# Multi-Hop Locks for Secure, Privacy-Preserving and Interoperable Payment-Channel Networks*

Giulio Malavolta†
Friedrich-Alexander-University Erlangen-Nürnberg
malavolta@cs.fau.de

Pedro Moreno-Sanchez†
Purdue University
pmorenos@purdue.edu

Clara Schneidewind
TU Wien
clara.schneidewind@tuwien.ac.at

Aniket Kate
Purdue University
aniket@purdue.edu

Matteo Maffei
TU Wien
matteo.maffei@tuwien.ac.at

**Abstract**

Tremendous growth in the cryptocurrency usage is exposing the inherent scalabilty issues with the permissionless blockchain technology. Among few alternatives, *payment-channel networks* (PCNs) have emerged as the most popular and practically deployed solution to overcome the scalability issues, allowing the bulk of payments between any two users to be carried out off-chain. Unfortunately, as reported in the literature and further demonstrated in this paper, current PCNs do not provide meaningful security and privacy guarantees.

In this work, we lay the foundations for the design of secure and privacy-preserving PCNs. For that, we formally define multi-hop locks, a novel cryptographic primitive that serves as a cornerstone for the design of secure and privacy-preserving PCNs, and design several provably secure cryptographic instantiations that make multi-hop locks compatible with the vast majority of cryptocurrencies. In particular, we show that (partial) homomorphic one-way functions suffice to construct multi-hop locks for PCNs supporting a script language (e.g., Bitcoin and Ethereum), and offer two constructions based on Schnorr and ECDSA that allow for the development of PCNs even *without scripts*. Further multi-hop locks constitute a generic primitive whose usefulness goes beyond regular PCNs and use those to realize atomic swaps and interoperable PCNs. Finally, our performance evaluation on a commodity machine finds that multi-hop locks operations can be performed in less than 100 milliseconds and require less than 500 bytes, even in the worst case. This shows the practicality of our approach towards enhancing security, privacy, interoperability, and scalability of today's cryptocurrencies.

## 1 Introduction

Cryptocurrencies are raising in popularity and are playing an increasing role in the worldwide financial ecosystem. Currently, just Bitcoin [40] has a market cap of 168 trillion USD and a growing number of addresses and payments. In fact, the number of Bitcoin addresses and transactions grew approximately by 30% in 2017, reaching a peak of more than $850,000$ unique addresses and more than $420,000$ transactions per day in Dec'17 [5]. This unimaginable striking demand has faced however strong scalability issues [21], which go well beyond the rapidly increasing size of the blockchain. For instance, the permissionless nature of the consensus algorithm used in Bitcoin today limits the transaction rate to tens of transactions per second, whereas other payment networks such as Visa support peaks of up to 47,000 transactions per second.

Among the various proposals to solve the scalability issue [23,24,35,44], *payment-channels* have emerged as the most widely deployed solution in practice. In a nutshell, the idea is that two users open a payment channel by uploading onto the blockchain a single transaction, which is meant to lock their bitcoins in a deposit secured by a Bitcoin smart contract. These users can then perform several payments between each other without uploading any further transaction, by simply agreeing on the new deposit balance. A transaction is required only at the end in order to close the payment channel and unlock the final deposit's balances, thereby drastically reducing the load on the blockchain. Further research in this area has proposed

*This is a draft (revision May 17, 2018)
†Both authors contributed equally and are considered to be co-first authors.

the concept of *payment-channel network* [44] (PCN), where two users not sharing a payment channel can still pay each other using a path of open channels between them. Unfortunately, as we discuss below in detail, current PCNs fall short of providing adequate security, privacy, and interoperability guarantees.

## 1.1 State-of-the-art in PCNs

Several practical deployments of PCNs exist today [2, 8, 11]. These practical efforts are based on a common reference description of the Lightning Network [9] aiming at interoperability. Unfortunately, this proposal is understudied in terms of its security and privacy guarantees.

PCNs have attracted plenty of attention also from academia [28,31,36,39]. Malavolta et al. [36] proposed a secure and privacy-preserving protocol for multi-hop payments. However, this requires non-trivial amount of data (i.e., around 5 MB) to be exchanged between the users in the payment path. Moreover, this solution requires a Hash Time-Lock Contract (HTLC) to be available in the cryptocurrency.

Green and Miers presented a hub-based privacy-preserving payment for PCNs [28]. This is based on primitives only available in Zcash and it cannot be seamlessly deployed in Bitcoin. Moreover, this approach is limited to paths with up to 3 users and the extension to support paths of arbitrary length remains an open problem.

Poelstra [42] introduced the notion of Scriptless Scripts. In a nutshell, he described the possibility of implementing the two-phase commit protocol used in the Lightning Network in a manner that no longer requires the existence of smart contracts in the cryptocurrency. Instead, he proposes to leverage the signature scheme used already in cryptocurrencies not only to validate a transaction, but also to validate it depending on a given condition. Although interesting, current proposals [43] lack formal security and privacy treatments and are based only on the Schnorr signature scheme, therefore being incompatible with cryptocurrencies like Bitcoin.

In summary, the existing proposals are neither generically applicable nor interoperable, since they rely on specific features (e.g., contracts) of individual cryptocurrencies. Furthermore, the theoretical foundations of PCNs are not properly understood: For instance, it is not clear how many rounds of communication a secure PCN requires, nor what security actually means.

## 1.2 Our contributions

In this work, we lay the theoretical foundations of PCNs and present the first interoperable, secure, and privacy-preserving cryptographic construction for PCNs. Specifically,

- We introduce a novel cryptographic primitive called *multi-hop locks*, whose security and privacy properties are defined as an ideal functionality in the UC model [20] in order to inherit the underlying composability guarantees (Section 3).
- We show how to realize multi-hop locks in a script-based setting as well as in a scriptless one[1] (Section 4). In particular, we demonstrate that (partially) homomorphic operations suffice to build any script-based multi-hop lock. Furthermore, we show how to realize multi-hop locks even without resorting to specific scripts. This approach is of special interest because it is more generic and it reduces the transaction size, and, consequently, the blockchain load. We give two concrete constructions based on Schnorr and ECDSA signatures, solving a long-standing problem in the literature [43]. This makes multi-hop locks compatible with the vast majority of cryptocurrencies (including Bitcoin and Ethereum). In fact, multi-hop locks are being implemented right now in the Lightning Network [1].
- We implemented our cryptographic constructions and show that they require at most 60 milliseconds to be computed and a communication overhead of less than 500 bytes in the worst case (Section 5). These results demonstrate that multi-hop locks are practical and ready to be deployed.
- We demonstrate that the usefulness of multi-hop locks goes beyond regular PCNs, showing how to leverage them to design atomic swaps and cross-currency PCNs (Section 6).

[1] In this work, we denote by *scriptless script* a modified version of a digital signature scheme so that a signature can only be created faithfully fulfilling a cryptograhpic condition [43]. The resulting signature is verifiable following the unmodified digital signature scheme. When applied to script-based systems like Bitcoin or Ethereum, they are accompanied by core scripts (e.g., script to verify the signature itself).

# 2 Background

In this section, we provide the required background on payment channel networks.

## 2.1 Payment Channels

A payment channel allows two users to exchange bitcoins without committing every single payment to the Bitcoin blockchain. For that, users first publish an on-chain transaction to deposit bitcoins into a multi-signature address controlled by both users. Such deposit also guarantees that all bitcoins are refunded at a mutually agreed time if the channel expires. Users can then perform off-chain payments by adjusting the balance of the deposit in favor of the payee. When no more off-chain payments are needed (or the capacity of the payment channel is exhausted), the payment channel is closed with a *closing* transaction included in the blockchain. This transaction sends the deposited bitcoins to each user according the most recent balance in the payment channel. We refer the reader to [23, 24, 37, 44] for further details.

## 2.2 A Payment Channel Network (PCN)

A PCN can be represented as a directed graph $\mathbb{G} = (\mathbb{V}, \mathbb{E})$, where the set $\mathbb{V}$ of vertices represents the Bitcoin accounts and the set $\mathbb{E}$ of weighted edges represents the payment channels. Every vertex $U \in \mathbb{V}$ has associated a non-negative number that denotes the fee it charges for forwarding payments. The weight on a directed edge $(U_1, U_2) \in \mathbb{E}$ denotes the amount of remaining bitcoins that $U_1$ can pay to $U_2$.

