m_{25}[\tau+t_u+2]\}$

Ideal world: $\mathcal{X}$ receives signatures form a corrupted sender. If the signature is valid $\mathcal{X}$ sends m_{27} to $\mathcal{F}_{update}$. If the update has already executed correctly, then $\mathcal{F}_{update}$ sends m_{24} to $\mathcal{E}$ in round $\tau+t_u+1$. Moreover, sends m_{28} to the simulator, which in turn, $\mathcal{X}$ sends m_{25} on behalf of γ_i.receiver to all corrupted parties in round $\tau+t_u+1$.

$\mathsf{EXEC}_{\mathcal{F}_{update},\mathcal{X},\mathcal{E}} := \{m_{24}[\tau+t_u+1], m_{25}[\tau+t_u+2]\}$

Lemma 5. *The finalizing phase of protocol* Π *GUC-emulates the finalizing phase of the functionality* $\mathcal{F}_{update}$.

Proof. Again we compare observed messages by $\mathcal{E}$ in the ideal world and the real world. Let τ be the starting round, and consider the following definitions for all messages that are used for *Confirmation* phase in $\mathcal{F}_{update}$ and Π.

- $m_{31} := (\texttt{ssid}_C, \mathsf{UPDATE}, \overline{\gamma}_i.\mathsf{id}, \mathsf{tx}_i^{\mathsf{trans}}.\mathsf{output})$
- $m_{32} := (\texttt{ssid}_C, \mathsf{UPDATED}, \overline{\gamma}_i.\mathsf{id}, \mathsf{tx}_i^{\mathsf{trans}}.\mathsf{output})$
- $m_{33} := (\texttt{ssid}_L, \mathsf{POST}, (\mathsf{tx}_i^{\mathsf{ep}}, \mathsf{sigSet}))$
- $m_{34} := (\texttt{sid}, \texttt{pid}, \texttt{FINALIZED}, \overline{\gamma}_i)$
- $m_{35} := (\texttt{sid}, \texttt{pid}, \texttt{post-texp}, \overline{\gamma}_i, \mathsf{tx}_i^{\mathsf{ep}})$

For each channel γ_i, both the sender and the receiver send messages to $\mathcal{E}$. We need to consider different corruption cases for these parties except the case that both of them are corrupted.

Case 1: γ_i.sender **honest,** γ_i.receiver **honest**

Real world: γ_i.sender generates $\mathsf{tx}_i^{\mathsf{in}}$, which transfers α_i coins from the sender to the receiver. Then, sends m_{31} to $\mathcal{G}_{channel}$ in round τ. If the update fails, the receiver sends m_{33} to $\mathcal{G}_{ledger}$ in round $\tau + t_u$ and post $\mathsf{tx}_i^{\mathsf{ep}}$ to the ledger. Finally, γ_i.receiver sends m_{34} to $\mathcal{E}$ in round $\tau + t_u$.

$\mathsf{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_{34}[\tau + t_u]\} \cup \mathsf{obsSet}(m_{31}, \mathcal{G}_{channel}, \tau) \cup \mathsf{obsSet}(m_{33}, \mathcal{G}_{ledger}, \tau + t_u)$

Ideal world: $\mathcal{F}_{update}$ generates $\mathsf{tx}_i^{\mathsf{in}}$ and updates the channel γ_i via sending m_{31} to $\mathcal{G}_{channel}$ in round τ. After the update execution, $\mathcal{F}_{update}$ sends m_{34} to $\mathcal{E}$ in round $\tau + t_u$ and on behalf of the receiver. If the update fails, $\mathcal{F}_{update}$ sends m_{35} to $\mathcal{X}$ and asks it to post $\mathsf{tx}_i^{\mathsf{ep}}$ on the ledger via message m_{33} to $\mathcal{G}_{ledger}$ in round $\tau + t_u$ on behalf of γ_i.receiver.

$\mathsf{EXEC}_{\mathcal{F}_{update},\mathcal{X},\mathcal{E}} := \{m_{34}[\tau + t_u]\} \cup \mathsf{obsSet}(m_{31}, \mathcal{G}_{channel}, \tau) \cup \mathsf{obsSet}(m_{33}, \mathcal{G}_{ledger}, \tau + t_u)$

Case 2: γ_i.sender **honest,** γ_i.receiver **corrupted**

Real world: In this case, we ignore messages that are sent directly from the receiver to $\mathcal{E}$. γ_i.sender generates $\mathsf{tx}_i^{\mathsf{in}}$, and sends m_{33} to $\mathcal{G}_{channel}$ to update the channel.

