Atomic Multi-Channel Updates with Constant Collateral in Bitcoin-Compatible Payment-Channel Networks

Christoph Egger[1], Pedro Moreno-Sanchez[2], and Matteo Maffei[2]

[1]Friedrich-Alexander University, Erlangen-Nüremberg, egger@cs.fau.de
[2]TU Wien, {pedro.sanchez,matteo.maffei}@tuwien.ac.at

Revision: May 29, 2019

Abstract

Current cryptocurrencies provide a heavily limited transaction throughput that is clearly insufficient to cater to their growing adoption. Payment-channel networks (PCNs) have emerged as an interesting solution to the scalability issue and are currently deployed by popular cryptocurrencies such as Bitcoin and Ethereum. While PCNs do increase the transaction throughput by processing payments off-chain and using the blockchain only as a dispute arbitrator, they unfortunately require high collateral (i.e., they lock coins for a non-constant time along the payment path) and are restricted to payments in a path from sender to receiver. These issues have severe consequences in practice. The high collateral enables griefing attacks that hamper the throughput and utility of the PCN. Moreover, the limited functionality hinders the applicability of current PCNs in many important application scenarios. Unfortunately, current proposals do not solve either of these issues, or they require Turing-complete language support, which severely limits their applicability.

In this work, we present AMCU, the first protocol for atomic multi-channel updates and reduced collateral that is compatible with Bitcoin (and other cryptocurrencies with reduced scripting capabilities). We provide a formal model in the Universal Composability framework and show that AMCU realizes it, thus demonstrating that AMCU achieves atomicity and state privacy. Moreover, the reduced collateral mitigates the consequences of griefing attacks in PCNs while the (multi-payment) atomicity achieved by AMCU opens the door to new applications such as credit rebalancing and crowdfunding that are not possible otherwise. Moreover, our evaluation results demonstrate that AMCU has a performance in line with that of the Lightning Network (the most widely deployed PCN) and thus is ready to be deployed in practice.

1 Introduction

The permissionless nature of major cryptocurrencies such as Bitcoin [31] largely hinders their transaction throughput, limiting it to tens of transactions per second [12]. In contrast, other (centralized) payment networks such as Visa caters a vast mass of users and payments by supporting a transaction throughput of up to tens of thousands of transactions per second [35]. Thus, permissionless cryptocurrencies suffer from a severe scalability issue preventing them from serving a growing base of payments.

In this state of affairs, payment channels have emerged as an interesting mitigation technique for the scalability issue and is currently deployed in popular cryptocurrencies such as Bitcoin or Ethereum [25, 13, 32]. In a nutshell, payment channels aim at establishing a two-party ledger that two users can privately maintain without resorting to the blockchain for every payment and yet ensuring that they can claim their rightful funds in the blockchain at any given time. For that, users first create a deposit transaction that establishes on-chain the initial balances for their two-party ledger. Then, both users issue ledger changes with each other through off-chain accountable messages. Finally, when they are done, they set the last agreed ledger state on the blockchain to get the corresponding coins. For instance, Alice can open a channel with Bob by publishing on the blockchain a transaction that transfers x coins from her to an address addr shared by Alice and Bob. Subsequent payments from Alice to Bob only require that Alice sends Bob an off-chain signed transaction of $y < x$ coins from addr to him. Bob can close the channel by signing and publishing on-chain the last transaction received by Alice. Interestingly, it is possible to generalize this technique to a network of payment channels where two users can pay each other if they are connected through a path of open payment channels [32].

The Lightning Network (LN) [32] for Bitcoin and the Raiden Network [7] for Ethereum are the most widely deployed PCNs in practice, and several implementations exist today [5, 3, 6]. Several academic efforts have focused on designing solutions to enhance the security [23, 28], privacy [17, 22, 27, 30, 29], concurrency [36, 23], availability [24], and routing mechanisms [21, 33] of PCNs. However, there exist fundamental challenges that remain open for PCNs that do not rely on Turing-complete languages such as the one available in Ethereum. In this paper we focus on two fundamental ones, namely, *restricted functionality* and *high collateral*. In fact, it has been conjectured that the collateral challenge cannot be solved without modifications to the Bitcoin script [26]. Here, we refute this conjecture by providing a solution for Bitcoin-compatible PCNs.[1]

Path Restriction. Current PCNs use a tailored two-phase commit protocol to ensure atomicity of a payment: First, the payment amount is locked at each channel in the payment path from the sender to the receiver; and second, each channel is updated (either accepting the payment or releasing the coins) from the receiver to the sender. This, however, limits the functionality of PCNs to payments along paths of payments channels from the sender to the receiver. In this work we observe that a protocol ensuring atomic updates for arbitrary sets of payment channels (not necessarily organized in a path structure) enables the design of off-chain applications that go beyond payments. For instance, a set of users in a PCN can leverage atomic updates in order to rebalance their payment channels when they are depleted or adapt them to facilitate economic interactions in the future.

Moreover, achieving the atomicity of a set of concurrent payments (i.e., *multi-payment atomicity*) enables an even wider range of interesting applications. For instance, consider a crowdfunding application where a set of users want to fund a given receiver by contributing a share of the total pot required by the receiver. Users can leverage multi-payment atomicity to ensure that either each protocol participant in fact contributes her share to the receiver, or coins go back to the original sender. Thus, (multi-payment) atomicity is crucial to unleash the full potential of current PCNs.

Collateral. The execution of a payment of α coins through n payment channels requires to put aside at least $n \cdot \alpha$ coins. Note that while locked, these coins cannot be used for other payments, thus the amount of time that these coins are locked is crucial. The payment protocol must ensure that each intermediate user can enforce on-chain an update in her payment channel in case of dispute with the channel counterparty. Moreover, the payment protocol must ensure that an intermediate honest user does not lose coins. Thus, coins are locked at each channel i for $t_i \geq t_{i+1} + \Delta$, where Δ is the worst-case confirmation time for an on-chain transaction . The rational behind it is that the payment protocol updates one channel at a time starting from the receiver. Thus, after intermediate user i has paid user $i + 1$, she has enough time to require the funds from user $i - 1$ (and eventually use the Δ time to query the funds on-chain if user $i - 1$ does not collaborate

[1] In the rest of the paper, we refer to Bitcoin-compatible PCNs unless otherwise stated.

off-chain).

Therefore, current payment protocols for PCNs require in the worst-case that at least $n \cdot \alpha$ coins are locked in a path of n payment channels for a time of $n \cdot \Delta$ (which is called *collateral* in the blockchain folklore). Thus, Bitcoin-compatible PCNs require a collateral of $\Theta(n^2\alpha\Delta)$ in the worst-case in units *coins* $\times$ *time*, while it has been shown that the collateral can be decreased to $\Theta(n\alpha\Delta)$ for Ethereum-based PCNs [26, 14].

Griefing Attack. The reduction of the collateral is crucial to mitigate the effect of griefing attacks in PCNs. In a nutshell, an adversary with two nodes in the PCN can perform the lock phase of the two-phase commit protocol, setting his nodes as sender and receiver. In this manner, by locking α coins in one of his payment channels, he manages to lock $n - 1 \cdot \alpha$ coins in the payment channels among intermediate users, having therefore an amplification factor of $n - 1$. The effect of this attack can be further amplified if the attacker uses several paths. Moreover, the adversary controlling the receiver can also lock $n \cdot \alpha$ coins among all payment channels in the path by simply refusing to release the solution to the cryptographic challenge associated to the payment and letting the payment fail. Note that in this case the adversary does not need to lock any of his coins. Moreover, although having the payment failed implies in principle that the adversary does not get the associated α coins, the sender might simply retry the payment after some time as a fallback mechanism.

The griefing attack is indeed an open problem in the blockchain community with negative effects for PCNs. First, note that coins on the channel at position i in the path under the attack are locked for a time of $i \cdot \Delta$. As Δ has to account for the time to enforce a transaction on-chain, it must be set to around one hour in the best case when building the PCN on top of Bitcoin. In fact, the LN [6] uses a default Δ value of 144 blocks, that is, approximately one day. Thus, a griefing attack launched over a path of length 7 will lock coins for up to a week. Second, the adversary can use the griefing attack to deplete the payment channel from competitors by setting them as intermediate nodes in the path between the adversarial sender and receiver.

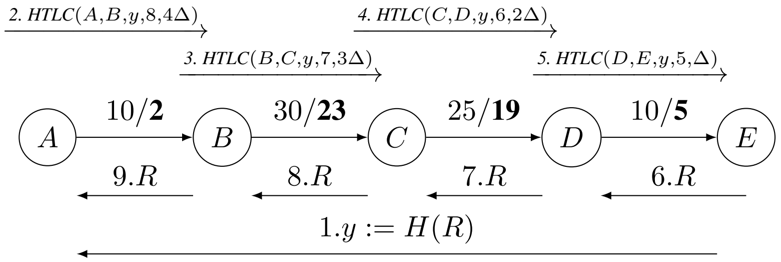
Thus, providing a solution to the high collateral used in PCNs is crucial, as it reduces the amplification factor for the attacker in the griefing attack and it enables a faster release of the coins in a path used for an unsuccessful payment, thereby improving the overall throughput of the PCN. Furthermore, reducing the collateral is crucial given the high volatility of the price of cryptocurrencies (e.g., in November 2018, the price of Bitcoin dropped by \$200 in only one day [4]).

This state of affairs naturally leads to the question: Is it feasible to design a protocol for payments in a PCN that is not path-restricted, reduces the collateral (by at least a factor of n), and is compatible with cryptocurrencies with a restricted scripting language (e.g., Bitcoin)?

Our Contributions. In this work, we give a positive answer to the aforementioned question, by presenting AMCU, the first cryptographic protocol for atomic multi-channel updates with constant collateral. Specifically,

- We provide a formal model in the Universal Composability framework for atomic multi-channel updates in Bitcoin-compatible PCNs, covering the security and privacy notions of interest, such as atomicity and value privacy (Section 3). Atomicity ensures that all payment channels involved in the protocol are updated or none of them is updated. Value privacy ensures that no (off-path) adversary can determine the transaction values.

- We present AMCU, a cryptographic instantiation compatible with cryptocurrencies with a restricted scripting language, such as Bitcoin (Section 5). The cornerstone of AMCU is the use of a MIMO transaction to synchronize the off-chain updates of multiple payment channels while ensuring that each payment channel can still be managed off-chain and is separate from each other after a run of the protocol. We formally prove that AMCU UC-realizes the ideal functionality and thus provides atomicity and value privacy. In fact, AMCU reduces the collateral to $\Theta(n\alpha\Delta)$ which eliminates the amplification factor in the griefing attacks. Moreover, AMCU achieves the same collateral as Ethereum-based solutions and yet it does not rely on smart contracts


2. HTLC(A,B,y,8,4Δ)
4. HTLC(C,D,y,6,2Δ)
3. HTLC(B,C,y,7,3Δ)
5. HTLC(D,E,y,5,Δ)
A
10/**2**
B
30/**23**
C
25/**19**
D
10/**5**
E
9.R
8.R
7.R
6.R
1.$y := H(R)$

(a) Example of payment in the LN from A to B for a value 5 using the HTLC contract. Non-bold (bold) numbers represent the balance of the channel before (after) the payment. We assume each user charges a fee of 1 coin.

Tx2	
In	**Out**
(A, B); 10; $\emptyset$	(A, B); 2; $\emptyset$ B; 8; $[R^* : y = H(R^*)]$ (A, B); 8; [*elapsed*(4Δ)]
$\mathsf{Sig}(A), \mathsf{Sig}(B)$	

Tx3	
In	**Out**
(B, C); 30; $\emptyset$	(B, C); 23; $\emptyset$ C; 7; $[R^* : y = H(R^*)]$ (B, C); 7; [*elapsed*(3Δ)]
$\mathsf{Sig}(B), \mathsf{Sig}(C)$	

Tx4	
In	**Out**
(C, D); 25; $\emptyset$	(C, D); 19; $\emptyset$ D; 6; $[R^* : y = H(R^*)]$ (C, D); 6; [*elapsed*(2Δ)]
$\mathsf{Sig}(C), \mathsf{Sig}(D)$	

Tx5	
In	**Out**
(D, E); 10; $\emptyset$	(C, D); 5; $\emptyset$ E; 5; $[R^* : y = H(R^*)]$ (D, E); 5; [*elapsed*(Δ)]
$\mathsf{Sig}(D), \mathsf{Sig}(E)$	

(b) Transactions required by a HTLC-based payment in the LN. Each transaction contains inputs, outputs and signatures. Each entry in the input/output fields is a triple of the form (address, coins, condition), where condition denotes the requirements to spend the coins at the address apart from the address' signature.

Figure 2.1: Illustrative example of a payment in the LN. Required messages are show in the left. Transactions required in messages 2 to 5 are shown in the right.

that narrow the protocol applicability. In this manner, AMCU solves the collateral challenge in PCNs [26].

- We evaluate the performance of AMCU (Section 6) and we show that AMCU requires only a collateral that is constant along the path. Moreover, it requires $3m + 2$ off-chain transactions, where m denotes the number of payment channels involved in the protocol. We also show that it requires only 3 rounds of communication independently of the number of participants and that communication and computation overheads are negligible even with commodity hardware. Moreover, we show that the performance is in line with the current LN protocol. These results demonstrate that AMCU is practical and ready to be deployed.