A PCN is used to perform off-chain payments between two users with no direct payment channel between them but rather connected by a path of open payment channels. For that, assume that $S$ wants to pay $\alpha$ bitcoins to $R$, which is reachable through a path of the form $S \to U_1 \to \ldots \to U_n \to R$. For their payment to be successful, every link must have a capacity $\gamma_i \geq \alpha'_i$, where $\alpha'_i = \alpha - \sum_{j=1}^{i-1} \mathit{fee}(U_j)$ (i.e., the initial payment value minus the fees charged by intermediate users in the path). If the payment is successful, edges from $S$ to $R$ are decreased by $\alpha'_i$. Importantly, to ensure that $R$ receives exactly $\alpha$ bitcoins, $S$ must start the payment with a value $\alpha^* = \alpha + \sum_{j=1}^{n} \mathit{fee}(U_j)$. We refer the reader to [28, 36, 37, 44] for further details.

The concepts of payment channels and PCNs have already attracted attention from academia [24, 28, 29, 33, 36, 37, 46]. In practice, the Lightning Network (LN) [9, 44] has emerged as the most prominent example. Currently, there exist several independent implementations of the LN for Bitcoin [2,3,8,11]. Moreover, the LN is also considered as a scalability solution in other blockchain-based payment systems such as Ethereum [10].

## 2.3 Multi-Hop Payments in the LN

A fundamental property that multi-hop payments have to fulfill is *atomicity*: Either the capacity of all channels in the path is updated or none of the channels is changed. Partial updates can lead to coin losses by the user. For instance, a user could pay certain bitcoins to the next user in the path but never receive the corresponding bitcoins from the previous neighbour. The LN tackles this challenge by relying on a smart contract called *Hash Time-Lock Contract* (HTLC) [44].

This contract locks $x$ bitcoins that can be released only if the contract's condition is fulfilled. The contract is defined in terms of a hash value $y := H(R)$ where $R$ is chosen uniformly at random, the amount of bitcoins $x$, and a timeout $t$, as follows:

**HTLC (Alice, Bob, $y$, $x$, $t$):**
1. If Bob produces the condition $R^*$ such that $H(R^*) = y$ before $t$ days,[2]Alice pays Bob $x$ bitcoins.
2. If $t$ days elapse, Alice gets back $x$ bitcoins.

We depict in Figure 1 an illustrative example of the use of HTLC in a payment. For ease of exposition, we assume that every user charges a fee of one bitcoin and the payment amount is 10 bitcoins. In this payment, Edward first sets up the payment by creating a random value $R$ and sending $H(R)$ to Alice. Then, the commitment round starts by Alice. She first sets on hold 13 bitcoins who is then followed by every intermediate user by setting on hold the received amount minus his/her own fee. After Dave sets 10 coins on hold with Edward, the latter knows that the corresponding payment amount is on hold at each channel and he can start the releasing phase. For that, he reveals the value $R$ to Dave allowing them to fulfill the

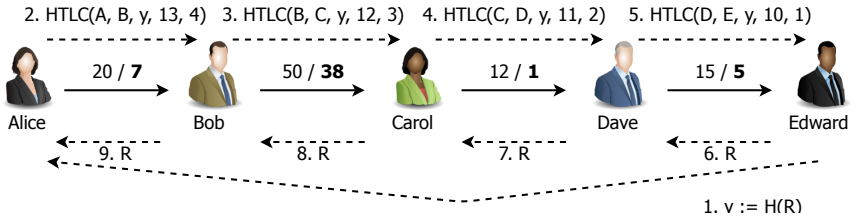


Figure 1: Illustrative example of a payment from Alice to Edward for value 10 using HTLC contract. Non bold (bold) numbers represent the capacity of payment channels before (after) the payment from Alice to Edward. We assume all users to charge a payment fee of 1 bitcoin. First, the condition is sent from Edward to Alice. The condition is then forwarded among users in the path to hold 10 bitcoin plus the fees left to pay at each payment channel. Finally, the receiver shows $R$, releasing the held bitcoin at each payment channel.

HTLC contract and settle the new capacity at the payment channel. The value $R$ is then passed backwards in the path allowing the settlement of each payment channel in the path.

# 3 Definition

In the following we introduce a new cryptographic primitive called *multi-hop lock*. This primitive generalizes the locking mechanism used for payments in state-of-the art PCNs such as the Lightning Network. As motivated in the previous section, we model the primitive such that it allows for an initial setup phase in which the first node of the path provides the other nodes on the path with some secret (path-specific) state. Formally, a multi-hop lock is defined with respect to a universe of users $\mathbb{U}$ and it is a five-tuple of PPT algorithms and protocols $\mathbb{L} = (\mathsf{KGen}, \mathsf{Setup}, \mathsf{Lock}, \mathsf{Rel}, \mathsf{Vf})$ defined as follows:

**Definition 1** (Multi-hop lock)**.** *A* multi-hop lock $\mathbb{L} = (\mathsf{KGen}, \mathsf{Setup}, \mathsf{Lock}, \mathsf{Rel}, \mathsf{Vf})$ *consists of the following efficient algorithms:*

- $\{(\mathsf{sk}_i, \mathsf{pk}), (\mathsf{sk}_j, \mathsf{pk})\} \leftarrow \langle \mathsf{KGen}_{U_i}(1^\lambda), \mathsf{KGen}_{U_j}(1^\lambda) \rangle$ : *On input the security parameter $1^\lambda$ the key generation protocol returns a shared public key* $\mathsf{pk}$ *and a secret key* $\mathsf{sk}_i$ *($\mathsf{sk}_j$, respectively) to $U_i$ and $U_j$.*
- $\{s_0^I, \ldots, (s_n^I, k_n)\} \leftarrow \left\langle \mathsf{Setup}_{U_0}(1^\lambda, U_1, \ldots, U_n), \ldots, \mathsf{Setup}_{U_n}(1^\lambda) \right\rangle$ : *On input a set of identities $(U_1, \ldots, U_n)$ and the security parameter $1^\lambda$, the setup protocol returns, for $i \in [0, n]$, a state $s_i$ to each party $U_i$. The receiver $U_n$ additionally receives a key $k_n$.*
- $\{(\ell, s_i^R), (\ell, s_{i+1}^L)\} \leftarrow \left\langle \mathsf{Lock}_{U_i}(s_i^I, \mathsf{sk}_i, \mathsf{pk}), \mathsf{Lock}_{U_{i+1}}(s_{i+1}^I, \mathsf{sk}_{i+1}, \mathsf{pk}) \right\rangle$ : *On input two initial states $s_i^I$ and $s_{i+1}^I$, two secret keys $\mathsf{sk}_i$ and $\mathsf{sk}_{i+1}$, and a public key* $\mathsf{pk}$*, the locking protocol is executed between two parties $(U_i, U_{i+1})$ and returns a lock $\ell$ and a right state $s_i^R$ to $U_i$ and the same lock $\ell$ and a left state $s_{i+1}^L$ to $U_{i+1}$.*
- $k' \leftarrow \mathsf{Rel}(k, (s^I, s^L, s^R))$ : *On input an opening key $k$ and a set of states $(s^I, s^L, s^R)$, the release algorithm returns a new opening key $k'$.*
- $\{0, 1\} \leftarrow \mathsf{Vf}(\ell, k)$ : *On input a lock $\ell$ and a key $k$ the verification algorithm returns a bit $b \in \{0, 1\}$.*

**Correctness.** A multi-hop lock is *correct* if the verification algorithm $\mathsf{Vf}$ always accepts a honestly generated lock-key pair. For a more detailed and formal correctness definition, we refer the reader to Appendix A.

**Key Ideas.** Figure 2 illustrates the usage of the different protocols underlying the multi-hop lock primitive: We assume an initial phase where a network of users is generated using the (interactive) $\mathsf{KGen}$ protocol for establishing pairwise links between users. Consequently, all users in this network are assumed to hold a secret key and a shared public key for each link that they created. We recreate thereby the opening of payment channels that compose the PCN.