$\mathsf{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \mathsf{obsSet}(m_{33}, \mathcal{G}_{channel}, \tau)$

Ideal world: $\mathcal{F}_{update}$ generates $\mathsf{tx}_i^{\mathsf{in}}$ and updates the channel γ_i via sending m_{33} to $\mathcal{G}_{channel}$ in round τ.

$\mathsf{EXEC}_{\mathcal{F}_{update},\mathcal{X},\mathcal{E}} := \mathsf{obsSet}(m_{33}, \mathcal{G}_{channel}, \tau)$

Case 3: γ_i.sender **corrupted,** γ_i.receiver **honest**

Real world: In this case, we only consider messages that are sent from the receiver. γ_i.receiver waits until time $\tau + t_u$. If message m_{32} is received in this round, the final transfer has been performed, so γ_i.receiver sends m_{34} to $\mathcal{E}$. If m_{32} is not received and the update fails, sends m_{33} to $\mathcal{G}_{ledger}$ in round $\tau + t_u$.

$\mathsf{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_{34}[\tau + t_u]\} \cup \mathsf{obsSet}(m_{33}, \mathcal{G}_{ledger}, \tau + t_u)$

Ideal world: $\mathcal{F}_{update}$ waits until receiving m_{32} from $\mathcal{G}_{channel}$. If this happens, the update is executed and $\mathcal{F}_{update}$ sends m_{34} to $\mathcal{E}$ on behalf of the receiver in round $\tau + t_u$. Otherwise, $\mathcal{F}_{update}$ sends m_{35} to $\mathcal{X}$ and asks it to send m_{33} to $\mathcal{G}_{ledger}$ on behalf of the receiver.

$\mathsf{EXEC}_{\mathcal{F}_{update},\mathcal{X},\mathcal{E}} := \{m_{34}[\tau + t_u]\} \cup \mathsf{obsSet}(m_{33}, \mathcal{G}_{ledger}, \tau + t_u)$

Lemma 6. *The respond phase of protocol* Π *GUC-emulates the respond phase of the functionality* $\mathcal{F}_{update}$.

Proof. Again we compare observed messages by $\mathcal{E}$ in the ideal world and the real world. Let τ be the starting round, and consider the following definitions for all messages that are used for *Confirmation* phase in $\mathcal{F}_{update}$ and Π.

- $m_{36} := (\texttt{ssid}_C, \mathsf{CLOSE}, \overline{\gamma_i}.\mathsf{id})$
- $m_{37} := (\texttt{ssid}_L, \mathsf{POST}, \overline{\mathsf{tx}_{i,j}^{\mathsf{p}}})$
- $m_{38} := (\texttt{sid}, \texttt{pid}, \mathsf{PAID}, \overline{\gamma}_i)$
- $m_{39} := (\texttt{ssid}_L, \mathsf{POST}, \overline{\mathsf{tx}_i^{\mathsf{r}}})$
- $m_{40} := (\texttt{sid}, \texttt{pid}, \mathsf{FORCE\text{-}REFUND}, \overline{\gamma}_i)$
- $m_{41} := (\texttt{sid}, \texttt{pid}, \texttt{post-pay}, \overline{\gamma}_i, \mathsf{tx}_{i,j}^{\mathsf{p}})$
- $m_{42} := (\texttt{sid}, \texttt{pid}, \texttt{post-refund}, \overline{\gamma}_i, \mathsf{tx}_i^{\mathsf{r}})$

For each channel γ_i, both the sender and the receiver send messages to $\mathcal{E}$ independently. We consider cases that the parties are honest.

Case 1: γ_i.receiver **honest, Pay**

Real world: In every round, γ_i.receiver checks whether one of transactions in $\{\mathsf{tx}_j^{\mathsf{ep}}\}_{j\in[1,n]}$ is observed on the ledger and $\tau < T - t_c - 2\Delta$. If so, she closes the channel γ_i via message m_{36} to $\mathcal{G}_{channel}$. When the channel become closed and $\mathsf{tx}_i^{\mathsf{state}}$ is found on the ledger, γ_i.receiver waits time Δ, and then, post transaction $\mathsf{tx}_{i,j}^{\mathsf{p}}$, which forces the payment. This is done by sending m_{37} to $\mathcal{G}_{ledger}$. The receiver finally sends m_{38} to $\mathcal{E}$ in round $\tau + t_c + 2\Delta$