- We demonstrate the general applicability of AMCU by showing its applications other than multi-hop payments (Section 7). The first one is the atomic rebalancing of coins among different payment channels in cryptocurrencies with Bitcoin-like scripting language. Secondly, we leverage the multi-payment atomicity property of AMCU to demonstrate its applicability to solve the crowdfunding problem, that is, to ensure that several users can fund a given receiver in such a manner that either all funds are collected by receiver or no payment is actually carried out. These applications demonstrate the usefulness of AMCU to unleash the full potential of current Bitcoin-compatible PCNs.

2 Background

2.1 Payment Channels

A payment channel enables the exchange of coins between two users without settling every single payment in the blockchain. Instead, a single on-chain transaction is used to deposit coins into a multi-signature address controlled by the two users. Consequent payments are carried out off-chain by exchanging signatures over updated states of the deposit. Finally, an additional on-chain transaction is required to close the channel and settle the deposited funds according to the last state.

There exist two types of payment channels: *unidirectional* and *bidirectional*. An unidirectional payment channel supports only payments from Alice to Bob, but not vice-versa. A bidirectional payment channel supports payments in both directions. We refer the reader to [25, 13, 32] for further details. In this work, we consider bidirectional payment channels.

2.2 Payment Channel Network (PCN)

A payment-channel network (PCN) can be represented as a directed graph $\mathcal{G} := (\mathcal{V}, \mathcal{E})$, where the set of nodes $\mathcal{V}$ denotes the blockchain addresses and the set of weighted edges $\mathcal{E}$ denotes open payment channels. Every node $v \in \mathcal{V}$ has associated a non-negative scalar that denotes the fee charged to forward payments in one of its open payment channels, denoted by *fee*(v). Every edge $(v_1, v_2) \in \mathcal{E}$ has associated a function *bal* that denotes the current balance of each node in the channel. For instance, *bal* (v_1, v_2) denotes the amount of remaining coins that v_1 can pay to v_2. Conversely, *bal* (v_2, v_1) denotes the amount of remaining coins that v_2 can pay to v_1.

The cornerstone of PCNs is the ability of enabling payments between any two users connected through a path of open payment channels. The success of a payment depends on the remaining balance in the payment channels that constitute the path from sender to receiver. In particular, assume that *s* wants to pay α coins to *r* through a path of the form *s* $\to v_1 \to v_2, \dots, v_n \to$ *r*. For the payment to be successful, the remaining balance (i.e., *bal* $(\cdot)$) at every payment channel must be at least $\alpha'_i := \alpha - \sum_{j=1}^{i-1} \mathit{fee}(v_i)$ (i.e., the initial payment value minus the fee charged by each intermediate user in the path).

If this requirement is fulfilled, the payment is carried out by updating each payment channel (v_i, v_{i+1}) as follows: *bal* (v_i, v_{i+1}) is reduced by α'_i while *bal* (v_{i+1}, v_i) is increased by α'_i.

2.3 Multi-Hop Payments Atomicity

A fundamental property required in a multi-hop payment is *atomicity*. In a nutshell, either the balance of all payment channels in a path is updated or no payment channel is modified. Note that partial updates might lead to coin losses by honest users. For instance, a user could update her channel with the next user to pay him a certain amount of coins but never receive the corresponding coins from the previous user in the path.

Currently deployed PCNs such as the LN tackle this problem by leveraging a tailored smart contract called *Hash Time-Lock Contract* (HTLC) [34]. This contract can be executed by two users sharing an open payment channel (e.g., Alice and Bob) and allows Alice to lock x coins that can be released only if the contract's condition is fulfilled. The contract's condition is defined based on a collision-resistant hash function H, a hash value $y := H(R)$, where R is chosen uniformly at random, the amount of coins x, and a timeout t. The HTLC contract, which we denote by HTLC(Alice, Bob, y, x, t), has the following clauses: (i) If Bob produces the condition R^* such that $H(R^*) = y$ before timeout t, Alice pays x coins to Bob; (ii) If timeout t expires, Alice gets back the previously locked x coins.

A multi-hop payment in the LN concatenates several HTLC aiming at an atomic payment, as shown in Figure 2.1. In a nutshell, the receiver of the payment creates the value R and gives $y := H(R)$ to the sender. Then, one HTLC is set at each payment channel (v_i, v_{i+1}) of the form HTLC($v_i, v_{i+1}, y, \alpha'_i, t_i$). The HTLC($v_i, v_{i+1}, y, \alpha'_i, t_i$) is translated into a transaction that redistributes the coins available at the channel (e.g., β^{now}) as follows. First, $\beta^{now} - \alpha'_i$ are sent to an address controlled by both v_i and v_{i+1}, effectively sending the coins back to the channel. Second, it sets α'_i coins to be spent by v_{i+1} if R^* is shown. Finally, the same α'_i coins are set to be spent by v_i if the corresponding t_i has elapsed.

When the last HTLC with the receiver is set, then the receiver reveals R^* to the previous user in the path in order to get the payment, starting thereby a chain reaction where each user transfers R^* to her predecessor in the path.

We note two important points in this protocol:

- Each HTLC uses a different number of coins α'_i. As described earlier, this accounts for the transactions fees that each intermediate user charges for providing the forwarding service.
- Each HTLC uses a different timeout t_i. These timeouts must be set such that $t_i \geq t_{i+1} + \Delta$ so that an intermediate user i who gets to know the outcome of the contract in the channel (v_i, v_{i+1}) has enough time Δ to react accordingly (e.g., show the corresponding opening information R^*) for her channel (v_{i-1}, v_i). Unfortunately, although staggered timeouts are crucial for the feasibility of HTLC-based multi-hop payments, they present a severe problem in practice.

In particular, this restriction in setting up timeouts implies that for every pending payment, some coins are held aside at each payment channel as *collateral* until the payment is completed. Although a payment can complete quickly if payment participants collaborate, the collateral can be held for long time in case a user misbehaves (or just goes offline) and the closing transaction solving the dispute must be included in the blockchain. Importantly, this collateral cost (i.e., the lost opportunity of using the value of coins held in reserve) grows with the length of the path and can be up to several hours in Bitcoin.[2]

3 Problem Statement

3.1 Problem Definition

In this section, we formalize the notion of PCN$^+$, a PCN providing atomic multi-channel updates with reduced collateral that can be leveraged for applications beyond payments. In particular, our definition extends the one of a PCN [22] in order to support bidirectional payment channels as well as the functionality required to perform the update of multiple channels that not necessary form a path.

Definition 3.1 (PCN$^+$). *A PCN$^+$ is defined as a graph $\mathcal{G} := (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the set of blockchain addresses and $\mathcal{E}$ is the set of currently opened payment channels. Each payment channel is defined by a tuple $(c_{\langle v_1, v_2 \rangle}, \beta_1^{init}, \beta_2^{init}, \beta_1^{now}, \beta_2^{now}, t)$, where $c_{\langle v_1, v_2 \rangle}$ denotes a channel identifier, β_i^{init} denotes the initial deposit amount of v_i in the channel, β_i^{now} denotes the current balance of v_i in the channel, and t is the channel's expiration time. A PCN$^+$ is defined with respect to a blockchain $\mathbb{B}$ that stores entries of the form $(v, \beta^{on\text{-}chain})$ where v denotes a Bitcoin address and $\beta^{on\text{-}chain}$ denotes its on-chain balance. For clarity, $\mathbb{B}[v]$ denotes the on-chain balance of v. Moreover, we let* time($\mathbb{B}$) *denote the current timestamp in the blockchain. A PCN$^+$ exposes three operations (*`openChannel`*,* `closeChannel`*,* `updateState`*) as described below:*

- `openChannel`$(v_1, v_2, \beta_1, \beta_2, t) \rightarrow \{1, 0\}$. *On input two nodes $v_1, v_2 \in \mathcal{V}$, two initial balances β_1, β_2 and a timeout t, if the operation is authorized by v_1 and v_2, v_1 owns at least β_1 coins (i.e., $\mathbb{B}[v_1] \geq$*

[2] An on-chain transaction is securely added to the blockchain only after one hour in the best case. The LN implementation [6] sets $\Delta = 144$ blocks, that corresponds to 1 day.

β_1) and v_2 owns at least β_2 coins (i.e., $\mathbb{B}[v_2] \geq \beta_2$), `openChannel` *creates a new payment channel $(c_{\langle v_1,v_2 \rangle}, \beta_1, \beta_2, \beta_1, \beta_2, t) \in \mathcal{E}$, where $c_{\langle v_1,v_2 \rangle}$ is a fresh channel identifier. Then it updates the blockchain as $\mathbb{B}[v_1] := \mathbb{B}[v_1] - \beta_1$, $\mathbb{B}[v_2] := \mathbb{B}[v_2] - \beta_2$ and returns 1. Otherwise, it returns 0.*

- `closeChannel`$(c_{\langle v_1,v_2 \rangle}) \to \{1, 0\}$. *On input a channel identifier $c_{\langle v_1,v_2 \rangle}$, the operation works as described below. Let $(c_{\langle v_1,v_2 \rangle}, \beta_1^{init}, \beta_2^{init}, \beta_1^{now}, \beta_2^{now}, t) \in \mathcal{E}$ the channel information for the channel $c_{\langle v_1,v_2 \rangle}$. Then:*
  - *If timeout t has expired in blockchain $\mathbb{B}$ (i.e., $t < \mathsf{time}(\mathbb{B})$),* `closeChannel` *updates $\mathbb{B}$ as $\mathbb{B}[v_1] := \mathbb{B}[v_1] + \beta_1^{init}$ and $\mathbb{B}[v_2] := \mathbb{B}[v_2] + \beta_2^{init}$, removes $(c_{\langle v_1,v_2 \rangle}, \beta_1^{init}, \beta_2^{init}, \beta_1^{now}, \beta_2^{now}, t)$ from $\mathcal{E}$ and returns* 0.
  - *Otherwise,* `closeChannel` *updates $\mathbb{B}$ as $\mathbb{B}[v_1] := \mathbb{B}[v_1] + \beta_1^{now}$ and $\mathbb{B}[v_2] := \mathbb{B}[v_2] + \beta_2^{now}$, removes $(c_{\langle v_1,v_2 \rangle}, \beta_1^{init}, \beta_2^{init}, \beta_1^{now}, \beta_2^{now}, t)$ from $\mathcal{E}$ and returns* 1.

- `updateState`$\left(\{(c_{\langle v_{2i-1},v'_{2i} \rangle}, \delta_i)\}_{i \in [1\ldots n]}\right) \to [b_i]_{i \in [1,n]}$. *On input a set of tuples of the form $(c_{\langle v_{2i-1},v'_{2i} \rangle}, \delta_i)$, where $c_{\langle v_{2i-1},v'_{2i} \rangle}$ denotes a channel identifier and δ_i the update value for this channel, the operation works as described below. First, it initalizes output as an empty array of length n. Second, for each tuple in the input set, let $(c_{\langle v_{2i-1},v_{2i} \rangle}, \beta_{2i-1}^{init}, \beta_{2i}^{init}, \beta_{2i-1}^{now}, \beta_{2i}^{now}, t_i) \in \mathcal{E}$ be the information for the channel $c_{\langle v_{2i-1},v_{2i} \rangle}$. Then:*
  - *If the operation is authorized by v_{2i-1} and $\beta_{2i-1}^{now} - \delta_i \geq 0$, then* `updateState` *updates the channel as $(c_{\langle v_{2i-1},v_{2i} \rangle}, \beta_{2i-1}^{init}, \beta_{2i}^{init}, \beta_{2i-1}^{now} - \delta_i, \beta_{2i}^{now} + \delta_i, t_i)$ and sets $\mathsf{output[i]=1}$.*
  - *Otherwise,* `updateState` *sets $\mathsf{output[i]=0}$.*

Finally, after all tuples are processed in the previous step, `updateState` *returns output.*

We note that `updateState` is a generic multi-channel update operation for PCNs and thus it provides, among others, the core functionality to update the network according to payments, as we discuss in Section 7.

3.2 Security and Privacy Goals

We now introduce informally the security and privacy notions of interest for PCN$^+$.

- **Atomicity:** We say that a PCN$^+$ achieves atomicity if for every state update call `updateState`$\left(\{(c_{\langle v_{2i-1},v'_{2i} \rangle}, \delta_i)\}_{i \in [1\ldots n]}\right)$ with output $[b_i]_{i \in [1,n]}$, either every $b_i = 1$ or every $b_i = 0$.

For this property to make sense, we have to assume that at least one of the protocol participants is honest. Otherwise, we end up in a situation where the adversary is running the protocol alone with payment channels under his control. Thus, the adversary can always break any notion of atomicity as he can always deviate from the protocol without any other user checking it.

- **Value Privacy:** We say that a PCN$^+$ achieves value privacy if for every state update call `updateState`$\left(\{(c_{\langle v_{2i-1},v'_{2i} \rangle}, \delta_i)\}_{i \in [1\ldots n]}\right)$, no adversary other than protocol participants can determine the transaction values δ_i. This property is the same as the one defined by Malavolta et al. [22], although we provide here a richer functionality (i.e., generic state updates instead of only payments). Notice also that protocol participants are clearly entitled to know their current balance before and after the multi-channel update is carried out, thus hiding transacting values from protocol participants seems inherently hard. Finally, although value privacy might seem at a first glance a relatively weak privacy property, it is not trivial to achieve: in fact, although there exist atomicity protocols for Ethereum-based PCNs that do achieve value privacy [14], several others do not [20, 26].