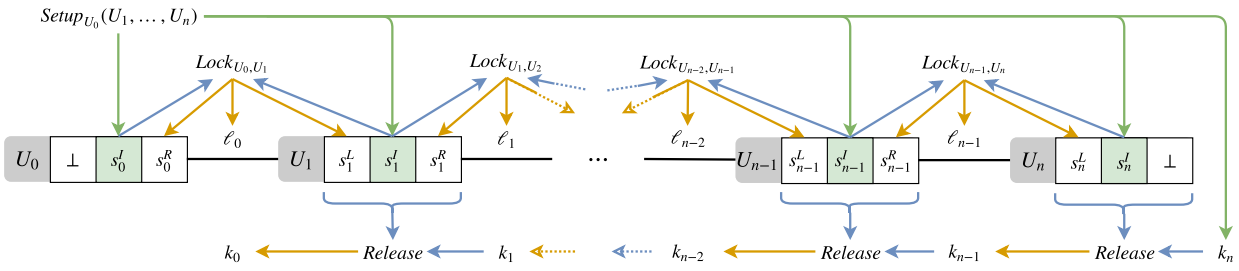


Figure 2: Illustration of the usage of the multi-hop lock primitive. It is assumed that links between the users on the path have been created upfront using the KGen algorithm and that the resulting public and secret keys are implicitly given as argument to the corresponding executions of the Lock protocol. Otherwise, the inputs (outputs) to (from) the Lock protocol and the Rel algorithm are indicated by blue (orange) arrows.

In the setup phase (depicted in green), the sender $U_0$ decides upon a path of users $(U_0, \ldots, U_n)$ and executes the Setup protocol with the nodes of the path. Each user $U_i$ on the path learns its initial state $s_i^I$ and the receiver $U_n$ additionally learns the initial opening key $k_n$. The introduction of the initial state at each intermediate user is crucial for security and privacy. Intuitively, we can use this initial state as "rerandomization factor" to ensure that locks in the same path are unlinkable for the adversary.

Next, in the locking phase, each pair of users jointly executes the pairwise Lock protocol, starting from $U_0$. Two users $U_i$ and $U_{i+1}$ run the Lock protocol on their initial states $s_i^I$ and $s_{i+1}^I$ to generate a lock $\ell_i$. The creation of this lock represents the commitment from $U_i$ to perform an application-dependent action if a cryptographic problem is solved by $U_{i+1}$. In the case of LN, this operation represents the commitment of $U_i$ to pay a certain amount of coins to $U_{i+1}$ if $U_{i+1}$ solves the cryptographic condition. Each user also learns some secret information $s_i^R$ (resp. $s_{i+1}^L$) that will be needed for releasing the lock later on. As the locks are directed, the usage of this information will differ depending on whether it was gained while creating a lock with a preceding (left) or with a subsequent (right) neighbor on the path: As $U_i$ creates a lock with its right neighbor, it learns the state $s_i^R$ denoting the state of its *right lock*. Correspondingly, $U_{i+1}$ learns the state $s_i^L$ denoting the state of its *left lock*. While these extra states are not present in the LN (i.e., every lock is based on the same cryptographic puzzle $H(R)$), having them is crucial for security. They make the releasing of different locks in the path independent from each other and therefore enable the possibility to ensure that a lock $\ell_i$ can only be released if $\ell_{i+1}$ has been released before.

Finally, after the entire path is locked, the receiver $U_n$ can use the information it received during the setup phase ($k_n$ and $s_n^I$) together with the information it learned from creating its left lock ($s_n^L$) to generate a key for releasing its left lock. In the same fashion each intermediate node can use its state (containing the initial state as well as the states from locking) to derive a valid key for its left lock from a valid key for its right lock using the Rel algorithm. This last phase resembles the opening phase of the LN where each pair of users settles the new balances for their deposit at each payment channel in the payment path.

## 3.1 Security and Privacy Definition

To model security and privacy in the presence of concurrent executions we resort to the universal composability framework from Canetti [20]. We allow thereby the composition of multi-hop locks with other application-dependent protocols while maintaining security and privacy guarantees. We model the players in our protocol as interactive Turing machines that communicate with a trusted functionality $\mathcal{F}$ via secure and authenticated channels. We model the attacker $\mathcal{A}$ as a PPT machine that corrupts a subset of users prior the execution of the interaction. Upon corruption of a user $U$, the attacker is provided with the internal state of $U$ and the incoming and outgoing communication of $U$ is routed through $\mathcal{A}$. Let $\mathsf{EXEC}_{\tau,\mathcal{A},\mathcal{E}}$ be the ensemble of the outputs of the environment $\mathcal{E}$ when interacting with the adversary $\mathcal{A}$ and parties running the protocol $\tau$ (over the random coins of all the involved machines).

**Definition 2** (Universal Composability)**.** *A protocol $\tau$ UC-realizes an ideal functionality $\mathcal{F}$ if for any PPT adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that for any environment $\mathcal{E}$ the ensembles $\mathsf{EXEC}_{\tau,\mathcal{A},\mathcal{E}}$ and $\mathsf{EXEC}_{\mathcal{F},\mathcal{S},\mathcal{E}}$ are computationally indistinguishable.*

**KeyGen($U_j, \{L, R\}$)**

Upon invocation by $U_i$:
send $(U_i, \{L, R\})$ to $U_j$
receive $b$ from $U_j$
if $b = \perp$ send $\perp$ to $U_i$ and abort
if $L$ insert $(U_i, U_j)$ into $\mathcal{U}$ and send $(U_i, U_j)$ to $U_i$
if $R$ insert $(U_j, U_i)$ into $\mathcal{U}$ and send $(U_j, U_i)$ to $U_i$

**Lock($lid$)**

Upon invocation by $U_i$:
if *getStatus*($lid$) $\neq$ Init or *getLeft*($lid$) $\neq U_i$ then abort
send ($lid$, Lock) to *getRight*($lid$)
receive $b$ from *getRight*($lid$)
if $b = \perp$ send $\perp$ to $U_i$ and abort
*updateStatus*($lid$, Lock)
send ($lid$, Lock) to $U_i$

**Setup($U_0, \ldots, U_n$)**

Upon invocation by $U_0$:
if $\forall i \in [0, n-1] : (U_i, U_{i+1}) \notin \mathcal{U}$ then abort
$\forall i \in [0, n-1]$: $lid_i \leftarrow\!\!\$\ \{0,1\}^\lambda$
insert $(lid_0, U_0, U_1, \mathsf{Init}, lid_1), (lid_{n-1}, U_{n-1}, U_n, \mathsf{Init}, \perp)$ into $\mathcal{L}$
send $(\perp, lid_0, \perp, U_1, \mathsf{Init})$ to $U_0$
send $(lid_{n-1}, \perp, U_{n-1}, \perp, \mathsf{Init})$ to $U_n$
$\forall i \in [1, n-1]$: insert $(lid_i, U_i, U_{i+1}, \mathsf{Init}, lid_{i+1})$ into $\mathcal{L}$
send $(lid_{i-1}, lid_i, U_{i-1}, U_{i+1}, \mathsf{Init})$ to $U_i$

**Release($lid$)**

Upon invocation by $U_i$:
if *getRight*($lid$) $\neq U_i$ or *getStatus*($lid$) $\neq$ Lock or
*getStatus*(*getNextLock*($lid$)) $\neq$ Rel
and *getNextLock*($lid$) $\neq \perp$ then abort
*updateStatus*($lid$, Rel)
send ($lid$, Rel) to *getLeft*($lid$)

Figure 3: Ideal functionality for multi-hop locks

**Ideal Functionality.** We formally define the ideal world functionality $\mathcal{F}$ for multi-hop locks in the following. For ease of exposition, we assume that each pair of users establishes only a single link per direction. The model can be easily extended to handle the more generic case. $\mathcal{F}$ works in interaction with a universe of users $\mathbb{U}$ and initializes two empty lists $(\mathcal{U}, \mathcal{L}) := \emptyset$, which are used to track the users and the locks, respectively. The list $\mathcal{L}$ represents a set of lock chains. The entries are of the form $(lid_i, U_i, U_{i+1}, f, lid_{i+1})$ where $lid_i$ is a lock identifier that is unique even among other lock chains in $\mathcal{L}$, $U_i$ and $U_{i+1}$ are the users connected by the lock, $f \in \{\mathsf{Init}, \mathsf{Lock}, \mathsf{Rel}\}$ is a flag that represents the status of the lock, and $lid_{i+1}$ is the identifier of the next lock in the path. For sake of better readability, we define functions operating on $\mathcal{L}$ extracting lock-specific information given the lock's identifier, such as the lock's status (*getStatus*($\cdot$)), the nodes it is connecting (*getLeft*($\cdot$), *getRight*($\cdot$)), and the next lock's identifier (*getNextLock*($\cdot$)). In addition we define an update function *updateStatus*($\cdot, \cdot$) that changes the status of lock to a new flag.