$\mathsf{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_{38}[\tau + t_c + 2\Delta]\} \cup \mathsf{obsSet}(m_{36}, \mathcal{G}_{channel}, \tau) \cup \mathsf{obsSet}(m_{37}, \mathcal{G}_{ledger}, \tau + t_c + \Delta)$

Ideal world: In every round, $\mathcal{F}_{update}$ checks if one of transactions in $\{\mathsf{tx}_j^{\mathsf{ep}}\}_{j\in[1,n]}$ is observed on the ledger and $\tau < T - t_c - 2\Delta$, sends m_{36} to $\mathcal{G}_{channel}$ to close the channel γ_i. After a successful closure, $\mathcal{F}_{update}$ after a time Δ, send m_{41} to the simulator. The $\mathcal{X}$ aggregates signatures required for spending $\mathsf{tx}_{i,j}^{\mathsf{p}}$ and sends m_{37} to $\mathcal{G}_{ledger}$. When this transaction appears on the ledger, $\mathcal{F}_{update}$ sends m_{38} to $\mathcal{E}$.

$\mathsf{EXEC}_{\mathcal{F}_{update},\mathcal{X},\mathcal{E}} := \{m_{38}[\tau + t_c + 2\Delta]\} \cup \mathsf{obsSet}(m_{36}, \mathcal{G}_{channel}, \tau) \cup \mathsf{obsSet}(m_{37}, \mathcal{G}_{ledger}, \tau + t_c + \Delta)$

Case 2: γ_i.sender **honest, Revoke**

Real world: In every round, when τ is larger than T and channel γ_i has been closed, but not any payment transaction $\mathsf{tx}_{i,j}^{\mathsf{p}}$ is on the ledger, γ_i.sender signs $\mathsf{tx}_i^{\mathsf{r}}$ and post it on the ledger via message m_{39} to $\mathcal{G}_{ledger}$. After observing $\mathsf{tx}_i^{\mathsf{r}}$ on the ledger, γ_i.sender sends m_{40} to $\mathcal{E}$.

$\mathsf{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_{40}[\tau + \Delta]\} \cup \mathsf{obsSet}(m_{39}, \mathcal{G}_{ledger}, \tau)$

Ideal world: In every round, when τ is larger than T and channel γ_i has been closed, $\mathcal{F}_{update}$ sends m_{42} to the simulator, which in turn, $\mathcal{X}$ sign $\mathsf{tx}_i^{\mathsf{r}}$ on behalf of γ_i.sender and sends m_{39} to $\mathcal{G}_{ledger}$. When $\mathsf{tx}_i^{\mathsf{r}}$ is observed on the ledger, $\mathcal{F}_{update}$ sends m_{40} to $\mathcal{E}$ again on behalf of γ_i.sender.

$\mathsf{EXEC}_{\mathcal{F}_{update},\mathcal{X},\mathcal{E}} := \{m_{40}[\tau + \Delta]\} \cup \mathsf{obsSet}(m_{39}, \mathcal{G}_{ledger}, \tau)$

Theorem 2. *For ideal functionalities* $\mathcal{G}_{channel}$, $\mathcal{G}_{clock}$, $\mathcal{F}_{GDC}$, *and* $\mathcal{G}_{ledger}$ *and for any* $T, \Delta \in \mathbb{N}$, *the protocol* Π *GUC-emulates the the functionality* $\mathcal{F}_{update}$.

This theorem follows directly from Lemmas 1 to 6.

C DISCUSSION ON SECURITY AND PRIVACY

In Section 3.1, we introduced the security and privacy goals of interest, atomicity, and strong value privacy. In Section 5.3, we informally showed that the security and privacy goals are achieved by our construction. Further, in Appendix B.4 we defined an ideal functionality $\mathcal{F}_{update}$ for multi-channel updates, and then we proved that the Thora protocol GUC-emulates the ideal functionality. In this section, formalize our security and privacy properties and then prove that $\mathcal{F}_{update}$ fulfills them.