3.3 Ideal World Functionality

Attacker Model. We model users in our protocol as interactive Turing machines that interact with a trusted functionality $\mathcal{F}$ via secure and authenticated channels. We model the attacker $\mathcal{A}$ as an interactive Turing machine that has access to an interface $\mathsf{corrupt}(\cdot)$ that on input a user identifier U provides the attacker with the inputs of U. Moreover, all subsequent incoming and outgoing communication of U is then routed through $\mathcal{A}$. We allow for byzantine (malicious) corruption of any set of users (i.e., we do *not* assume honest majority) but only allow for efficient adversaries that run in *probabilistic polynomial time* (PPT) and accept a *negligible* success probability from their side. For clarity of presentation, we consider *static* corruption only.[3] We note that our attacker model is in line with the state-of-the-art in the literature [14, 15, 18, 22, 23].

Communication Model. We model the communication with the secure message transmission functionality $\mathcal{F}_{smt}$. This functionality informs $\mathcal{A}$ whenever a communication between two users happens and allows the attacker to delay the delivery of the messages arbitrarily. However, the adversary cannot read nor change the content of the messages. We refer the reader to [9] for a concrete description of this functionality.

Moreover, we assume a synchronous communication network as modeled by $\mathcal{F}_{syn}$, where the execution of the protocol happens in discrete rounds. In particular, the users are always aware of the current round and if a message is created at round i, then this message is delivered at the beginning of round $(i+1)$. The adversary can decide about the order in which messages arrive in a given round, but he cannot change the order of messages sent between honest parties. The latter can be achieved by including counters in the messages. For simplicity, we assume that computation is instantaneous. We refer the reader to [9, 19] for more details about $\mathcal{F}_{syn}$.

Modeling on-chain balances via global ledger functionality. We consider a global ideal ledger functionality $\mathcal{L}$ in the global UC (GUC) model [11], since the ledger functionality contains publicly available information that can be updated not only by our ideal functionality but also other protocols simultaneously. In particular, the state of $\mathcal{L}$ is entirely public and it consists of a set of tuples $((v, \mathsf{txid}), \beta)$ that denote an account v created in transaction with id txid, its current on-chain balance β. We present the details of $\mathcal{L}$ in Appendix B.

Notation. We denote by $\mathsf{time}(\mathcal{L})$ the current timestamp in the ledger. We denote by $\texttt{send}_s$ and $\texttt{receive}_s$ the interfaces to send and receive messages through the $\mathcal{F}_{smt}$ functionality.

Assumptions. For readability, we assume that there exists only one channel open at a time between any pair of users. This can be easily relaxed by adding an additional identifier to each channel apart from the two users controlling it. We assume that if users i and j are willing to deposit β_i and β_j coins in a shared channel $c_{\langle v_1, v_2 \rangle}$, they have exactly those coins in their respective addresses v_i and v_j. In practice, if a user i has more coins than β_i in her address, she can split it into two addresses so that this assumption holds. Moreover, we assume that users have agreed upon a timeout T_Δ used to freeze current coins at their payment channels to perform the `updateState` operation: We assume that the `updateState` call requires a time smaller than T_Δ, which can be easily achievable by adjusting the value of T_Δ as the system parameter. Finally, we assume that blockchain follows an UTXO model (as in Bitcoin), that is, each address can be spent only once.

Operations. We model our operations in the UC Framework [9, 8] as shown in Figure 3.1.

The `openChannel` operation simply ensures that both users agree on the opening of the channel (steps 1-2) and, if so, it creates a new channel account where channel counterparties deposit their coins (step 3). Finally, the functionality stores the information about the new channel to be used in future invocations.

The `closeChannel` operation first checks if the channel is still active (step 1). If so, then it closes it by enforcing the last agreed off-chain balance in the ledger (step 2).

[3] The extension of our protocols to support adaptive corruption is an interesting open problem.

The `updateChannel` operation updates the current state of the channel. The operations enters in several rounds where all participants are asked to agree on the execution of next phase and the response is replied to all other participants in the protocol (steps 1-4). The different phases model the different communication rounds required to achieve atomic multi-channel updates.

Finally, if the pre-agreed time $T_\triangle$ has not elapsed yet, then all the channels are updated with their corresponding updated values. Otherwise, all users are notified of the fact that the fallback phase has been reached (step 5).

Restrictions on the Environment. In this work we consider a restricted environment that is not allowed to perform a set operations that we separate in two groups. First, the environment never asks to open a channel $c_{\langle v_1, v_2 \rangle}$ that has already been opened; to close a channel that has been closed before; to update a channel that has not been opened before; or to close a channel in a stale state if a newer state has been satisfactorily updated. This first group of restrictions are purely to simplify our presentation: Honest parties could just return $\perp$ when the environment tries to re-open or re-close a channel, or use a stale state. Moreover, our construction can be easily integrated with the standard revocation mechanism from PCNs to prevent malicious users to use stale states. [32]

Second, we assume that the environment does not invoke `updateState` requests including channels that do not have the sufficient balance. In practice, this assumption can be easily relaxed if users refuse to participate in a `updateState` protocol that requests a channel without sufficient balance. This blocking mechanism could lead to deadlocks, a known concurrency issue in PCNs [22]. Techniques to avoid deadlocks is an interesting and orthogonal problem considered in the literature [22, 36].

Universal composability. In this section, we provide the definition of universal composability. We follow the definition of Canetti [9] and extend it to consider the interaction with the ledger functionality $\mathcal{L}$, following the literature [14, 15].

Definition 3.2 (Universal Composability)**.** *Let* $\textsf{exec}_{\Pi,\mathcal{A},\mathbb{Z}^*}$ *denote the random variable (over the local random choices of all involved machines) describing the output of the restricted environment* $\mathbb{Z}^*$ *when interacting with adversary* $\mathcal{A}$ *and parties running the protocol* Π. *We say that protocol* Π *UC-emulates the ideal functionality* $\mathcal{F}$ *with respect to a ledger* $\mathcal{L}$ *if for any adversary* $\mathcal{A}$ *there exists a simulator* $\mathcal{S}$ *such that, for any restricted environment* $\mathbb{Z}^*$ *the distributions of* $\textsf{exec}_{\Pi,\mathcal{A},\mathbb{Z}^*}$ *and* $\textsf{exec}_{\mathcal{F},\mathcal{S},\mathbb{Z}^*}$ *are indistinguishable (i.e., the probability that* $\mathbb{Z}^*$ *outputs* 1 *after interacting with* $\mathcal{A}$ *and* Π *differs at most negligibly from the probability that* $\mathbb{Z}^*$ *outputs* 1 *after interacting with* $\mathcal{F}$ *and* $\mathcal{S}$.

Discussion. Here we discuss how $\mathcal{F}$ captures the security and privacy notions of interest for a PCN$^+$.

Atomicity: In the steps 1 to 4 of `updateChannel`, $\mathcal{F}$ queries every user before going to the next step. If any user refuses, then no channel is updated, every user is notified and the `updateChannel` is aborted. In step 5, $\mathcal{F}$ notifies all users of the protocol outcome: either a successful update or a fallback if the update is unsuccessful (i.e., $T_\triangle$ has expired). In the former, $\mathcal{F}$ atomically updates the balances of all channels involved in the protocol. In the latter, $\mathcal{F}$ does not update any channel and notifies all users. Therefore, as $\mathcal{F}$ is a trusted party, the protocol outcome is the same for all users, which shows that our model captures atomicity.

Value privacy: $\mathcal{F}$ interacts only with the channel owners, without leaking any information to third parties (for off-chain `updateChannel` operations).

openChannel:

Upon receiving a message $(\mathsf{sid}, \underline{\textsf{open-channel}}, v_j, \beta_i, \beta_j, \mathsf{txid}_i, \mathsf{txid}_j, \sigma_i)$ from v_i (symmetrical for v_j), proceed as follows:

1. $\texttt{send}_s(\mathsf{sid}, \underline{\textsf{ch-op-notify}})$ to v_j and $\texttt{receive}_s(\mathsf{sid}, \underline{\textsf{ch-op-notify}}, \sigma_j)$ from v_j. If $\sigma_j = \perp$ aborts. Otherwise, continue.
2. Create a channel identifier $c_{\langle v_1, v_2\rangle}$ and $\texttt{send}_s(\mathsf{sid}, \underline{\textsf{commit-transfer}}, (\{(\mathsf{txid}_i, v_i), (\mathsf{txid}_j, v_j)\}, \{(c_{\langle v_1, v_2\rangle}, \beta_i + \beta_j)\}, 0, \{\sigma_i, \sigma_j\}))$ to $\mathcal{L}$ and $\texttt{receive}_s(\mathsf{sid}, b)$ from $\mathcal{L}$. If $b = \perp$, $\texttt{send}_s(\mathsf{sid}, \underline{\textsf{ch-op-abort}})$ to v_i and v_j. Otherwise, continue.
3. Otherwise, $\mathcal{F}^+_{pcn}$ stores the tuple $(c_{\langle v_1, v_2\rangle}, \mathsf{txid}, \beta_1, \beta_2)$ in $\mathbb{C}$. Finally, $\texttt{send}_s(\mathsf{sid}, \underline{\textsf{ch-op-success}})$ to v_i, v_j, and the simulator $\mathcal{S}$.

closeChannel:

Upon receiving a message $(\mathsf{sid}, \underline{\textsf{close-channel}}, c_{\langle v_i, v_j\rangle}, \sigma_{i,j})$ from v_i (symmetrical for v_j), proceed as follows:

1. Let $c := (c_{\langle v_i, v_j\rangle}, \mathsf{txid}, \beta_i^{now}, \beta_i^{now},)$ be the tuple in $\mathbb{C}$ that represents the channel between v_i and v_j.
2. $\texttt{send}_s(\mathsf{sid}, \underline{\textsf{commit-transfer}}, \{(c_{\langle v_i, v_j\rangle}, \mathsf{txid})\}, \{(v_i, \beta_i^{now}), (v_j, \beta_j^{now})\}, \{\sigma_{i,j}\})$ to $\mathcal{L}$ and $\texttt{receive}_s(\mathsf{sid}, b)$ from $\mathcal{L}$. If $b = \perp$, then $\texttt{send}_s(\mathsf{sid}, \underline{\textsf{ch-cl-abort}})$ to v_i and v_j. Otherwise, remove c from $\mathbb{C}$ and $\texttt{send}_s(\mathsf{sid}, \underline{\textsf{ch-cl-success}})$ to v_i, v_j and the simulator $\mathcal{S}$.

updateState:

Upon receiving a message $(\mathsf{sid}, \underline{\textsf{state-update}}, \{(c_{\langle v_{2i-1}, v'_{2i}\rangle}, \delta_i)\}_{i \in [1 \ldots n]})$, proceed as described below. Let v_0 be a pre-defined user among the set of users (i.e., the one with the lowest identifier after sorting them lexicographically).