The interfaces of the functionality $\mathcal{F}$ are specified in Figure 3. The KeyGen interface allows a user to establish a link with another user (specifiying whether it wants to be the left or the right part of the link). The Setup interface allows a user $U_0$ to setup a path (starting from $U_0$) along previously established links. The Lock interface allows a user to create a lock with its right neighbor on a previously created path and finally the Release algorithm allows a user to release the lock with its left neighbor, in case that the user is either the receiver or its right lock has been released before. Internally, the locks are modelled by identifiers that are unique across all paths that have been created. Consequently, each lock identifier also identifies the path along which it was established.

## 3.2 Discussion

We discuss how the security and privacy properties for multi-hop locks are modeled by the ideal functionality.

**Atomicity.** Loosely speaking, atomicity means that every user in a path is able to release its left lock in case that his right lock was already released. This is enforced by $\mathcal{F}$ as i) it is keeping track of the chain of locks and their current status in the list $\mathcal{L}$ and ii) the Release interface of $\mathcal{F}$ allows one to release a lock $lid$ (changing the flag to Rel) if and only if $lid$ is locked and the follow-up lock (*getNextLock*($lid$)) was already released.

**Consistency.** A multi-hop lock is consistent if no attacker can release his left lock without its right lock being released before. This prevents scenarios where some multi-hop lock is released before the receiver is reached. To see why our ideal functionality models this property, observe that the Release interface allows a user to release the left lock if and only if the right lock has already been released or if the user itself is the receiver.

**Relationship Anonymity.** Relationship anonymity [14] requires that each intermediate node does not learn any information about the set of users in a multi-hop lock beyond its direct neighbors. This property is satisfied by $\mathcal{F}$ as the lock identifiers are sampled at random and during the locking phase a user only learns the identifiers of its left and right lock as well as its left and right neighbor.

# 4 Constructions

Here we describe our constructions for multi-hop locks.

## 4.1 Cryptographic Building Blocks

Throughout this work we denote by $\lambda \in \mathbb{N}^+$ the security parameter. Given a set $S$, we denote by $x \leftarrow\!\!\$\, S$ the sampling of and element uniformly at random from $S$, and we denote by $x \leftarrow A(in)$ the output of the algorithm $A$ on input $in$. We denote by $\mathsf{min}(a, b)$ the function that takes as input two integers and returns the smaller of the two. In the following we briefly recall the cryptographic building blocks of our schemes.

**Homomorphic One-Way Functions.** A function $g : \mathcal{D} \to \mathcal{R}$ is one-way if, given a random element $x \in \mathcal{R}$, it is hard to compute a $y \in \mathcal{D}$ such that $g(y) = x$. We say that a function $g$ is homomorphic if $\mathcal{D}$ and $\mathcal{R}$ define two abelian groups and for each pair $(a, b) \in \mathcal{D}^2$ it holds that $g(a \circ b) = g(a) \circ g(b)$, where $\circ$ denotes the group operation. Throughout this work we denote the corresponding arithmetic group additively.

**Commitment Scheme.** A commitment scheme $\mathsf{COM}$ consists of a commitment algorithm $(\mathsf{decom}, \mathsf{com}) \leftarrow \mathsf{Commit}(1^\lambda, m)$ and a verification algorithm $\{0, 1\} \leftarrow \mathsf{V}_{\mathsf{com}}(\mathsf{com}, \mathsf{decom}, m)$. The commitment algorithm allows a prover to commit to a message $m$ without revealing it. In a second phase, the prover can convince a verifier that the message $m$ was indeed committed by showing the unveil information $\mathsf{decom}$. The security of a commitment scheme is captured by the standard ideal functionality $\mathcal{F}_{\mathsf{com}}$ [20].

**Non-Interactive Zero-Knolwedge.** Let $R$ be an NP relation and let $L$ be the set of positive instances, i.e., $L := \{x \mid \exists w \text{ s.t. } R(x, w) = 1\}$. A non-interactive zero-knowledge proof [17] scheme $\mathsf{NIZK}$ consists of an efficient prover algorithm $\pi \leftarrow \mathsf{P}_{\mathsf{NIZK}}(w, x)$ and an efficient verifier $\{0, 1\} \leftarrow \mathsf{V}_{\mathsf{NIZK}}(x, \pi)$. A $\mathsf{NIZK}$ scheme allows the prover to convince the verifier about the existence of a witness $w$ for a certain statement $x$ without revealing any additional information. The security of a $\mathsf{NIZK}$ scheme is modelled by the following ideal functionality $\mathcal{F}_{\mathsf{NIZK}}$: On input $(\mathsf{prove}, \mathit{sid}, x, w)$ by the prover, check if $R(x, w) = 1$ and send $(\mathsf{proof}, \mathit{sid}, x)$ to the verifier if this is the case.

**Homomorphic Encryption.** One of the building blocks of our work is the additive homomorphic encryption scheme $\mathsf{HE} := (\mathsf{KGen}_{\mathsf{HE}}, \mathsf{Enc}_{\mathsf{HE}}, \mathsf{Dec}_{\mathsf{HE}})$ from Paillier [41]. The scheme supports homomorphic operation over the ciphertexts of the form $\mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}, m) \cdot \mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}, m') = \mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}, m + m')$. We assume that Paillier's encryption scheme satisfies the notion of $\mathsf{ecCPA}$ security, as defined in the work of Lindell [34].

**Schnorr Signatures.** Let $\mathbb{G}$ be an elliptic curve group of order $q$ with base point $G$ and let $H : \{0, 1\}^* \to \{0, 1\}^{|q|}$ be a collision resistant hash function (modelled as a random oracle). The key generation algorithm $\mathsf{KGen}_{\mathsf{schnorr}}(1^\lambda)$ of a Schnorr signature [47] samples some $x \leftarrow\!\!\$\, \mathbb{Z}_q$ and sets the corresponding public key as $Q := x \cdot G$. To sign a message $m$, the signing algorithm $\mathsf{Sig}_{\mathsf{schnorr}}(\mathsf{sk}, m)$ samples some $k \leftarrow\!\!\$\, \mathbb{Z}_q$, computes $e := H(Q \| k \cdot G \| m)$, sets $s := k - xe$, and returns $\sigma := (R, s)$, where $R := k \cdot G$. The verification $\mathsf{Vf}_{\mathsf{schnorr}}(\mathsf{pk}, \sigma, m)$ returns 1 if and only if $s \cdot G = R + H(Q \| R \| m) \cdot Q$. Schnorr signatures are known to be strongly unforgeable against the discrete logarithm assumption [27]. We assume the existence of a 2-party