C.1 Atomicity

For our multi-channel updates, let $U := \{(\gamma_i, \alpha_i)\}_{i\in[1,n]}$ be the set of updates. Each tuple (γ_i, α_i) contains a channel γ_i, which will be updated, and a value α_i which determines the update amount of that channel. For each channel γ_i, we define the following possible outcomes. We define γ_i as *successful* if α_i coins have been transferred from the sender to the receiver. I.e., γ_i.balance(γ_i.sender) has been decreased by α_i and additionally γ_i.balance(γ_i.receiver) has been increased by α_i at the end of the protocol execution. We define γ_i as *reverted* if, at the end of the protocol execution, the channel balance is the same as at the start of the protocol execution. A successful or reverted channel γ_i can be *compensated* if one of the users is malicious and deviates from protocol at the cost of losing her funds to the neighboring user without affecting the security of other users. We define γ_i as *punished* if there is an honest node that receives the total channel balance via the channel punishment mechanism. For every other outcome, we say a channel is *invalid*. A channel can have multiple outcomes, e.g., reverted and compensated.

Now, we define a security game $\mathtt{Atom}_{\mathcal{A},\Pi}$ as follows. The adversary $\mathcal{A}$ selects a set of n channels $\{\gamma_1, \gamma_2, ..., \gamma_n\}$, chooses the corrupted users from the users of these channels, selects dealer and sends these values to the challenger. The challenger sets `sid` and `pid` to two random identifiers. With these parameters, the challenger starts running Thora from the *Initialization* phase on the input of the channels set for the given dealer. The behavior of honest parties can be simulated directly by the challenger, and every time a corrupted party needs to be contacted, the challenger sends the query to $\mathcal{A}$ and waits for the corresponding answer. $\mathcal{A}$ can respond correctly, wrongly, not at all, manipulate the ledger by posting (valid) transactions, updating channels, etc.

After the protocol execution terminates, we say that $\mathcal{A}$ wins if one of the following cases holds after the execution.

(1) There exists two channels γ_i, γ_j, each with at least one honest user, where γ_i is successful, and γ_j is reverted, and none of the channels are compensated.
(2) There exists any channel γ_i without two corrupted nodes such that γ_i is invalid or channel γ_j with two honest users such that γ_j is punished.

Definition 2. We say that a multi-channel updates protocol achieves atomicity if for every PPT adversary $\mathcal{A}$, the adversary wins the $\mathtt{Atom}_{\mathcal{A},\Pi}$ game with negligible probability.

Theorem 3. *The multi-channel updates functionality $\mathcal{F}_{update}$ achieves atomicity property defined in Definition 3.*

Proof. Assume that there is an adversary $\mathcal{A}$ that can win the game $\mathtt{Atom}_{\mathcal{A},\Pi}$, which implies that at least one of the two conditions (1) or (2) from the game definitions holds.

Suppose that (1) holds. We have two possible scenarios. First, $\mathcal{F}_{update}$ has created $\mathsf{tx}_i^{\mathsf{trans}}$ in the *Finalizing* phase, and has updated the channel γ_i using $\mathsf{tx}_i^{\mathsf{trans}}$ successfully. Second, at least one $\mathsf{tx}_k^{\mathsf{ep}}$ and $\mathsf{tx}_{i,k}^{\mathsf{p}}$ are on the ledger.

If we are in the first case, both γ_i and γ_j have been entered into the Finalizing phase of $\mathcal{F}_{update}$ because both have at least one honest user. If γ_j.receiver is honest, $\mathcal{F}_{update}$ forces the payment of γ_j either by updating with $\mathsf{tx}_j^{\mathsf{trans}}$ or posting $\mathsf{tx}_j^{\mathsf{ep}}$ in the finalizing phase. Note that if one $\mathsf{tx}^{\mathsf{ep}}$ appears on the ledge, as γ_j.receiver is honest, $\mathcal{F}_{update}$ forces the payment in the response phase.

Now consider the case that γ_j.receiver is malicious. By the assumption γ_j.sender is honest. As $\mathcal{F}_{update}$ has started the finalizing phase, $\mathsf{tx}_j^{\mathsf{trans}}$ should be generated, and γ_j should be tried to be updated using $\mathsf{tx}_j^{\mathsf{trans}}$ unless γ_j.receiver does not cooperate in the updating. In this case, γ_j will be compensated.

If we are in the second case, if γ_j.receiver is honest, $\mathcal{F}_{update}$ forces the payment on behalf of her in the response phase. If γ_j.receiver is malicious, she has refused to force the payment by posting $\mathsf{tx}_{k,j}^{\mathsf{p}}$ and γ_j would be compensated. It follows that (1) cannot hold.

Similarly, (2) cannot hold: The only possible outcomes that the ideal functionality allows for channels with at least one honest nodes are successful, reverted, or compensated. Since both (1) and (2) cannot hold, it follows that such an adversary does not exist.