1. For each channel in $\{(c_{\langle v_{2i-1}, v'_{2i}\rangle}, \delta_i)\}$ do the following actions: (i) $\texttt{send}_s(\mathsf{sid}, \underline{\textsf{setup-query}}, \{(c_{\langle v_{2i-1}, v'_{2i}\rangle}, \delta_i)\}$ to v_{2i-1} and v_{2i}; (ii) $\texttt{receive}_s(\mathsf{sid}, b)$ from v_{2i-1} and $\texttt{receive}_s(\mathsf{sid}, b')$ from v_{2i}; (iii) If $b = \perp$ or $b' = \perp$, do $\texttt{send}_s(\mathsf{sid}, v, \underline{\textsf{setup-abort}})$ for all v in $\mathcal{V}$. Otherwise, $\texttt{send}_s(\mathsf{sid}, v, \underline{\textsf{setup-success}})$ for all v in $\mathcal{V}$.
2. For each channel in $\{(c_{\langle v_{2i-1}, v'_{2i}\rangle}, \delta_i)\}$ do the following actions: (i) $\texttt{send}_s(\mathsf{sid}, \underline{\textsf{lock-query}}, \{(c_{\langle v_{2i-1}, v'_{2i}\rangle}, \delta_i)\}$ to v_{2i-1} and v_{2i}; (ii) $\texttt{receive}_s(\mathsf{sid}, b)$ from v_{2i-1} and $\texttt{receive}_s(\mathsf{sid}, b')$ from v_{2i-1}; (iii) If $b = \perp$ or $b' = \perp$, do $\texttt{send}_s(\mathsf{sid}, v, \underline{\textsf{lock-abort}})$ for all v in $\mathcal{V}$. Otherwise, $\texttt{send}_s(\mathsf{sid}, v, \underline{\textsf{lock-success}})$ for all v in $\mathcal{V}$.
3. For each channel in $\{(c_{\langle v_{2i-1}, v'_{2i}\rangle}, \delta_i)\}$ do the following actions: (i) $\texttt{send}_s(\mathsf{sid}, \underline{\textsf{consume-query}}, \{(c_{\langle v_{2i-1}, v'_{2i}\rangle}, \delta_i)\})$ to v_{2i-1} and v_{2i}; (ii) $\texttt{receive}_s(\mathsf{sid}, b)$ from v_{2i-1} and $\texttt{receive}_s(\mathsf{sid}, b')$ from v_{2i}; (iii) If $b = \perp$ or $b' = \perp$, do $\texttt{send}_s(\mathsf{sid}, v, \underline{\textsf{consume-abort}})$ for all v in $\mathcal{V}$. Otherwise, $\texttt{send}_s(\mathsf{sid}, v, \underline{\textsf{consume-success}})$ for all v in $\mathcal{V}$.
4. For each channel in $\{(c_{\langle v_{2i-1}, v'_{2i}\rangle}, \delta_i)\}$ do the following actions: (i) $\texttt{send}_s(\mathsf{sid}, \underline{\textsf{enable-query}}, \{(c_{\langle v_{2i-1}, v'_{2i}\rangle}, \delta_i)\}$ to v_{2i-1} and v_{2i}; (ii) $\texttt{receive}_s(\mathsf{sid}, b)$ from v_{2i-1} and $\texttt{receive}_s(\mathsf{sid}, b')$ from v_{2i}; (iii) If $b = \perp$ or $b' = \perp$, do $\texttt{send}_s(\mathsf{sid}, v, \underline{\textsf{enable-abort}})$ for all v in $\mathcal{V}$. Otherwise, $\texttt{send}_s(\mathsf{sid}, v, \underline{\textsf{enable-success}})$ for all v in $\mathcal{V}$
5. If $T_\Delta < \mathsf{time}(\mathcal{L})$ update each tuple in $\mathcal{E}$ corresponding to $\{c_{\langle v_{2i-1}, v'_{2i}\rangle}\}$ as $(c_{\langle v_{2i-1}, v'_{2i}\rangle}, \beta^{now}_{2i-1} - \delta_{2i-1,2i}, \beta^{now}_{2i} + \delta_{2i-1,2i})$. Moreover, do $\texttt{send}_s(\mathsf{sid}, \underline{\textsf{update-success}})$ to $v_i \in \mathcal{V}$. Otherwise, $\texttt{send}_s(\mathsf{sid}, \underline{\textsf{update-fallback}})$ to $v_i \in \mathcal{V}$.

Figure 3.1: Ideal Functionality for PCN$^+$

4 Solution Overview

4.1 System assumptions

Assumptions. We assume that every user participating in the protocol is aware of all other participants and can send confidential and authenticated messages. This can be realized in practice by establishing TLS channels among the participants. We also assume that the participants in the protocol agree on a *coordinator*, that is, one of the participants that help with the coordination of the protocol phases. This can be easily set by choosing the first user in a pre-agreed sorted list (e.g., by sorting lexicographically their blockchain addresses). Note that the coordinator serves to reduce the number of network messages, but she has not advantage in terms of security or privacy. We further assume that they underlying blockchain follows the UTXO model (e.g., like in Bitcoin), where each output can be spent only once. We note that virtually all cryptocurrencies today (with the exception of Ethereum) follow the UTXO model.

UTXO vs Account Model. Addresses in the UTXO model work differently to the account model (e.g., in Ethereum). Transactions in Bitcoin are linked by transaction identifiers, while they are linked by addresses in Ethereum. In a bit more detail, coins held at an address in Ethereum can be spent without indicating where they came from (e.g., what previous transactions sent those coins). In Bitcoin (and the UTXO model in general), instead, a transaction must indicate the origin (i.e., previous transaction) of each coin to be transferred. In the example shown in Figure 4.1, the transaction Tx^3_{ETH} does not specify the origin of the 6 coins stored at address B. This implies that even if Tx^2_{ETH} is not in the blockchain, Tx^3_{ETH} can still be added after Tx^1_{ETH} (with the only difference that B would hold 2 coins instead of 6). The transaction Tx^3_{BTC}, instead, indicates that it specifically uses the coins held at the address B created at Tx^2_{BTC}. This implies that if Tx^2_{BTC} is not in the blockchain, Tx^3_{BTC} cannot be added to the blockchain although B has also received coins in Tx^1_{BTC}. This a key difference that we leverage in our solution: A transaction cannot be added to the blockchain if it points to a previous transaction that has not been published, *even if the pointed address has received arbitrarily many other coins from other transactions*.

4.2 Protocol overview

Our protocol for atomic multi-channel updates (AMCU) proceeds in four phases where the protocol participants create *authenticated off-chain transactions*, as depicted in Figure 4.2. Notice that all following phases are conducted off-chain, which is crucial for scalability and privacy reasons. In the following, we describe these phases.

Phase I: Setup. The first phase requires to freeze the coins available at each channel involved in the protocol. Doing this naively (i.e., locking the complete balance in the channel at once) would lock more coins than required, unnecessarily increasing the collateral in the protocol. Instead, during the setup phase, the balance at each payment channel is split in two, effectively creating thereby two sub-channels: one sub-channel is set with the coins required for the present protocol session, while the other one is set with the remaining coins, which can then be freely spent.

In the illustrative example shown in Figure 4.2, the setup phase starts with the user A collaborating with user B to create the transaction Tx^A_{setup}, where they split the 10 coins they have in the channel in two sub-channels: one sub-channel with 8 coins to be used in the rest of the protocol session and one sub-channel with the rest (i.e., 2 coins). This transaction is signed by both users so that it can be eventually enforced on-chain if required. The rest of the users behave analogously. Note that operations at each channel in this phase of the protocol can be carried out in parallel. Finally, this phase ends when all users acknowledge each other of the success of this phase. For simplicity, we denote this by sending OK to the coordinator (A in this

Tx^1_{ETH}

In	Out
A; 5; $\emptyset$	B; 2; $\emptyset$ A; 3; $\emptyset$
Sig(A)	

Tx^2_{ETH}

In	Out
C; 10; $\emptyset$	B; 4; $\emptyset$ C; 6; $\emptyset$
Sig(C)	

Tx^3_{ETH}

In	Out
B; 6; $\emptyset$	D; 1; $\emptyset$ B; 5; $\emptyset$
Sig(B)	

Tx^1_{BTC}

In	Out
$Tx^*_{BTC}[A]$; 5; $\emptyset$	B; 2; $\emptyset$ A; 3; $\emptyset$
Sig(A)	

Tx^2_{BTC}

In	Out
$Tx^*_{BTC}[C]$; 10; $\emptyset$	B; 4; $\emptyset$ C; 6; $\emptyset$
Sig(C)	

Tx^3_{BTC}

In	Out
$Tx^2_{BTC}[B]$; 4; $\emptyset$	D; 1; $\emptyset$ B; 3; $\emptyset$
Sig(B)	

Figure 4.1: Illustrative example account model (left) vs UTXO model (right). Here, we assume that transactions are submitted to the blockchain in order (e.g., Tx^1 before Tx^2). Here, $Tx^2_{BTC}[B]$ denotes the address B created as output in Tx^2_{BTC}. Tx^* denotes the identifier of a previous transaction not specified here.

example). At this point, we consider only the sub-channel with the amount of coins required for the rest of the protocol. For instance, in our running example, we consider only the sub-channel with 8 coins between A and B. We abuse the notation and call it channel in the rest of this presentation.

Phase II: Lock. At this phase, the off-chain state at each payment channel is superseded by a new state with same balance but locked until a certain pre-agreed time (the system parameter $T_\triangle$) in the future, which can be realized through the timelock mechanism.

In our running example, the lock phase starts with user A collaborating with user B to create the transaction Tx^A_{lock}, where they simply transfer the coins from the channel $Tx^A_{setup}[(A_3, B_3)]$ to the channel (A_4, B_4) controlled also by them, but adding the condition that this can be enforced only until the $T_\triangle$ time has elapsed. As in the previous phase, this transaction is signed by both users so that it becomes enforceable on-chain after the condition has been satisfied; the rest of users behave analogously and operations at each channel can be performed in parallel; the phase ends when the coordinator receives the acknowledgement from all parties.

This channel configuration provides two key features. First, if an adversary prevents any of the future phases to continue, the channel's state created at this phase can be retaken as valid after the $T_\triangle$ has elapsed, having thereby a safe fallback mechanism. Second, the locking of each channel provides a period of $T_\triangle$ time where user can jointly carry out the rest of the phases to build the atomic multi-channel update that will supersede the currently frozen state.

Phase III: Consume. In the consume phase, each pair of users sharing a channel updates its state in order to transfer the coins to the receiver. However, we have to ensure that atomicity is preserved, that is, either all channels do transfer the coins to the corresponding receiver, or none of them does it. The cornerstone of our approach towards this goal is that each transaction that sends the coins to the expected receiver is appended with an additional fresh input address that does not exist yet. In this manner, the whole transaction is not valid (i.e., cannot be enforced on-chain) until the fresh address is created and funded (cf. UTXO vs account model before).

Following with our running example, the consume phase starts with user A collaborating with user B to create the transaction $Tx^A_{consume}$. In this transaction, they transfer 7.99 coins from the channel to the intended receiver (B in this case). The remaining 0.01 coins come from a fresh address $e_{A,B}$ that has not been funded yet. Hence, even if B attempts to submit $Tx^{A}_{consume}$ to the blockchain, miners will reject it as one of the inputs includes an identifier for a transaction that has not been included yet in the blockchain.

We note that, at this point, the coins at the channel between A and B (i.e., coins at the address (A_3, B_3) created in Tx^{A}_{setup}) are referred by two simultaneous and contradictory transactions, none of which can be enforced yet. First, the Tx^{A}_{lock} transfers the coins back to a fresh channel (A_4, B_4), but is timelocked until T_Δ elapses. Second, $Tx^{A}_{consume}$ transfers the coins to the receiver end of the channel, but $e_{A,B}$ must be funded by Tx_{enable} first. We also note the that the rest of channels in the protocol are in an analogous situation. Thus, what we need to do in the rest of the protocol is to enable the fresh addresses $e_{i,j}$ in an atomic manner, so that all channels become *spendable* simultaneously.

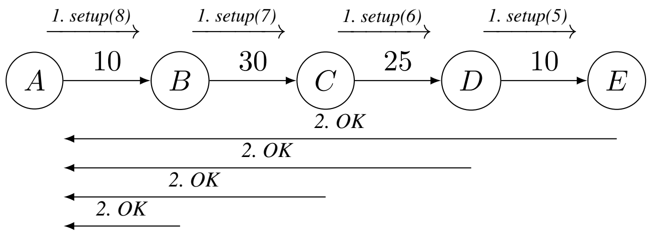
Phase IV: Finalize. In this phase, protocol participants jointly create a MIMO transaction that transfers coins from each channel back to a fresh channel controlled by the same participants, except for a small amount that is used to fund the $e_{i,j}$ fresh address introduced for the atomicity purpose. In principle, collecting signatures from all participants to make this transaction valid should suffice, as this would enable atomically each of the $Tx^{i}_{consume}$ transactions. In other words, after Tx_{enable} is signed by all users, each pair of users have a valid off-chain state (i.e., a chain of signed transactions) that pays to the intended receiver. For instance, for the channel between A and B, the transaction Tx_{enable} enables the address $e_{A,B}$ that in turn makes valid the transaction $Tx^{A}_{consume}$ that transfers the coins from the channel to B (as expected by the protocol). Thus, A and B would accept this as a valid transition of state as they could enforce it by submitting Tx_{enable} and $Tx^{A}_{consume}$ to the blockchain (in this order).

However, there is a subtlety that needs to be handled. After the time T_Δ has expired, each channel has two states that can be enforced in the blockchain. For instance, in the case of the channel between A and B, they could use Tx^{A}_{lock} to transfer the coins from $Tx^{A}_{setup}[(A_4, B_4)]$ to another address also managed by them. Additionally, they could also use Tx_{enable} and $Tx^{A}_{consume}$ to get the coins transferred to B in this case. Thus, at this point there are two contradictory (still off-chain) states that can be included in the blockchain.

In order to solve this issue, we have to add a condition to Tx_{enable} of the type *make invalid if* T_Δ *has elapsed*. Unfortunately, such a transaction is not built-in in restricted scripting languages such as the one in Bitcoin: timelock statements allow to make a transaction invalid before a given time and valid afterwards, but not vice-versa, which, however, is what we need here (see [2] for more details). Thus, we simulate it by letting the protocol participants create a transaction $Tx_{disable}$ that is defined to revert the effect of Tx_{enable}, that is, to send back all the coins on each channel to another channel managed by the same users. For instance, in our running example, $Tx_{disable}$ transfers the coins from $Tx_{enable}[(A_5, B_5)]$ and $Tx_{enable}[e_{A,B}]$ to (A_7, B_7), an address handled by A and B. Note that the transaction $Tx_{disable}$ can be added to the blockchain after T_Δ has elapsed. This ensures that during T_Δ, no user would revoke the state formed by Tx_{enable} and the corresponding $Tx^{i}_{consume}$.