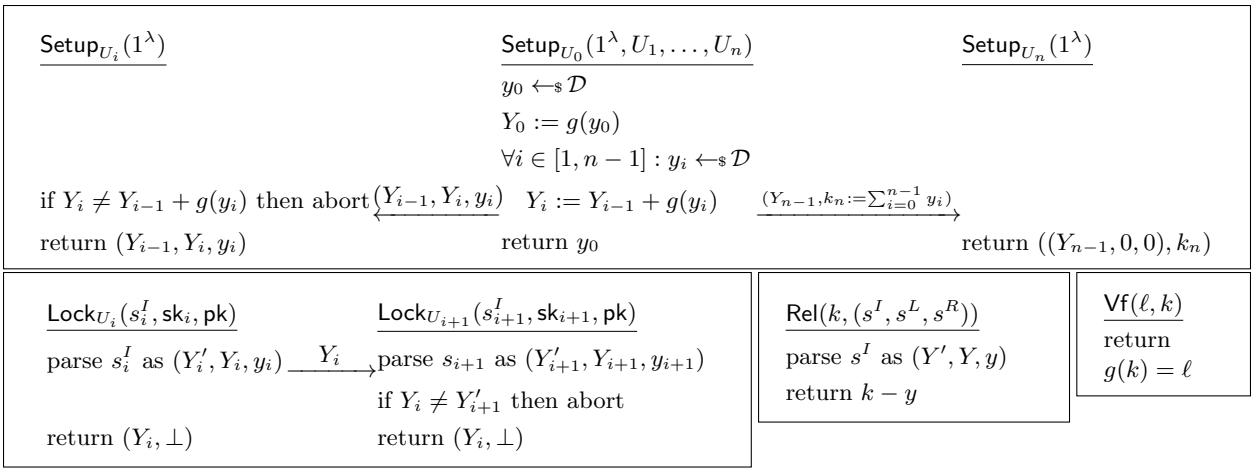


Figure 4: Algorithms and protocols for the generic construction

protocol $\Pi_{\mathsf{KGen}}^{\text{schnorr}}$ where the two players, on input $x_0$ and $x_1$, set a shared public key $Q := (x_0 + x_1) \cdot G$. Such a protocol can be realized using standard techniques [38].

**ECDSA Signatures.** Let $\mathbb{G}$ be an elliptic curve group of order $q$ with base point $G$ and let $H : \{0,1\}^* \to \{0,1\}^{|q|}$ be a collision resistant hash function. The key generation algorithm $\mathsf{KGen}_{\mathsf{ECDSA}}(1^\lambda)$ samples a private key as a random value $x \leftarrow\!\!\$\ \mathbb{Z}_q$ and sets the corresponding public key is $Q := x \cdot G$. To sign a message $m$, the signing algorithm $\mathsf{Sig}_{\mathsf{ECDSA}}(\mathsf{sk}, m)$ samples some $k \leftarrow\!\!\$\ \mathbb{Z}_q$ and computes $e := H(m)$. Let $(r_x, r_y) := R = k \cdot G$, the algorithm computes $r := r_x \bmod q$ and $s := \frac{e + rx}{k} \bmod q$. The signature consists of $(r, s)$. The verification algorithm $\mathsf{Vf}_{\mathsf{ECDSA}}(\mathsf{pk}, \sigma, m)$ recomputes $e = H(m)$ and returns 1 if and only if $(x, y) = \frac{e}{s} \cdot G + \frac{r}{s} \cdot Q$ and $r = x \bmod q$. It is a well known fact that for every valid signature $(r, s)$, also the pair $(r, -s)$ is a valid signature. To make the signature *strongly* unforgeable we augment the verification equation with a check that $s \leq \frac{q-1}{2}$. We assume the existence of an interactive protocol $\Pi_{\mathsf{KGen}}^{\text{ECDSA}}$ executed between two users where the one receives $(x_0, Q, \mathsf{sk})$, where $\mathsf{sk}$ is a Paillier secret key and $Q = x_0 \cdot x_1 \cdot G$, whereas the other obtains $(x_1, Q, \mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}, x_0 \cdot x_1))$, where $\mathsf{pk}$ is the corresponding Paillier public-key. An efficient protocol that fits these requirements has been recently proposed by Lindell [34].

**Anonymous Communication.** We assume an anonymous communication channel $\Pi_{\mathsf{anon}}$ available among peers of the network, which is modelled by the ideal functionality $\mathcal{F}_{\mathsf{anon}}$ which anonymously delivers messages to users in the network.

## 4.2 Generic Construction

An interesting question related to multi-hop locks is under which class of hard problems such a primitive exists. A generic construction using trapdoor permutation was given (implicitly) in [36]. Here we propose a scheme from any homomorphic one-way function. Examples of homomorphic one-way functions include discrete logarithm and the learning with error problem [45]. Let $g : \mathcal{D} \to \mathcal{R}$ be a homomorphic one-way function, and let $\Pi_{\mathsf{anon}}$ be an anonymous communication channel. The algorithms of our construction are given in Figure 4. Note that the key generation algorithm simply returns the identities of the users and therefore it is omitted.

In the setup algorithm, the user $U_0$ initializes the multi-hop lock by sampling $n$ values $(y_0, \ldots, y_{n-1})$ from the domain of $g$. Then it sends (via $\Pi_{\mathsf{anon}}$) a triple $(g(\sum_{j=0}^{i-1} y_j), g(\sum_{j=0}^{i} y_j), y_i)$ to each intermediate user. The intermediate user $U_i$ can then check that the triple is well formed using the homomorphic properties of $g$. Two contiguous users $U_i$ and $U_{i+1}$ can agree on the shared value of $\ell_i := Y_i = g(\sum_{j=0}^{i} y_j)$ by simply comparing the second and first element of their triple, respectively. Note that publishing a valid opening key $k$ such that $g(k) = \ell$ corresponds to inverting the one-way function $g$. The opening of the locks can be triggered by the last node in the chain $U_n$: The initial key $k_n := \sum_{i=0}^{n-1} y_i$ consists of a valid pre-image of

$\ell_{n-1} := Y_{n-1}$. As soon as the "right" lock is released, each intermediate user $U_i$ has enough information to release its "left" lock. To see this, observe that $g(k_{i+1} - y_i) = g(\sum_{j=0}^{i} y_i - y_i) = g(\sum_{j=0}^{i-1} y_i) = Y_{i-1}$. For the security of the construction, we state the following theorem. Due to space constrains, the proof is deferred to Appendix B.

**Theorem 1.** *Let $g$ be a homomorphic one-way function and let $\Pi_{\mathsf{anon}}$ be an anonymous communication channel, then the construction in Figure 4 UC-realizes the ideal functionality $\mathcal{F}$.*

**Discussion.** The generic construction presented here requires a cryptocurrency supporting scripts that define (partially) homomorphic operations. This construction is therefore of special interest in blockchain technologies such as Ethereum [7] and Hyperledger Fabric [13], where any user can freely deploy a smart contract without restrictions in the cryptographic operations available. We stress that any function with homomorphic properties are suitable to implement our construction. For instance, lattice-based functions (e.g., from the learning with errors problem) can be used for applications where post-quantum cryptography is required. However, many cryptocurrencies, leaded by Bitcoin, do not support unrestricted scripts and the deployment of generic multi-hop locks require non-trivial changes (i.e., a hard-fork). To overcome this practical challenge, we propose scriptless multi-hop locks, where digital a signature scheme that can simultaneously be used for authorization and locking.

### 4.3 Schnorr-based Construction

In the following we cast the idea of Poelstra [42] in our framework. The crux of the construction is that a lock shall consist only of a message $m$ and a public key $\mathsf{pk}$ of a given signature scheme (Schnorr in this case) and can be released only with a valid signature $\sigma$ of $m$ under $\mathsf{pk}$. As discussed before, this enables to use multi-hop locks in conjunction to cryptocurrencies that do not support custom scripts. On a high-level, our scheme resembles Poelstra's paradigm, with the crucial difference that it requires one less round of communication. The construction exploits specific homomorphic properties of Schnorr signatures, with a novel usage of the randomness, which we also leverage later to support interoperable Schnorr-ECDSA locks.

Let $\mathbb{G}$ be an elliptic curve group of order $q$ with base point $G$ and let $H : \{0,1\}^* \to \{0,1\}^{|q|}$ be a hash function. The Schnorr-based construction is formally described in Figure 5. The key generation algorithm consists of an execution of the $\Pi^{\text{schnorr}}_{\mathsf{KGen}}$ protocol. At the end of a successful run, $U_i$ receives $(x_i, \mathsf{pk})$ whereas $U_j$ obtains $(x_j, \mathsf{pk})$, where $\mathsf{pk} := (x_i + x_j) \cdot G$. The setup of a multi-hop lock is very similar to the setup of the generic construction in Section 4.2 except that the one-way function $g$ is now instantiated with discrete logarithm over elliptic curves. Each intermediate user $U_i$ receives a triple $(Y_{i-1}, Y_i, y_i)$ such that $Y_i := Y_{i-1} + y_i \cdot G$, from $\Pi_{\mathsf{anon}}$. For technical reasons, the initiator of the multi-hop lock also includes a proof of wellformedness for each $Y_i$.