C.2 Strong value privacy

For a protocol Π and an adversary $\mathcal{A}$, we define another game `VPriv` to capture the strong value privacy property. $\mathcal{A}$ selects dealer, and chooses a set of n channels $\{\gamma_1, \gamma_2, ..., \gamma_n\}$, where for each channel γ_i both γ_i.receiver and γ_i.sender are honest or semi-honest parties. In other words, corrupted parties involved in the protocol do not deviate from the protocol during the execution. The goal $\mathcal{A}$ is to guess the payment values regarding the channels with both honest senders and honest receivers. $\mathcal{A}$ has access to messages sent from honest parties to corrupted ones and publicly auditable parameters, like transactions posted to the ledger.

$\mathcal{A}$ sends the set of channels to the challenger. The challenger sets `sid` and `pid` to two random identifiers. Then, the challenger starts simulating Thora from the *Initialization* phase on the input of the channels set for the given dealer. We assume that messages honest parties receive from $\mathcal{E}$ about the payment values (`REQ-VALUE` messages) are not leaked to any other parties. Moreover, we assume the values $\mathcal{E}$ sends to the receiver and the sender of a single channel are the same.

By the end of the protocol simulation, $\mathcal{A}$ sends the set $\{\alpha'_{i_1}, \alpha'_{i_2}, ..., \alpha'_{i_k}\}$ to the challenger, each α'_{i_j} is the guess of $\mathcal{A}$ for the payment value in channel γ_{i_j} where both the sender and the receiver are honest. We say that $\mathcal{A}$ wins the game if there is at least one $j \in [1, k]$ such that $\alpha'_{i_j} = \alpha_{i_j}$.

Definition 3. We say that a multi-channel updates protocol achieves strong value privacy if for every PPT adversary $\mathcal{A}$, the adversary wins the $\mathtt{VPriv}_{\mathcal{A},\Pi}$ game with negligible probability.

Theorem 4. *The multi-channel updates functionality $\mathcal{F}_{update}$ achieves the strong value privacy property.*

Proof. We assume that k is negligible with regard to the size of the domain which payment values can be chosen from. Thus, without any leaked information about payment values, the probability of the adversary winning the game is negligible.

Suppose that there is an adversary $\mathcal{A}$ that can win the game $\mathtt{VPriv}_{\mathcal{A},\Pi}$ with a non-negligible probability. It means that there is a payment value α_{i_j}, where $\mathcal{A}$ is able to extract some information about the value and guess α'_{i_j}, such that $\alpha'_{i_j} = \alpha_{i_j}$. The only ways to get information about α_{i_j} are the messages $\mathcal{F}_{update}$ sends to corrupted parties and transactions that are posted to the ledger.

α_{i_j} is encoded only in four types of transactions. $\mathsf{tx}^{\mathsf{state}}_{i_j}$, $\{\mathsf{tx}^{\mathsf{p}}_{i_j,k}\}_{k\in[1,n]}$, $\mathsf{tx}^{\mathsf{r}}_{i_j}$, and $\mathsf{tx}^{\mathsf{trans}}_{i_j}$. γ_{i_j}.sender is honest so all these transactions are created by $\mathcal{F}_{update}$. $\mathsf{tx}^{\mathsf{r}}_{i_j}$ and $\mathsf{tx}^{\mathsf{trans}}_{i_j}$ are never sent to other parties inside exchanged messages. Moreover, because γ_{i_j}.receiver is honest, $\mathcal{F}_{update}$ will not sent $\mathsf{tx}^{\mathsf{state}}_{i_j}$, $\mathsf{tx}^{\mathsf{p}}_{i_j,k}$ neither to γ_{i_j}.receiver nor other parties.

On the other hand, since all parties are honest or semi-honest and do not deviate from the protocol, we expect the final update using transaction $\mathsf{tx}^{\mathsf{trans}}_{i_j}$ to be executed successfully for all channels, and no $\mathsf{tx}^{\mathsf{ep}}$ is required to be posted on the ledger. Therefore, in the *respond* phase, $\mathsf{tx}^{\mathsf{state}}_{i_j}$, $\mathsf{tx}^{\mathsf{p}}_{i_j,k}$, or $\mathsf{tx}^{\mathsf{r}}_{i_j}$ are not required to be posted on the ledger, and $\mathcal{A}$ has no way to observe these transactions.