We make three considerations here. First, the $Tx_{disable}$ should be created and signed before the Tx_{enable} so that users make sure that they can void Tx_{enable} if required. Second, the T_Δ used in $Tx_{disable}$ should be the same as the one used in the different Tx^{i}_{lock} so that (if required), users must choose between submitting the pair of transactions (Tx_{enable}, $Tx_{disable}$) or the transactions Tx^{i}_{lock}. Finally, it is theoretically possible that miners choose to mine Tx_{enable} and refuse to mine $Tx_{disable}$. A miner can always censor transactions and this is an interesting but orthogonal problem. Moreover, even if $Tx_{disable}$ gets censored, honest users do not directly lose their coins as Tx_{enable} sends the coins from one channel to another channel owned by the same two users.

Setup:



Tx^A_{setup}

In	Out
$Tx^*[(A_1, B_1)]; 10; \emptyset$	$(A_2, B_2); 2; \emptyset$ $(A_3, B_3); 8; \emptyset$
$\mathsf{Sig}(A_1), \mathsf{Sig}(B_1)$	

Tx^B_{setup}

In	Out
$Tx^*[(B'_1, C_1)]; 30; \emptyset$	$(B'_2, C_2); 23; \emptyset$ $(B'_3, C_3); 7; \emptyset$
$\mathsf{Sig}(B'_1), \mathsf{Sig}(C_1)$	

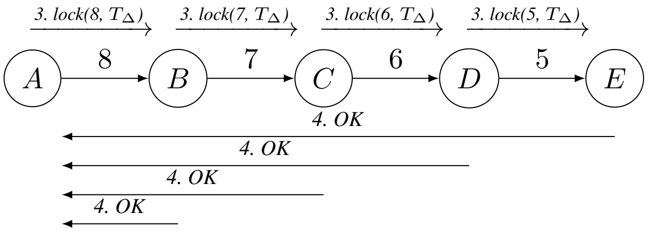
Tx^C_{setup}

In	Out
$Tx^*[(C'_1, D_1)]; 25; \emptyset$	$(C'_2, D_2); 19; \emptyset$ $(C'_3, D_3); 6; \emptyset$
$\mathsf{Sig}(C'_1), \mathsf{Sig}(D_1)$	

Tx^D_{setup}

In	Out
$Tx^*[(D'_1, E_1)]; 10; \emptyset$	$(D'_2, E_2); 5; \emptyset$ $(D'_3, E_3); 5; \emptyset$
$\mathsf{Sig}(D'_1), \mathsf{Sig}(E_1)$	

Lock:



Tx^A_{lock}

In	Out
$Tx^A_{setup}[(A_3, B_3)]; 8; \emptyset$	$(A_4, B_4); 8; \emptyset$
$\mathsf{Sig}(A_3), \mathsf{Sig}(B_3); [\mathit{elapsed}(T_\triangle)]$	

Tx^B_{lock}

In	Out
$Tx^B_{setup}[(B'_3, C_3)]; 7; \emptyset$	$(B'_4, C_4); 7; \emptyset$
$\mathsf{Sig}(B'_3), \mathsf{Sig}(C_3); [\mathit{elapsed}(T_\triangle)]$	

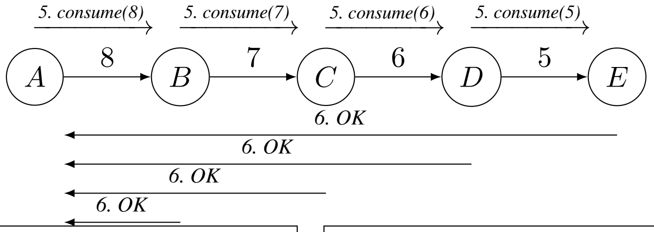
Tx^C_{lock}

In	Out
$Tx^C_{setup}[(C'_3, D_3)]; 6; \emptyset$	$(C'_4, D_4); 6; \emptyset$
$\mathsf{Sig}(C'_3), \mathsf{Sig}(D_3); [\mathit{elapsed}(T_\triangle)]$	

Tx^D_{lock}

In	Out
$Tx^D_{setup}[(D'_3, E_3)]; 5; \emptyset$	$(D'_4, E_4); 5; \emptyset$
$\mathsf{Sig}(D'_3), \mathsf{Sig}(E_3); [\mathit{elapsed}(T_\triangle)]$	

Consume:



$Tx^{A}_{consume}$

In	Out
$Tx_{enable}[(A_5, B_5)]$; 7.99; $\emptyset$ $Tx_{enable}[e_{A,B}]$; 0.01; $\emptyset$	B_6; 8; $\emptyset$
$\mathsf{Sig}(A_3), \mathsf{Sig}(B_3)$	

$Tx^{B}_{consume}$

In	Out
$Tx_{enable}[(B'_5, C_5)]$; 6.99; $\emptyset$ $Tx_{enable}[e_{B,C}]$; 0.01; $\emptyset$	C_6; 7; $\emptyset$
$\mathsf{Sig}(B'_5), \mathsf{Sig}(C_5)$	

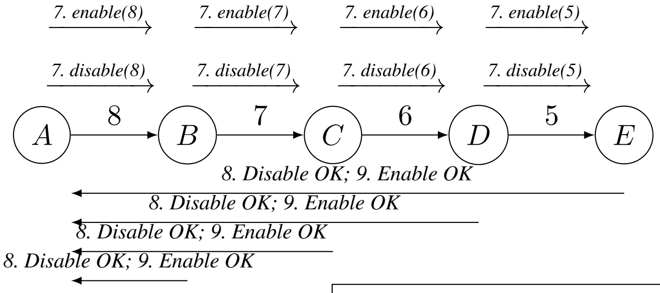
$Tx^{C}_{consume}$

In	Out
$Tx_{enable}[(C'_5, D_5)]$; 5.99; $\emptyset$ $Tx_{enable}[e_{C,D}]$; 0.01; $\emptyset$	D_6; 6; $\emptyset$
$\mathsf{Sig}(C'_5), \mathsf{Sig}(D_5)$	

$Tx^{D}_{consume}$

In	Out
$Tx_{enable}[(D'_5, E_5)]$; 4.99; $\emptyset$ $Tx_{enable}[e_{D,E}]$; 0.01; $\emptyset$	E_6; 5; $\emptyset$
$\mathsf{Sig}(D'_5), \mathsf{Sig}(E_5)$	

Finalize:



Tx_{enable}

In	Out
$Tx^{A}_{setup}[(A_3, B_3)]$; 8; $\emptyset$	(A_5, B_5); 7.99; $\emptyset$ $e_{A,B}$; 0.01; $\emptyset$
$Tx^{B}_{setup}[(B'_3, C_3)]$; 7; $\emptyset$	(B'_5, C_5); 6.99; $\emptyset$ $e_{B,C}$; 0.01; $\emptyset$
$Tx^{C}_{setup}[(C'_3, D_3)]$; 6; $\emptyset$	(C'_5, D_5); 5.99; $\emptyset$ $e_{C,D}$; 0.01; $\emptyset$
$Tx^{D}_{setup}[(D'_3, E_3)]$; 5; $\emptyset$	(D'_5, E_5); 4.99; $\emptyset$ $e_{D,E}$; 0.01; $\emptyset$
$(\mathsf{Sig}(A_3), \mathsf{Sig}(B_3)), (\mathsf{Sig}(B'_3), \mathsf{Sig}(C_3)),$ $(\mathsf{Sig}(C'_3), \mathsf{Sig}(D_3)), (\mathsf{Sig}(D'_3), \mathsf{Sig}(E_3)); \emptyset$	

$Tx_{disable}$

In	Out
$Tx_{enable}[(A_5, B_5)]$; 7.99; $\emptyset$ $Tx_{enable}[e_{A,B}]$; 0.01; $\emptyset$	(A_7, B_7); 8; $\emptyset$
$Tx_{enable}[(B'_5, C_5)]$; 6.99; $\emptyset$ $Tx_{enable}[e_{B,C}]$; 0.01; $\emptyset$	(B'_7, C_7); 7; $\emptyset$
$Tx_{enable}[(C'_5, D_5)]$; 5.99; $\emptyset$ $Tx_{enable}[e_{C,D}]$; 0.01; $\emptyset$	(C'_7, D_7); 6; $\emptyset$
$Tx_{enable}[(D'_5, E_5)]$; 4.99; $\emptyset$ $Tx_{enable}[e_{D,E}]$; 0.01; $\emptyset$	(D'_7, E_7); 5; $\emptyset$
$(\mathsf{Sig}(A_5), \mathsf{Sig}(B_5)), (\mathsf{Sig}(B'_5), \mathsf{Sig}(C_5)),$ $(\mathsf{Sig}(C'_5), \mathsf{Sig}(D_5)), (\mathsf{Sig}(D'_5), \mathsf{Sig}(E_5));$ $[elapsed(T_{\triangle})]$	

Figure 4.2: Overview for the atomic multi-channel update (AMCU) protocol (span over two pages). For each protocol step, we first illustrative example of protocol messages required to execute a call `updateState`$((c_{\langle A,B\rangle}, 8), (c_{\langle B,C\rangle}, 7), (c_{\langle C,D\rangle}, 6), (c_{\langle D,A\rangle}, 5))$. The numbers for each text represent the message sequence. Messages with the same number can be handled in parallel. For readability, we denote the success of a round by having every user send an OK message. In the actual protocol, they broadcast it to everybody else. Second, we show the transactions required to execute the illustrative example in the left. Each user U handles keys U_i with her neighbor in the right and U'_i with her neighbor in the left. For readability, we denote by $\mathsf{Sig}(U)$ the signature of the transaction by the secret key associated to U. Finally, we denote by $Tx[addr]$, the *addr* created as output in Tx.

5 The AMCU Protocol

5.1 Building Blocks

In the AMCU protocol, we require the following building blocks:

- **Timelock Mechanism:** We require a timelock mechanism available in the blockchain that enforces that a transaction is added to the blockchain only after a certain time (set as parameter of the transaction) has elapsed. In practice, virtually all cryptocurrencies implement such timelock mechanism where the time is defined as the block height. In a bit more detail, assume a transaction can be appended with a block height h. Let h^* be the current block height in the blockchain. Then, the transaction won't be included in the blockchain (i.e., will be rejected by the miners) while it holds that $h < h^*$. We refer the reader to [1] for more details. We note that as the blockchain is probabilistically extended, the block height at a certain point in time can only be estimated. Thus, this could lead to longer timeouts to safely account for the probabilistic bias. We remark that this is an orthogonal problem common to many blockchain applications (even in Ethereum-based ones) that rely on the same time management mechanism.
- **Digital Signature Scheme:** A digital signature scheme is a tuple of algorithms $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ defined as follows. $\mathsf{sk}, \mathsf{vk} \leftarrow \mathsf{Gen}(1^\lambda)$ takes as input the security parameter 1^λ and returns a public of signing and verification keys $(\mathsf{sk}, \mathsf{vk})$. $\sigma \leftarrow \mathsf{Sign}(\mathsf{sk}, m)$ takes as input the signing key sk and a message m and returns a signature σ. Finally, $\{1, 0\} \leftarrow \mathsf{Verify}(\mathsf{vk}, m, \sigma)$ takes as input a verification key vk, a message m and a signature σ and returns 1 if σ is a valid signature on message m created with the signing key corresponding to vk. Otherwise, it returns 0. We refer the reader to [10] for the security definition in the UC framework.
- **MIMO Transactions:** A MIMO transaction supports multiple addresses as input and several addresses as output. Such a transaction is valid on-chain if the following conditions hold: (i) each input address has been previously funded with a certain the amount of coins I_i; (ii) Let O_j the amount of coins set to the output j, then $\sum_i I_i = \sum_j O_j$; (iii) The complete transaction is signed with the signing keys associated to each input address. MIMO transactions are available in virtually all cryptocurrencies today.

5.2 Formal Description of the Protocol

We formalize our PCN$^+$ protocol in the ($\mathcal{F}_{smt}$,$\mathcal{F}_{syn}$,$\mathcal{L}$)-hybrid model, as detailed in Figure 5.1, where $\mathcal{F}_{syn}$ and $\mathcal{F}_{smt}$ are taken from Canetti [8] and $\mathcal{L}$ is defined in Appendix B. We choose this definition of $\mathcal{L}$ over other existing ones [14, 15, 22, 23] because it models the timeout functionality, which is a key operation in our protocol. For a given session identifier sid, we will also use sid_n as a shorthand for (sid, n). We assume a well-ordering on the set of participants which can, for example, be realized by lexicographical order of their public keys and denote the first participant, who also acts as a leader, by v_0. The set of all participants is denoted by $\mathcal{V}$. For readability, we assume that every user is able to compute a transaction identifier from the transaction content itself.