Prior to the locking phase, two users $U_i$ and $U_{i+1}$ (implicitly) agree on the value $Y_i$ and on a message $m$ to be signed. Each message is assumed to be unique for each session (e.g., contains a transaction identifier). The locking algorithm consists of an "incomplete" distributed signing of $m$. First, the two parties agree on a randomly chosen element $R_0 + R_1$ using a standard coin tossing protocol, then they set the randomness of the signature to be $R := R_0 + R_1 + Y_i$. Note that at this point the parties cannot complete the signature since they do not know the discrete logarithm of $Y_i$. Instead, they jointly compute the value $s := r_0 + r_1 + e \cdot (x_0 + x_1)$ as if $Y_i$ was not part of the randomness, where $e$ is the hash of the transcript so far. The resulting $(R, s)$ is *not* a valid signature on $m$, since the additive term $y^*$ (where $y^* \cdot G = Y_i$) is missing from the computation of $s$. However, rearranging the terms, we have that $(R, s + y^*)$ is a valid signature on $m$. This implies that, once the discrete logarithm of $Y_i$ is revealed, a valid signature $m$ can be computed by $U_{i+1}$. Leveraging this observation, $U_{i+1}$ can enforce an *atomic* opening: The subsequent locking (between $U_{i+1}$ and $U_{i+2}$) is conditioned on some $Y_{i+1} = Y_i + y_{i+1} \cdot G$. This way, the opening of the right lock reveals the value $y^* + y_{i+1}$ and $U_{i+1}$ can immediately extract $y^*$ and open its left lock with a valid signature on $m$. The security of the construction is shown by the following theorem. We refer the reader to Appendix B for a full proof.

**Theorem 2.** *Let $\mathsf{COM}$ be a secure commitment scheme, let $\mathsf{NIZK}$ be a non-interactive zero knowledge proof, let $\Pi^{schnorr}_{\mathsf{KGen}}$ be a secure shared key generation protocol, and let $\Pi_{\mathsf{anon}}$ be an anonymous communication channel. If Schnorr signatures are strongly existentially unforgeable, then the construction in Figure 5 UC-realizes the ideal functionality $\mathcal{F}$.*

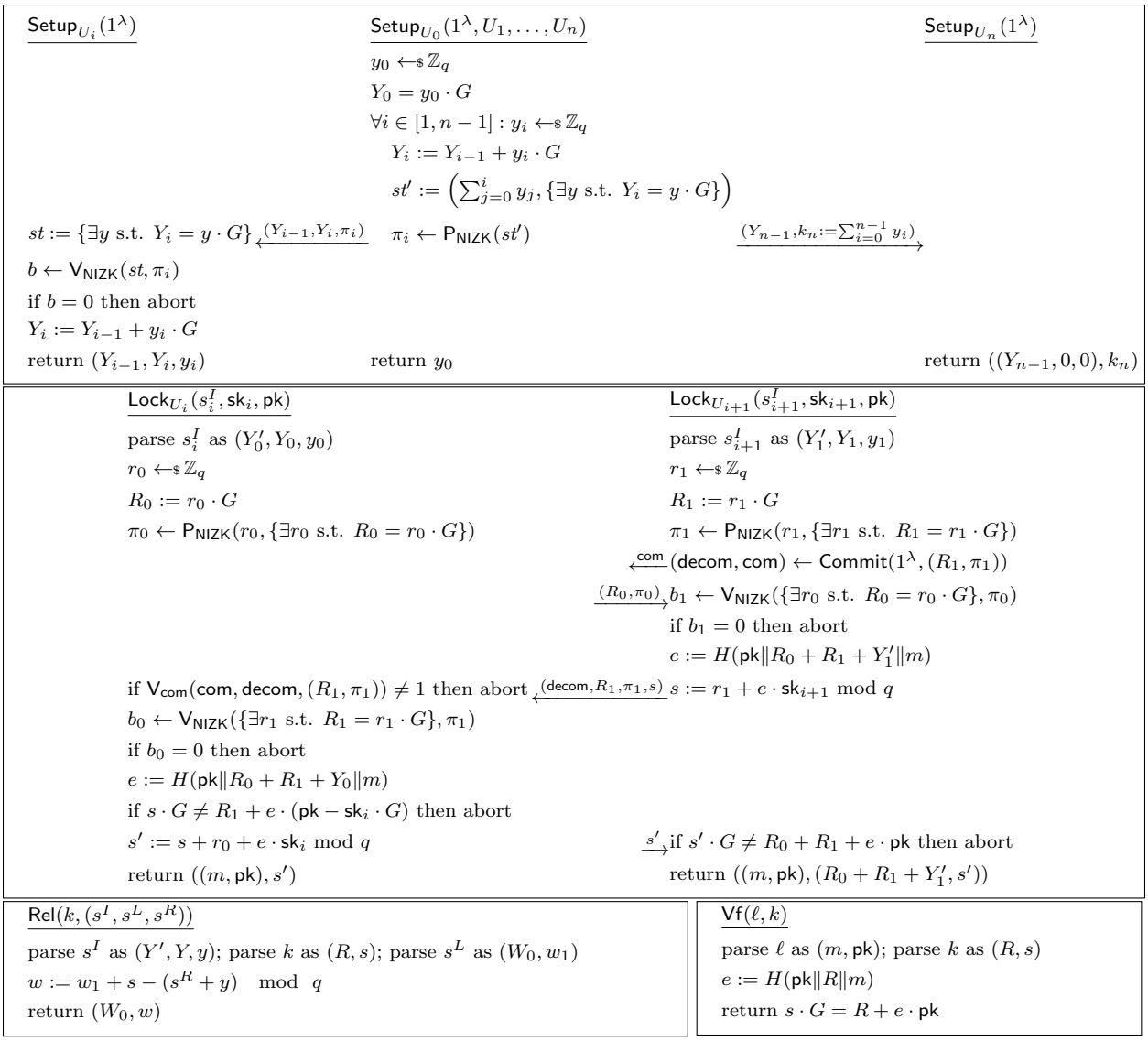


Figure 5: Algorithms and protocols for the Schnorr-based construction

**Discussion.** The Schnorr-based construction, first sketched by Poelstra and formally defined and characterized in this work, can be deployed in any cryptocurrency that uses the Schnorr digital signature scheme as authorization mechanism for transactions, independently of the script language supported. However, this construction is not compatible with those system, prominently Bitcoin, that rely on ECDSA to authorize the transactions. Therefore, an ECDSA-based scriptless multi-hop lock is desirable both from a practical prospective and it is an interesting theoretical problem whether it can be done at all. To the best of our knowledge, this was considered as an open problem [43]. The core difference is that the Schnorr-based construction exploits the linear structure of the signature, whereas the ECDSA signing algorithm completely break this linearity feature. In the following, we show how to overcome this problem, introducing our novel scriptless ECDSA-based construction for multi-hop locks.

## 4.4 ECDSA-based Construction

In the following, we present an instantiation of multi-hop locks based on ECDSA signatures: Locks are of the form $(\mathsf{pk}, m)$ and can only be opened with a valid ECDSA signature $\sigma$ on $m$ under $\mathsf{pk}$.