The `openChannel` and `closeChannel` work as defined for other PCNs. Thus, we focus on the description of `updateChannel`. Here, step 1 and 2 ensure that all participants want to participate in the protocol and if so, they create their corresponding Tx_{setup} and Tx_{lock} in step 3. Then, the coordinator sends an unsigned version of the Tx_{enable} and $Tx_{disable}$ to all participants, who sign the $Tx_{disable}$ first. This ensures that they have the fallback mechanism signed before they enable the transfer of coins in Tx_{enable}. Similarly, protocol participants sign their corresponding $Tx_{consume}$ in step 5. As before, this ensures that every participant can send the coins from Tx_{enable} to the corresponding receiver if the protocol is successful. Finally, steps 6 and 7 finalize the protocol by exchanging the signatures for the Tx_{enable} and advancing the round for a new execution of `updateChannel`. The error cases are explained in Section 5.4.

openChannel$(\mathsf{sid}, v_j, \beta_i, \beta_j, \sigma_i, t)$**:**

The caller v_i creates a transaction $Tx_1 := (\{(\{v_i\}, \mathsf{txid}_i), \{v_j\}, \mathsf{txid}_j\}, \{(\{v_i, v_j\}, \beta_i + \beta_j)\}, 0)$, calculates its transaction id txid_1 as well as $Tx_2 := (\{(\{v_i, v_j\}, \mathsf{txid}_1)\}, \{(\{v_i\}, \beta_i), (\{v_j\}, \beta_j)\}, 0)$, signs both and forwards $\mathsf{Sig}(Tx_2)$ to v_j. Upon receiving the signature v_j signs Tx_1 and Tx_2 and sends both signatures to v_i. Finally v_i signs Tx_1 and sends the signature to v_j. Finally v_i sends $(\mathsf{sid}, \underline{\textsf{commit-transfer}}, \{(\{v_i\}, \mathsf{txid}_i), \{v_j\}, \mathsf{txid}_j\}, \{(\{v_i, v_j\}, \beta_i + \beta_j)\}, 0, \{\sigma_i, \sigma_j\})$ to $\mathcal{L}$.

closeChannel$(\mathsf{sid}, c_{\langle v_i, v_j\rangle}, \sigma_{i,j})$**:**

Party v_i uses the stored σ_j to send $(\mathsf{sid}, \underline{\textsf{commit-transfer}}, \{(\{v_i, \mathcal{P}_j\}, \mathsf{txid}_1)\}, \{(\{v_i\}, \beta_i), (\{v_j\}, \beta_j)\}, 0, \{\sigma_i, \sigma_j\})$ to $\mathcal{L}$.

updateState$(\mathsf{sid}, \{(c_{\langle v_i, v_j\rangle}, \delta_{i,j})\}_{i,j \in [1 \ldots n]})$**:**

Let $\mathcal{E} := \{(c_{\langle v_{2i-1}, v'_{2i}\rangle}, \delta_i)\}_{i \in [1 \ldots n]}$ be the set of all updates, $\mathcal{V}$ the set of all users affected by the state update and v_0 the node initiating the `updateState` protocol.

Main Protocol:

1. v_0 sends the message $(\mathsf{sid}, \underline{\textsf{init-update}}, \mathcal{E})$ to all parties in $\mathcal{V}$.
2. All parties v_i receive $(\mathsf{sid}, \underline{\textsf{init-update}}, \mathcal{E})$ and validate $\mathcal{E}$ and decide whether to continue with the protocol. In case they do not continue they send $(\mathsf{sid}, \underline{\textsf{reject-update}})$ to all $v_i \in \mathcal{V}$.
3. Then, for all $(c_{\langle v_i, v_j\rangle}, \delta_{i,j}) \in \mathcal{E}$, the participants v_i and v_j create $Tx_{setup} := (\{(\{v_i, v_j\}, \mathsf{txid}_1)\}, \{(\{v_i\}, \delta_{i,j}), (\{v_i\}, \beta_i - \delta_{i,j}), (\{v_j\}, \beta_j)\}, 0)$ and exchange signatures. Then create $Tx_{lock} := (\{(\{v_i\}, \mathsf{txid}_{Tx_{setup}})\}, \{(\{v_i\}, \delta_{i,j})\}, t + \Delta)$, where β_i and β_j are set to the current state of the channel and t the current time sign it and exchange their signatures and send $(\mathsf{sid}, \underline{\textsf{accept-update}})$ to v_0 if all of them succeed and $(\mathsf{sid}, \underline{\textsf{reject-update}})$ to all $v_i \in \mathcal{V}$ otherwise.
4. Upon receiving $(\mathsf{sid}, \underline{\textsf{accept-update}})$ from all participants, v_0 sends $(\mathsf{sid}, \underline{\textsf{transactions-update}, \mathsf{Gen}_{Tx_{enable}}(v_0, \mathcal{E})}, \mathsf{Gen}_{Tx_{disable}}(v_0, \mathcal{E}, \mathsf{txid}_{Tx_{enable}}))$ to all $v_i \in \mathcal{V}$. Then v_i create $\sigma_{Tx_{disable}} := \mathsf{Sig}(Tx_{disable})$ and sends $(\mathsf{sid}, \underline{\textsf{disable-update}}, \sigma_{Tx_{disable}})$ to all other parties $v_j \in \mathcal{V}$.
5. After receiving all $(\mathsf{sid}, \underline{\textsf{disable-update}}, \sigma_{Tx_{disable}})$, for each $(c_{\langle v_i, v_j\rangle}, \delta_{i,j}) \in \mathcal{E}$, v_i and v_j create $Tx^{i,j}_{consume} := (\{(\{v_i\}, \mathsf{txid}_{Tx_{setup}}), (\{v_j\}, \mathsf{txid}_{Tx_{enable}})\}, \{(\{v_j\}, \delta_{i,j})\}, t + \Delta)$ and exchange signatures.
6. All parties v_i create $\sigma_{Tx_{enable}} := \mathsf{Sig}(Tx_{enable})$ and send $(\mathsf{sid}, \underline{\textsf{enable-update}}, \sigma_{Tx_{enable}})$ to all other parties $v_j \in \mathcal{V}$.
7. Once they received all $(\mathsf{sid}, \underline{\textsf{enable-update}}, \sigma_{Tx_{enable}})$, all parties advance the round with $\mathcal{F}_{syn}$.

Error Cases:

1. If the protocol aborts before the party v_i has created $(\mathsf{sid}, \underline{\textsf{get-transfer}}, \mathsf{txid}_{Tx_{enable}})$, the party aborts the protocol
2. If a party receives $(\mathsf{sid}', \underline{\textsf{notify-transfer}}, \mathsf{txid}_{Tx_{enable}})$ from $\mathcal{L}$
   - if $Tx_{disable}$ is already valid send $(\mathsf{sid}, \underline{\textsf{commit-transfer}}, Tx_{disable}, \{\sigma^k_{Tx_{disable}}\}_{k \in \mathcal{V}})$ to $\mathcal{L}$
   - else send $(\mathsf{sid}, \underline{\textsf{commit-transfer}}, Tx^{i,j}_{consume}, \{\sigma^k_{Tx_{consume}}\}_{k \in \{i,j\}})$ for all channels to $\mathcal{L}$

$\mathsf{Gen}_{Tx_{enable}}(v, \mathcal{E})$:

Let $\mathcal{V}^{in} := \{(v, \cdot)\}$, $\mathcal{V}^{out} := \{(\{v_j\}, \epsilon) | (c_{\langle v_i, v_j\rangle}, \delta_{i,j}) \in \mathcal{E}\}$ where ϵ is the minimum value supported by $\mathcal{L}$ and return $(\mathcal{V}^{in}, \mathcal{V}^{out}, 0)$

$\mathsf{Gen}_{Tx_{disable}}(v, \mathcal{E}, \mathsf{txid}_{Tx_{enable}})$:

Let $\mathcal{V}^{in} := \{(\{v_j\}, \mathsf{txid}_{Tx_{enable}}) | (c_{\langle v_i, v_j\rangle}, \delta_{i,j}) \in \mathcal{E}\}$, $\mathcal{V}^{out} := \{v\}$ and return $(\mathcal{V}^{in}, \mathcal{V}^{out}, t + \Delta)$

Figure 5.1: AMCU protocol

5.3 Discussion

Reducing the Memory Overhead. As presented so far, the AMCU protocol requires that users store several transactions off-chain as part of the final state. For instance, for the success case, user A needs to store Tx^A_{setup}, Tx_{enable} and $Tx^A_{consume}$ as part of the new state. The same applies for the fallback cases. We note that if channel users are honest and collaborate with each other, they can reduce the memory overhead by simplifying the required number of transactions to represent the state.

The main idea is that all three possible states contain a common transaction Tx^i_{setup} that is also the one required to trigger the rest of transactions in the state. In other words, independently of the state the protocol ends up, each user must submit first Tx^i_{setup} to the blockchain to be able to enforce the other transactions in the state. Therefore, users can replace the three transactions in the state by a single one that represents the same distribution of coins, after having invalidated Tx^i_{setup}. For instance, assume the protocol execution depicted in Figure 4.2. Further assume that the protocol ends up with each user holding a valid state as defined for the *success*. Here, A and B must hold Tx^A_{setup}, $Tx^A_{consume}$ and Tx_{enable} as the state information regarding their shared channel. Now, if they collaborate, they can revoke Tx^A_{setup} using the standard mechanism already implemented in the Lightning Network [32]. This mechanism allows users to atomically replace that revocation information through another transaction. This transaction should contain then the same outcome as if Tx^A_{setup}, $Tx^A_{consume}$ and Tx_{enable} are enforced on-chain. In our example, this transaction should transfer 10 coins from (A_1, B_1) and distribute them as 8 coins to B_6 and 2 coins to (A_2, B_2).

Here, we have used the example of A and B, but the rest of users can perform analogous operations. Moreover, the same technique can be applied to the state resulting from the other protocol outcomes.

Accountability. The AMCU sacrifices strong privacy guarantees such as relationship anonymity [22] to achieve not only atomicity and reduced collateral but also a notion of accountability. In particular, if in any of the protocol phases one of the users reports a failure instead of success, the protocol allows the blaming user to provide a proof of misbehavior. In a nutshell, provided that all users have agreed on the set of addresses composing the channels set as protocol inputs, the steps of the protocol are deterministically defined. Thus, at each step a user can blame the channel counterparty if it does not provide the signature for the transaction created at that phase. Note that the counterparty can also show that she was falsely blamed by actually providing the missing signature. In this case, the protocol can continue to the following phase.

5.4 Security Analysis

The security of AMCU is established by the following theorem. We postpone the security proof to Appendix A. We note that in Section 3, we have discussed how the ideal functionality $\mathcal{F}^+_{pcn}$ achieves atomicity and value privacy. Here, Theorem 5.1 shows that AMCU UC-realizes $\mathcal{F}^+_{pcn}$. Thus, this demonstrates that AMCU provides both atomicity and value privacy.

Theorem 5.1. *If the signature scheme is EU-CMA secure [16], then AMCU UC-realizes $\mathcal{F}^+_{pcn}$ in the ($\mathcal{F}_{smt}$,$\mathcal{F}_{syn}$,)-hybrid-model.*

We now give an intuition on how AMCU achieves atomicity and value privacy.

Atomicity. AMCU aims at enforcing the following invariant: If the coins – held at one of the channels – are sent to the intended receiver (i.e., the $Tx_{consume}$ is ready to be pushed on the blockchain), then all the other senders should be in the condition of pushing their $Tx_{consume}$ on the blockchain too. Notice that parties may push such transactions on the blockchain, but in an ideal case they will not since the whole protocol is supposed to run offchain.

Let us now illustrate how this invariant is enforced by considering the most significant execution cases. First, assume that some participants reach step 3 ($\{Tx^i_{lock}\}$ are signed) but some other participant aborts. In this case, each Tx^i_{lock} transfers the coins from one channel to another fresh channel owned by the same two participants and the coins become available again. Second, some participants exchange the signatures for $\{TxC^i\}$, $Tx_{disable}$, and Tx_{enable} in this order (steps 4 to 6). Here there are three possibilities: (i) some of them publishes Tx_{enable} before $T_\triangle$ has expired. In this case, everybody can use their corresponding $Tx^i_{consume}$ to send the coins to their intended receivers, thereby successfully completing the protocol; (ii) some participant publishes Tx_{enable} after $T_\triangle$. In this case, everybody can publish $Tx_{disable}$ to go back to the initial coin distribution; and (iii) some user publishes Tx^i_{lock} after $T_\triangle$. In this case, Tx_{enable} is invalid as the referred inputs have been spent and each other participant can use their Tx^j_{lock} to get the coins back into a fresh channel.

Value Privacy. AMCU must achieve the following invariant: For a successful execution of `updateChannel`, the transaction values must be known only to protocol participants. This is easy to see as `updateChannel` is a peer-to-peer protocol executed only among protocol participants on secret, authenticated channels (i.e., no auxiliary third party is involved).

6 Evaluation

In this section, we evaluate the performance of AMCU. Here, we denote by n the number of protocol participants and m the number payment channels.

Implementation-level Optimizations. At each phase, each pair of users can create the signatures over the corresponding transaction independently from other users. Moreover, the setup, lock and consume phases can be performed in parallel as they create transactions that cannot be enforced (i.e., $Tx^i_{consume}$) or transactions that result in a safe fallback state (i.e., Tx^i_{setup} and Tx^i_{lock}). Finally, details about Tx_{enable} and $Tx_{disable}$ can be exchanged in parallel. However, they have to be signed sequentially to ensure that every user gets $Tx_{disable}$ (i.e., fallback mechanism), before Tx_{enable} is signed.

Number of transactions. Let us assume a transaction on m channels. AMCU requires two transactions independently on the number of channels (i.e., Tx_{enable} and $Tx_{disable}$). Moreover, it requires m setup transactions, m lock transactions, and m consume transactions. We note, however, that these $3 \cdot m + 2$ transactions are handled off-chain, thus they do not impose any on-chain overhead.

Number of rounds. Our protocol requires 3 synchronization rounds. First, each party must check that all others received the expected signatures on the Tx^i_{setup}, Tx^i_{lock}, and $Tx^i_{consume}$. Second, parties must synchronize to get the signatures over the $Tx_{disable}$. Finally, in the last round, they jointly sign the transaction Tx_{enable} to finalize the protocol. In summary, AMCU requires a constant (i.e., 3) number of rounds, independently on the number of parties.

Communication overhead. First, for each channel $c_{\langle v_i, v_j\rangle}$ both v_i and v_j sign Tx^i_{setup}, Tx^i_{lock} as well as $Tx^i_{consume}$ and exchange the signatures. Moreover, each user signs $Tx_{disable}$ and transmits the signature. Then, for each channel $c_{\langle v_i, v_j\rangle}$ Finally, each user transmits an additional signature over the transaction Tx_{enable}. Thus, AMCU requires the exchange of $2n + 6m$ signatures for n users and m channels in total. Moreover, as a constant number of signatures are included in every message, the communication overhead is almost only limited by the network latency.

Computation time. The AMCU protocol does not require any costly cryptography. In particular, it requires that each user verifies locally the signatures for the involved transactions. Moreover, each user must compute three signatures per channel and two extra signatures independent of the number of her channels. These are also simple computations than can be executed in negligible time even with commodity hardware.

Comparison with LN. While the LN requires m transactions (i.e., an HTLC transaction per channel), AMCU requires $3m + 2$ transactions. However, while the $3m$ transactions can be handled in parallel (see implementation-level optimizations), the m transactions are inherently sequential in the LN. In fact, the LN requires $2n$ synchronization rounds among channel counterparties while AMCU requires only 3 rounds, independently of the number of channels in the path. Regarding communication overhead, LN requires $2m$ messages while AMCU requires $2n + 6m$. As before, while AMCU requires more messages, the overall protocol execution time is similar as the $6m$ messages can be handled in parallel.

7 Applications

Payment Channel Networks (PCN). The cornerstone of a PCN consists of its capability of perform multi-hop payments, that is, a payment between a sender and a receiver that do not have an opened channel between them, but rather are connected through a path of opened payment channels. We can use AMCU to design the first Bitcoin-compatible PCN with constant collateral as follows. Assume a pair of sender s and receiver r that want to carry out a payment through a path of intermediate users $v_1, \ldots, v_n$. Further assume that s wants to send β coins to r and that each v_i charges a fee of γ_i. Such payment can be carried out in AMCU as a call to `updateState` of the form: $\texttt{updateState}(\{(c_{\langle s,v_1\rangle}, \beta + \sum_{i\in[1,n]} \gamma_i), (c_{\langle v_1,v_2\rangle}, \beta + \sum_{i\in[2,n]} \gamma_i), \ldots, (c_{\langle v_n,r\rangle}, \beta)\})$.

Rebalancing. Another fundamental challenge for practical PCNs consists in the *refunding of payment channels*. Repeated payment patterns in a PCN lead to depleted channels. A depleted channel is forcing two on-chain transactions per channel to top it up: (i) closing of the channel and (ii) opening of a new channel with fresh balances. Avoiding the refunding of depleted channels is not a desirable alternative either: users need to choose longer and thus more expensive (in terms of fees) routes.

We can leverage AMCU to encode the first Bitcoin-compatible rebalancing protocol with constant collateral: prior work achieved a similar result in Ethereum [14, 15, 20], but it was an open question whether or not the same could be done in Bitcoin-compatible blockchains. Assume that users A, B, C have jointly agreed in rebalancing 20 coins in the loop $A \rightarrow B \rightarrow C$. This implies that the three users need to commit to the following rebalancing state: $c_{\langle A,B\rangle} := [-20, 10, 100]$, $c_{\langle B,C\rangle} := [-30, -5, 70]$ and $c_{\langle A,C\rangle} := [-150, 60, 85]$. Such a rebalancing can be then carried out in AMCU as a call to `updateState` of the form: $\texttt{updateState}(\{(c_{\langle A,B\rangle}, 20), (c_{\langle B,C\rangle}, 20), (c_{\langle A,C\rangle}, -20)\})$.

Crowdfunding. Assume a set of users $v_1, \ldots, v_n$ that jointly want to fund another receiver user R. In such setting, crowdfunding consists of a multi-payment operation where each sender v_i sends an amount β_i of coins to R so that $\sum_i \beta_i$ is the funding amount expected by the receiver. For such a protocol, multi-payment atomicity is highly desirable as it ensures that either every user v_i actually pays the expected β_i or each user gets her coins back.

Assume that each sender v_i has a direct payment channel to the receiver. If a sender j is connected through a path to the receiver instead, our protocol can be trivially extended by including all channels in the path from v_j to R. In such setting, a `updateState` is a crowdfunding operation among a set of users $v_1, \ldots, v_n$ for a receiver R if it is of the form $\texttt{updateState}(\{(c_{\langle v_i,R\rangle}, \beta_i)\}_{i\in[1,n]})$.

8 Related Work

The severe scalability issues present in virtually all current cryptocurrencies have motivated a wide range of proposals for PCNs from both in academia and industry. In this section, we situate AMCU in the landscape

	Required blockchain	Value Privacy	Atomicity	Collateral	Functionality
Multi-HTLC [22]	Bitcoin	Yes	Yes	staggered	PCN
AMHL [23]	Bitcoin	Yes	Yes	staggered	PCN
Lightning Network [32]	Bitcoin	Yes	No	staggered	PCN
Perun [14, 15]	Ethereum	Yes	Yes	constant	Generic
Sprites [26]	Ethereum	No	Yes	constant	PCN
Raiden [7]	Ethereum	No	Yes	constant	PCN
Revive [20]	Ethereum	No	Yes	-	Rebalancing
BOLT [17]	ZCash	Yes	Yes	-	Payment hub
AMCU	Bitcoin	Yes	Yes	constant	Generic

Table 1: Comparison among state-of-the-art in the literature for PCN protocols.

of the state-of-the-art (see Table 1). We consider four comparison points among the considered off-chain protocols that we have grouped by the required underlying blockchain.

First, we consider privacy in terms of value privacy. Value privacy is not achieved by some of the Ethereum-based approaches. This is due to the fact that they rely on a smart contract that handles the payment on behalf of the users for each of the channels in a path. In AMCU, value privacy is achieved as the protocol is executed in a peer-to-peer fashion among the protocol participants.

Second, we study atomicity. All protocols achieve it, except for the Lightning Network, which is vulnerable to a Wormhole attack [23], which is fixed in the AMHL construction and in AMCU, as all participants are aware of all channels used in the protocol due to the use of the MIMO transaction. The only systems achieving atomicity with a constant collateral, however, were up to now only those based on Ethereum, and it was conjectured that the same was not possible in Bitcoin-compatible PCNs [26]. We refute this conjecture by presenting the first Bitcoin-compatible offchain payment system with constant collateral. AMCU shows thus that generic applications possible today in Ethereum-based solutions like Perun, can potentially be deployed in cryptocurrencies with restricted scripting language.

Third, we consider the collateral. In particular, we note two possible scenarios: (i) staggered, where each channel in the payment path requires to hold coins for a time period longer that the one in the next channel; (ii) constant, where the time that coins are required to be locked is the same at all channels in the path. Constant collateral has the clear advantage in practice of reducing the amount of time that coins are locked in the PCN, which can then be faster reused for other payments. It also mitigates the attacker power for griefing attacks. Ethereum-based systems can include atomic payments within the contract code and thus can have a reduced collateral, while it was not known how to provide similar guarantees in Bitcoin. AMCU allows for multi-hop payments with a constant collateral.

Finally, we compare the functionality provided by each alternative. Most of the considered protocols have been tailored to offer a PCN functionality. Revive is an off-chain payment system that provides rebalancing operation built-in. Moreover, BOLT is tailored to payment hubs (i.e., payments with a single intermediary) and it is unclear how to extend it to support multi-hop payments. Although Perun [14] initially provided a PCN functionality, its extension [15] shows that it is possible to leverage Turing-complete language to build generic applications. AMCU also offers a generic `updateState` protocol that can be used to encode different protocols compatible with Bitcoin, eliminating the requirement for a Turing-complete language. We have shown for instance how to leverage `updateState` to encode a payment, a rebalancing operation and a crowdfunding operation.

9 Conclusions

In this work, we presented AMCU, the first multi-channel update protocol for PCNs on cryptocurrencies with restricted scripting that achieves constant collateral. We define channel updates in terms of an ideal functionality and prove our protocol secure in the Universal Composability framework. We further show how AMCU mitigates the griefing attack in PCNs and, at the same time, enables the design of a large class of applications of practical interest, such as rebalancing procedures, crowd-funding, and more.

As a future work, we intend to explore cryptographic techniques to strengthen the privacy of individual channel updates with respect to the other protocol parties. Furthermore, it would be interesting to formalize and analyze the various forms of accountability provided by the current PCN constructions, formally exploring the connection between atomicity, accountability, and privacy.

Acknowledgements

This work has been partially supported by the the European Research Council (ERC) under the European Union's Horizon 2020 research (grant agreement 771527-BROWSEC); by Netidee through the projectEtherTrust (grant agreement 2158) and PROFET (grant agreement P31621); by the Austrian Research Promotion Agency through the Bridge-1 project PR4DLT (grant agreement 13808694); by COMET K1 SBA, ABC; by Chaincode Labs; by the Austrian Science Fund (FWF) through the Lisa Meitner program; by the German research foundation (DFG) through the collaborative research center 1223; by the German Federal Ministry of Education and Research (BMBF) through the project PROMISE (16KIS0763); and by the state of Bavaria at the Nüremberg Campus of Technology (NCT). NCT is a research cooperation between the Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU) and the Technische Hochschule Nürnberg Georg Simon Ohm (THN).

References

[1] Bitcoin protocol documentation. https://en.bitcoin.it/wiki/Protocol_documentation.

[2] Bitcoin script wiki. https://en.bitcoin.it/wiki/Script.

[3] C-lightning network. https://github.com/ElementsProject/lightning.

[4] Coinmarketcap. Website. https://coinmarketcap.com/currencies/bitcoin.

[5] Eclair network. https://github.com/ACINQ/eclair.

[6] Lightning network daemon. Github repository. https://github.com/lightningnetwork/lnd.

[7] Raiden network. https://raiden.network/.

[8] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2000. http://eprint.iacr.org/2000/067.

[9] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science*, pages 136–145. IEEE Computer Society Press, October 2001.

[10] Ran Canetti. Universally composable signatures, certification and authentication. Cryptology ePrint Archive, Report 2003/239, 2003. `https://eprint.iacr.org/2003/239`.

[11] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In Salil P. Vadhan, editor, *TCC 2007: 4th Theory of Cryptography Conference*, volume 4392 of *Lecture Notes in Computer Science*, pages 61–85. Springer, Heidelberg, February 2007.

[12] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed E. Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, Dawn Song, and Roger Wattenhofer. On scaling decentralized blockchains - (A position paper). In *Financial Cryptography and Data Security - FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, Christ Church, Barbados, February 26, 2016, Revised Selected Papers*, pages 106–125, 2016.

[13] Christian Decker. Eltoo: A Simple Layer2 Protocol for Bitcoin. pages 1–24.

[14] S. Dziembowski, L. Eckey, S. Faust, and D. Malinowski. Perun: Virtual payment hubs over cryptocurrencies. In *2019 2019 IEEE Symposium on Security and Privacy (SP)*, volume 00, pages 311–328.

[15] Stefan Dziembowski, Sebastian Faust, and Kristina Hostakova. Foundations of State Channel Networks. Technical Report 320, 2018.

[16] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, April 1988.

[17] Matthew Green and Ian Miers. Bolt: Anonymous payment channels for decentralized currencies. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 17: 24th Conference on Computer and Communications Security*, pages 473–489. ACM Press, October / November 2017.

[18] Ethan Heilman, Leen Alshenibr, Foteini Baldimtsi, Alessandra Scafuro, and Sharon Goldberg. TumbleBit: An Untrusted Bitcoin-Compatible Anonymous Payment Hub. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*, 2017.

[19] Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. Universally composable synchronous computation. In Amit Sahai, editor, *TCC 2013: 10th Theory of Cryptography Conference*, volume 7785 of *Lecture Notes in Computer Science*, pages 477–498. Springer, Heidelberg, March 2013.

[20] Rami Khalil and Arthur Gervais. Revive: Rebalancing off-blockchain payment networks. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 17: 24th Conference on Computer and Communications Security*, pages 439–453. ACM Press, October / November 2017.

[21] Giulio Malavolta, Pedro Moreno-Sanchez, Aniket Kate, and Matteo Maffei. SilentWhispers: Enforcing security and privacy in decentralized credit networks. In *ISOC Network and Distributed System Security Symposium – NDSS 2017*. The Internet Society, February / March 2017.

[22] Giulio Malavolta, Pedro Moreno-Sanchez, Aniket Kate, Matteo Maffei, and Srivatsan Ravi. Concurrency and privacy with payment-channel networks. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 17: 24th Conference on Computer and Communications Security*, pages 455–471. ACM Press, October / November 2017.