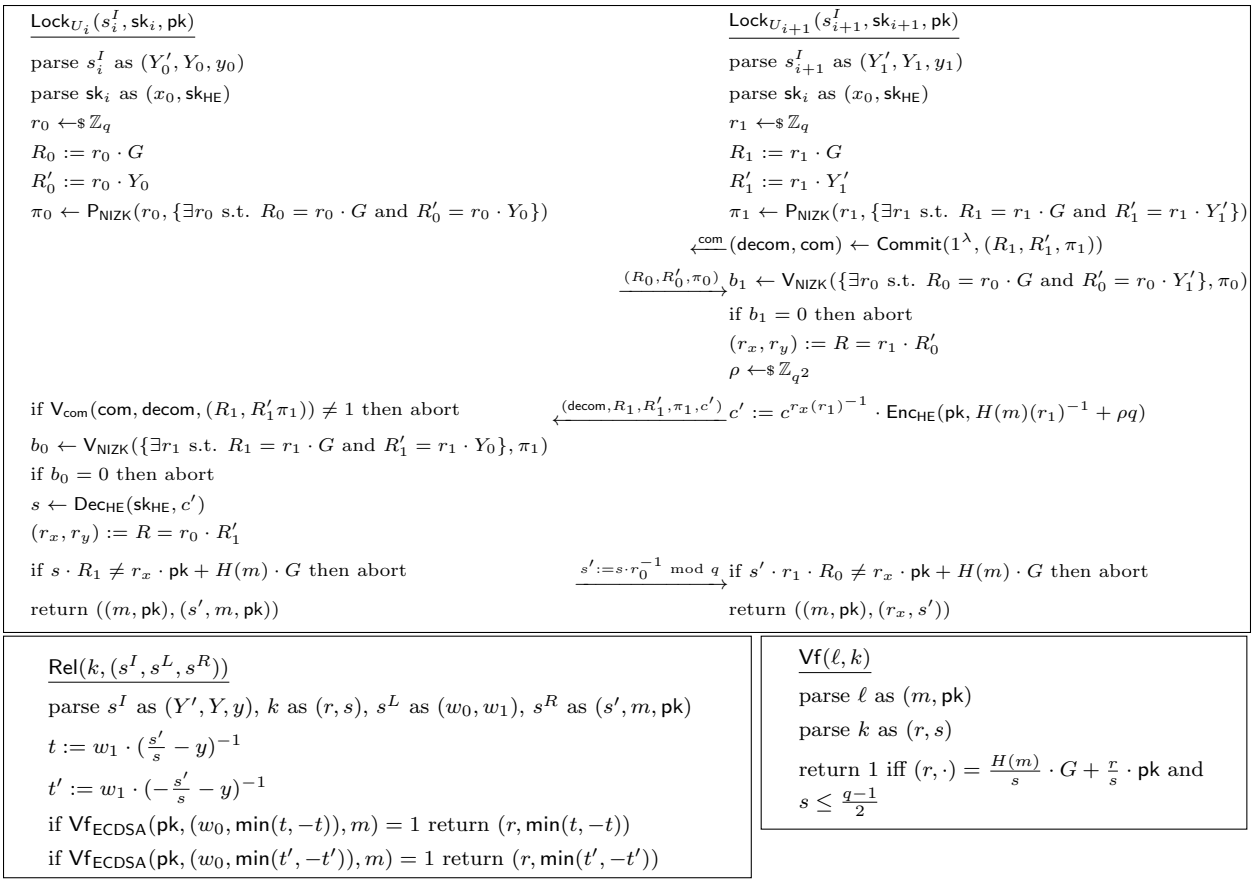


Figure 6: Algorithms and protocols for the ECDSA-based construction. The Setup protocol is as defined in Figure 5

Let $\mathbb{G}$ be an elliptic curve group of order $q$ with base point $G$ and let $H : \{0,1\}^* \to \{0,1\}^{|q|}$ be a hash function. The ECDSA-based construction is shown in Figure 6. Each pair of users $(U_i, U_j)$ generates a shared ECDSA public key $\mathsf{pk} = (x_i \cdot x_j) \cdot G$ via the $\Pi_{\mathsf{KGen}}^{\mathrm{ECDSA}}$ protocol. Additionally, $U_i$ receives his share $x_0$ and a Paillier secret key, whereas and $U_j$ receives the share $x_1$ and an encryption $c$ of $x_0 \cdot x_1$. The corresponding key generation protocol is fully described in [34]. The setup of a multi-hop lock is identical to the Schnorr-based construction and can be found in Figure 5.

The locking algorithm is initiated by two users $U_i$ and $U_{i+1}$ who agree on a message $m$ (which encodes a unique id) and on a value $Y_i := y^* \cdot G$ of unknown discrete logarithm. The two parties then run a coin tossing protocol to agree on a randomness $R = (r_0 \cdot r_1) \cdot Y_i$. When compared to the Schnorr instance, the crucial technical challenge here is that the randomnesses are composed multiplicatively due to the structure of the ECDSA signature and therefore, the trick applied in the Schnorr construction no longer applies here. $R$ is computed through a Diffie-Hellman-like protocol, where the parties exchange $r_0 \cdot Y_i$ and $r_1 \cdot Y_i$ and locally recompute $R$. As before, the shared ECDSA signature is computed by "ignoring" the term $Y_i$, since the parties are unaware of its discrete logarithm. The corresponding tuple $\left(r_x, s' := \frac{r_x \cdot (x_i \cdot x_{i+1}) + H(m)}{r_0 \cdot r_1}\right)$ is jointly computed using the encryption of $x_i \cdot x_{i+1}$ and the homomorphic properties of Paillier encryption. This effectively means that $(r_x, s') = (r_x, s^* \cdot y^*)$, where $(r_x, s^*)$ is a valid ECDSA signature on $m$. In order to check the validity of $s'$, the parties additionally need to exchange the value $R^* := (r_0 \cdot r_1) \cdot G = (y^*)^{-1} \cdot R$. The computation of $R^*$ (together with the corresponding consistency proof) is piggybacked in the coin tossing. Given $R^*$, the validity of $s'$ can be easily verified by both parties by recomputing it "in the exponent".

From the perspective of $U_{i+1}$, releasing his left lock without a key for his right lock implies solving the discrete logarithm of $Y_i$. On the converse, once the right lock is released, the value $y^* + y_{i+1}$ is revealed

(where $y_{i+1}$ is part of the state of $U_{i+1}$) and a valid signature can be computed as $\left(r_x, \frac{s'}{y^*}\right)$. The security of the construction is established by the following theorem (see Appendix B for a full proof).

**Theorem 3.** *Let* $\mathsf{COM}$ *be a secure commitment scheme, let* $\mathsf{NIZK}$ *be a non-interactive zero knowledge proof, let* $\Pi^{ECDSA}_{\mathsf{KGen}}$ *be a secure shared key generation protocol, and let* $\Pi_{\mathsf{anon}}$ *be an anonymous communication channel. If ECDSA signatures are strongly existentially unforgeable and Paillier encryption is* $\mathsf{ecCPA}$ *secure, then the construction in Figure 6 UC-realizes the ideal functionality* $\mathcal{F}$*.*

**Discussion.** The ECDSA-based multi-hop lock brings several advantages to Bitcoin apart from being fully compatible and the fact that it can be deployed today without even a soft fork. As the release information for a lock is a single signature independently of the cryptographic condition, it reduces the transaction size and therefore the associated fees and the amount of memory eventually required in the blockchain when the payment channel is closed. Moreover, it improves the fungibility of Bitcoin, a crucial property for any currency: The transaction encoding of a multi-hop lock is structurally identical to a direct payment between two persons, therefore blending conditional payments into regular payments. This set of features have attracted the attention of the Bitcoin community and our proposed solution has been started to be implemented and tested.

### 4.5 Interoperable Multi-hop Locks

We observe that, when instantiated over the same elliptic curve $\mathbb{G}$, the setup protocols of the Schnorr and ECDSA constructions are identical. This means that in principle the initiator of the lock does not need to be aware of whether each intermediate lock will be computed using the ECDSA or Schnorr method. This opens the door to interoperable multi-hop locks: Given a unified setup, the intermediate pair of users can generate locks using an arbitrary locking protocol. The resulting multi-hop lock is a chaining of (potentially) different locks and the release algorithm needs to be adjusted accordingly. For the case of ECDSA-Schnorr the user needs to extract the value $y^*$ from the right Schnorr signature $(R^*, s^*)$ and his state $s^R := s' = s^* - y^* + y_{i+1}$ and $s^I := (Y_i, Y_{i+1}, y_{i+1})$. Given $y^*$, he can factor it out of its left state $s^L = ((r, s \cdot y^*), m, \mathsf{pk})$ and recover a valid ECDSA signature.

The complementary case (Schnorr-ECDSA) is handled mirroring this algorithm. Similar techniques also apply to the generic construction, when the one-way function is instantiated appropriately (i.e., with discrete logarithm over the same curve). This flexibility in the choice of the locking algorithm immediately enables cross-currency atomic swaps and payment channels (see Section 6). The security for the interoperable multi-hop locks follows similar to the standard case.