[23] Giulio Malavolta, Pedro Moreno-Sanchez, Clara Schneidewind, Aniket Kate, and Matteo Maffei. Privacy-preserving Multi-hop locks for blockchain scalability and interoperability. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24 - February 27, 2019*, 2019.

[24] Patrick McCorry, Surya Bakshi, Iddo Bentov, Andrew Miller, and Sarah Meiklejohn. Pisa: Arbitration Outsourcing for State Channels. Technical Report 582, 2018.

[25] Patrick Mccorry, Malte Möser, Siamak F. Shahandasti, and Feng Hao. Towards Bitcoin Payment Networks. In *Proceedings, Part I, of the 21st Australasian Conference on Information Security and Privacy - Volume 9722*, pages 57–76, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[26] Andrew Miller, Iddo Bentov, Ranjit Kumaresan, Christopher Cordi, and Patrick McCorry. Sprites and State Channels: Payment Networks that Go Faster than Lightning. *arXiv:1702.05812 [cs]*, February 2017.

[27] Pedro Moreno-Sanchez, Aniket Kate, Matteo Maffei, and Kim Pecina. Privacy preserving payments in credit networks: Enabling trust with privacy in online marketplaces. In *ISOC Network and Distributed System Security Symposium – NDSS 2015*. The Internet Society, February 2015.

[28] Pedro Moreno-Sanchez, Navin Modi, Raghuvir Songhela, Aniket Kate, and Sonia Fahmy. Mind Your Credit: Assessing the Health of the Ripple Credit Network. In *Proceedings of the 2018 World Wide Web Conference*, WWW '18, pages 329–338, Republic and Canton of Geneva, Switzerland, 2018. International World Wide Web Conferences Steering Committee.

[29] Pedro Moreno-Sanchez, Tim Ruffing, and Aniket Kate. PathShuffle: Credit Mixing and Anonymous Payments for Ripple. *PoPETs*, 2017(3):110, 2017.

[30] Pedro Moreno-Sanchez, Muhammad Bilal Zafar, and Aniket Kate. Listening to Whispers of Ripple: Linking Wallets and Deanonymizing Transactions in the Ripple Network. *Proceedings on Privacy Enhancing Technologies*, 2016(4):436–453, October 2016.

[31] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. page 9.

[32] Joseph Poon and Thaddeus Dryja. The Bitcoin Lightning Network: Scalable off-chain instant payments. pages 1–59.

[33] Stefanie Roos, Pedro Moreno-Sanchez, Aniket Kate, and Ian Goldberg. Settling payments fast and private: Efficient decentralized routing for path-based transactions. In *ISOC Network and Distributed System Security Symposium – NDSS 2018*. The Internet Society, February 2018.

[34] Daira Hopwood Sean Bowe. Hashed time-locked contract transactions. Bitcoin Improvement Proposal. `https://github.com/bitcoin/bips/blob/master/bip-0199.mediawiki`.

[35] Manny Trillo. Stress test prepares visanet for the most wonderful time of the year. http://www.visa.com/blogarchives/us/2013/10/10/stress-test-prepares-visanet-for-the-most-wonderful-time-of-the-year/index.html, 2013. Accessed: 2017-08-07.

[36] Shira Werman and Aviv Zohar. Avoiding deadlocks in payment channel networks. In Joaquin Garcia-Alfaro, Jordi Herrera-Joancomartí, Giovanni Livraga, and Ruben Rios, editors, *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pages 175–187, Cham, 2018. Springer International Publishing.

A Security of AMCU protocol

Our proof will proceed by describing a simulator $\mathcal{S}$ that interacts with the ideal functionality $\mathcal{F}^{+}_{pcn}$ and emulates an indistinguishable protocol execution for an adversary $\mathcal{A}$ against π^{+}_{pcn}. As a visual clue we use pine green to mark messages in the ideal world and blue violet to mark messages in the real world.

We focus on the `updateState` method. The security `openChannel` and `closeChannel` follows along the same lines.

The simulator starts by forwarding all corruption requests to the functionality. It also maintains a consistent set of signing keys for the simulated honest users so it can sign transactions on their behalf. If it receives any invalid message from $\mathcal{A}$ it aborts the execution.

Triggered by honest user. If the update was triggered by an honest user, $\mathcal{S}$ continues as follows. First, $\mathcal{S}$ receives $(\mathsf{sid}, \underline{\mathsf{setup\text{-}query}}, \mathcal{E})$ from $\mathcal{F}^{+}_{pcn}$. It then sends $(\mathsf{sid}, \underline{\mathsf{init\text{-}update}}, \mathcal{E})$ on behalf of that honest user to $\mathcal{A}$. If $\mathcal{A}$ responds with $(\mathsf{sid}, \underline{\mathsf{reject\text{-}update}})$, it sends $(\mathsf{sid}, \perp)$ to the functionality as response to $\underline{\mathsf{setup\text{-}query}}$ and aborts the simulation, otherwise it sends $(\mathsf{sid}, \top)$. $\mathcal{S}$ then waits for the $(\mathsf{sid}, v, \underline{\mathsf{setup\text{-}success}})$ message from $\mathcal{F}^{+}_{pcn}$ and creates Tx_{setup} and Tx_{lock} on behalf of honest users neighboring compromised users and then simulates the exchange of signatures with the adversary and sends $(\mathsf{sid}, \perp)$ as response to $\underline{\mathsf{lock\text{-}query}}$ to $\mathcal{F}^{+}_{pcn}$ if any of these fail.

$\mathcal{S}$ creates $(\mathsf{sid}, \underline{\mathsf{transactions\text{-}update}}, \mathsf{Gen}_{Tx_{enable}}(v_0, \mathcal{E}), \mathsf{Gen}_{Tx_{disable}}(v_0, \mathcal{E}, \mathsf{txid}_{Tx_{enable}}))$ and sends it to the adversary on behalf of v_0 and collects the $(\mathsf{sid}, \underline{\mathsf{disable\text{-}update}}, \sigma_{Tx_{disable}})$ responses from the adversary. If $\mathcal{A}$ does not send the response for all compromised users, $\mathcal{S}$ sends $(\mathsf{sid}, \perp)$ as response to $\underline{\mathsf{lock\text{-}query}}$ to $\mathcal{F}^{+}_{pcn}$ and aborts and sends $(\mathsf{sid}, \top)$ otherwise. $\mathcal{S}$ then simulates the following interaction for any channel shared between compromised and honest participants: Create $Tx^{i,j}_{consume}$ and simulate the exchanged signatures with the adversary on $\mathsf{txid}_{Tx_{consume}}$. If any of the simulations fail send $(\mathsf{sid}, \perp)$ to $\mathcal{F}^{+}_{pcn}$ as response to $\underline{\mathsf{consume\text{-}query}}$ and abort, otherwise send $(\mathsf{sid}, \top)$.

Next, $\mathcal{S}$ simulates the creation of signatures on Tx_{enable} for all honest users and sends them to $\mathcal{A}$. If $\mathcal{A}$ produces signatures on Tx_{enable} for all corrupted parties, $\mathcal{S}$ sends $(\mathsf{sid}, \top)$ as response to $\underline{\mathsf{enable\text{-}query}}$ and $(\mathsf{sid}, \perp)$ otherwise. Finally advances the round with $\mathcal{F}_{syn}$.

Triggered by $\mathcal{A}$. Instead of receiving $(\mathsf{sid}, \underline{\mathsf{setup\text{-}query}}, \mathcal{E})$ from $\mathcal{F}^{+}_{pcn}$, $\mathcal{S}$ receives $(\mathsf{sid}, \underline{\mathsf{init\text{-}update}}, \mathcal{E})$ from $\mathcal{A}$, sends $(\mathsf{sid}, \underline{\mathsf{state\text{-}update}}, \{(c_{\langle v_{2i-1}, v_{2i}\rangle}, \delta_i)\}_{i \in [1 \dots n]})$ to $\mathcal{F}^{+}_{pcn}$ and receives $(\mathsf{sid}, \underline{\mathsf{setup\text{-}query}}, \{(c_{\langle v_{2i-1}, v_{2i}\rangle}, \delta_i)\}_{i \in [1 \dots n]})$. If $\mathcal{A}$ sends $(\mathsf{sid}, \underline{\mathsf{reject\text{-}update}})$ for any compromised user, it sends $(\mathsf{sid}, \perp)$ to the functionality and aborts the simulation, otherwise it sends $(\mathsf{sid}, \top)$. Then it creates Tx_{setup} and Tx_{lock} on behalf of honest users neighboring compromised users and simulates the exchange of signatures with the adversary. If any of these fail, it sends $(\mathsf{sid}, \perp)$ as response to $\underline{\mathsf{lock\text{-}query}}$ and $(\mathsf{sid}, \top)$ otherwise.

$\mathcal{S}$ then receives $(\mathsf{sid}, \underline{\mathsf{transactions\text{-}update}}, \mathsf{Gen}_{Tx_{enable}}(v_0, \mathcal{E}), \mathsf{Gen}_{Tx_{disable}}(v_0, \mathcal{E}, \mathsf{txid}_{Tx_{enable}}))$ from the adversary and responds with with $(\mathsf{sid}, \underline{\mathsf{disable\text{-}update}}, \sigma_{Tx_{disable}})$. If $\mathcal{A}$ does not send the response for all compromised users $\mathcal{S}$ sends $(\mathsf{sid}, \bot)$ as response to $\underline{\mathsf{lock\text{-}query}}$ to $\mathcal{F}^{+}_{pcn}$ and abort, otherwise send $(\mathsf{sid}, \top)$. Next, $\mathcal{S}$ simulates the following interaction for any channel shared between compromised and honest participants: Create $Tx^{i,j}_{consume}$ and simulate the exchange signatures with the adversary on $\mathsf{txid}_{Tx_{consume}}$. If any of the simulations fail send $(\mathsf{sid}, \bot)$ to $\mathcal{F}^{+}_{pcn}$, as response to $\underline{\mathsf{consume\text{-}query}}$ and abort. Otherwise send $(\mathsf{sid}, \top)$.

$\mathcal{S}$ then simulates the creation of signatures on Tx_{enable} for all honest users and sends them to $\mathcal{A}$. If $\mathcal{A}$ produces signatures on Tx_{enable} for all corrupted parties, $\mathcal{S}$ sends $(\mathsf{sid}, \top)$ as response to $\underline{\mathsf{enable\text{-}query}}$ and $(\mathsf{sid}, \bot)$ otherwise. Finally, advance the round with $\mathcal{F}_{syn}$.

B Ledger Functionality

In this section, we describe the ledger functionality $\mathcal{L}$ that serves to model a blockchain. We use then $\mathcal{L}$ to interact with other ideal functionalities to model PCNs.

Notation. We denote by $\mathbb{B}$ a set of tuples of the form $((\mathcal{V}, \mathsf{txid}), \beta)$ where $\mathcal{V}$ is a set of addresses (e.g., a multi-sig defining a channel) that were created at transaction txid. Then, β denotes the amount of coins that are held in the address $\mathcal{V}$. Moreover, we denote by T the current timestamp in the ledger.

Assumptions. We assume that $\mathbb{B}$ and T are publicly available.

Ledger $\mathcal{L}$

Init

Upon receiving a message $(\mathsf{sid}, \underline{\textsf{init}})$, set $\mathbb{B} = \emptyset$, $\mathbb{P} = \emptyset$ and $T = 0$.

Create account

Upon receiving a message $(\mathsf{sid}, \underline{\textsf{create-acc}}, \mathcal{V}, \beta)$ insert the tuple $((\mathcal{V}, \cdot), \beta,)$ in $\mathbb{B}$. Update $T := T + 1$.

Commit transfer

Assume the reception of a message $(\mathsf{sid}, \underline{\textsf{commit-transfer}}, \{(\mathcal{V}_i, \mathsf{txid}_i)\}_{i \in [1,n]}, \{(\mathcal{V}_j, \beta_j)\}, t, \boldsymbol{\sigma} := \{\sigma_i\}_{i \in [1,n]})$

1. Let $\{((\mathcal{V}_i, \mathsf{txid}_i), \beta_i, t_i)\}$ be the set of tuples in $\mathbb{B}$ corresponding to $\{(\mathcal{V}_i, \mathsf{txid}_i)\}_{i \in [1,n]}$. Then, $\mathcal{L}$ checks the following conditions:
    - $\sum_i \beta_i = \sum_j \beta_j$
    - For all signer groups $\mathcal{V}_j$ and signers $v_j \in \mathcal{V}_j$ there exists a signature $\sigma_j \in \boldsymbol{\sigma}$
    - $t < T$
2. If any of the previous conditions is not satisfied, $\mathcal{L}$ returns the message $(\mathsf{sid}, \perp)$. Otherwise, $\mathcal{L}$ removes the entries in $\{(\mathcal{V}_i, \mathsf{txid}_i, \beta_i\}$ from $\mathbb{B}$ and for each $\mathcal{V}_j$, it inserts a new entry of the form $(\mathcal{V}_j, \mathsf{txid}, \beta_j)$. where txid is the transaction id of the commited transaction.
3. Finally, send $(\mathsf{sid}, \underline{\textsf{notify-transfer}}, \mathsf{txid})$ to all users and update $T := T + 1$.

Figure B.1: Ledger functionality $\mathcal{L}$