## 5 Performance Analysis

Here we describe the implementation and evaluation of the different constructions of multi-hop locks, demonstrating their practicality and deployment readiness.

### 5.1 Implementation

We have developed a prototypical Python implementation to demonstrate the feasibility of our construction and evaluated its performance in terms of computation time, computation cost, and communication overhead. We have used the Charm library [6, 12] for the cryptographic operations. We have instantiated Schnorr and ECDSA signatures over the elliptic curve *secp256k1* (i.e., the one used in Bitcoin) and we have implemented the homomorphic one-way function with the discrete log function $g(x) := x \cdot G$ over the same curve. Zero-knowledge protocols for the knowledge of discrete logarithms have been implemented using $\Sigma$ protocols [22] and made non-interactive using the Fiat-Shamir transformation [26]. For a commitment scheme we have used SHA-256 modeled as a random oracle [15].

| | | Generic | Schnorr | ECDSA |
|---|---|---|---|---|
| Setup | Time (ms) | $0.3 \cdot n$ | $1 \cdot n$ | $1 \cdot n$ |
| | Comm (bytes) | $96 \cdot n$ | $128 \cdot n$ | $128 \cdot n$ |
| Lock | Time (ms) | – | 2 | 60 |
| | Comm (bytes) | 32 | 256 | 416 |
| Rel | Time (ms) | – | 0.002 | 0.02 |
| | Comm (bytes) | 0 | 0 | 0 |
| Vf | Time (ms) | – | 0.6 | 0.06 |
| | Comm (bytes) | 0 | 0 | 0 |
| Comp Cost (gas) | | $350849 \cdot n$ | 0 | 0 |
| Lock size (bytes) | | 32 | $32 + \|m\|$ | $32 + \|m\|$ |
| Open size (bytes) | | 32 | 64 | 64 |

Table 1: Comparison of the resources required to execute the algorithms for the different multi-hop locks. We denote by $n$ the length of the path. We denote the negligible computation times by – (e.g., single memory read). We denote the size of an application-dependent message by $|m|$ (e.g., a transaction in a payment-channel network).

## 5.2 Evaluation

We conducted our experiments on a machine with an Intel Core i7, 3.1 GHz and 8 GB RAM. In this analysis, we consider the following four algorithms: Setup, Lock, Rel, Vf. We do not investigate on KGen as we use off-the-shelf algorithms without modification. Moreover, the key generation is executed once and for all upon creating a link and thus does not affect the online performance of multi-hop locks. We refer the reader to [38] and [34] for a detailed performance evaluation of the key generation for Schnorr and ECDSA, respectively.

For each algorithm considered here, we report three measurements. First, the computation time required by the two parties sharing a multi-hop lock. Second, the cost of such computation in the Ethereum blockchain for the generic construction. We do not consider this measurement for scriptless approaches as they do not require smart contracts. Finally, we measure the communication overhead.

Additionally, in this analysis we consider the memory overhead for the application to handle the information (i.e., the lock and the opening) required by the multi-hop locks. The results of our performance analysis are summarized in Table 1.

**Computation Time.** We measure the computation time required by the users to perform the different algorithms. For the case of two-party algorithms (e.g., Setup and Lock) we consider the time for the two users together. We make two main observations: First, script-based construction based on discrete logarithm is faster than scriptless multi-hop locks, while ECDSA is slower than Schnorr among the scriptless options. Second, all the algorithms require computation time of at most one millisecond on a commodity hardware.

**Computation Cost.** We measure the computation cost in terms of the gas required by a smart contract implementing the corresponding algorithm in Ethereum. Naturally, we consider this cost only for the generic approach based on discrete logarithm. We observe that setting up the corresponding contract requires 350849 unit of gas per hop. At the time of writing, each multi-hop lock requires considerably less that 0.01 USD.

**Communication Overhead.** We measure the communication overhead as the amount of information that users need to exchange during the execution of interactive protocols, in particular, Setup and Lock. As expected, the generic construction based on discrete logarithm requires less communication overhead than scriptless constructions. Among the scriptless constructions, ECDSA requires a higher communication overhead. This higher communication overhead is mainly due to having the signing key distributed multiplicatively instead of additively and a more complex structure of the final signature when compared to Schnorr.

**Application Overhead.** We measure the overhead incurred by the application in terms of the memory required to handle application-dependent data, i.e., information defining the lock and the opening. In tune with the rest of measurements, the generic construction based on discrete logarithm requires the smallest amount of memory, both for lock and opening information. Scriptless approaches require the same amount of memory from the application.

# 6 Applications

In this section, we describe how multi-hop locks can be seamlessly applied in current blockchain applications to improve their security and privacy.

## 6.1 Payment-Channel Networks

As described in Section 2, the Lightning Network relies on the HTLC contract to perform multi-hop payments between any two users. Each hop must correctly set the HTLC and authorize it with an ECDSA transaction. The final transaction in a hop consists of the complete HTLC and two signatures, one from each of the parties sharing a payment channel, among other fields. This solution has privacy issues, as described throughout this paper.

A first approach towards a solution could consist on deploying the generic construction for multi-hop locks to inherit their security and privacy properties, at the cost of adding support for homomorphic functions in the Bitcoin scripting language. Although sound, this approach lacks deployability as adding new functionality in a cryptocurrency requires consensus from the majority of the users.

More interestingly, the scriptless ECDSA multi-hop lock construction presented in this work can be seamlessly adopted in the Lightning Network, conveying thereby several advantages. First, it eliminates the privacy issues existing in the current Lightning Network due to the use of the HTLC contract. Second, it reduces the transaction size as a single signature is required per transaction. This has the benefit of lowering the communication overhead, the transaction fees, and the blockchain memory requirements for closing a payment channel. In fact, we have received initial feedback from the Lightning Network community indicating the suitability of our ECDSA-based construction and that initial implementation and testing has been triggered.

The applicability of our proposals are not restricted to the Lighting Network or Bitcoin: There exist other payment-channel networks that could similarly take advantage of the scriptless multi-hop locks presented in this work. For instance, the Raiden Network has been presented as a payment-channel network for the scalability issue in Ethereum. The adoption of our ECDSA scriptless multi-hop locks would bring to the Raiden Network the same benefits as to the Lightning Network. Similarly, the scriptless Schnorr-based multi-hop locks could be seamlessly deployed in payment-channel networks where Schnorr is used for authorization instead of ECDSA.

## 6.2 Atomic Swaps

Assume two users $U_0$ and $U_1$ holding coins in two different cryptocurrencies that want to exchange them. An *atomic swap* protocol ensures that either the coins are swapped or the balances are untouched, i.e., the exchange must be performed atomically. Current atomic swaps protocols [18] leverage the HTLC contract to perform the swap. In a nutshell, an atomic swap can be seen as a multi-hop payment over a path of the form $(U_0, U_1, U_0)$. This approach inherits the privacy concerns of HTLC contract. Scriptless multi-hop locks enhance also this application domain with formally proven security guarantees.

Additionally, our constructions contribute to the *fungibility* of the coins, a crucial aspect in any currency and therefore also in cryptocurrencies. Current protocols rely on transactions that are clearly distinguishable from regular payments (i.e., one to one payments). In particular, atomic swaps transactions contain the HTLC contract, in contrast with regular transactions. Scriptless multi-hop locks eliminate this issue since even atomic swaps transactions only require a single signature from a public key, making them indistinguishable from regular payments. Similar arguments also apply for multi-hop payments in PCNs.

### 6.3 Interoperable PCNs

In the plethora of cryptocurrencies existing today, an interesting problem consists in performing a multi-hop payment where each link represents a payment channel defined in a different cryptocurrency. In this manner, a user with a payment channel funded in a given cryptocurrency can use it to pay to another user with a payment channel in a different cryptocurrency. Currently, the InterLedger protocol [48] tackles this problem and proposes a mechanism to perform cross-currency multi-hop payments. This protocol relies on the HTLC protocol aiming at ensuring the atomicity of the payment across different hops.

However, apart from the already discussed issues associated with HTLC, the InterLedger protocol mandates that all cryptocurrencies implement HTLC contracts. This obviously hinders the deployment of this approach. Instead, it is possible to use the different multi-hop locks constructions presented in this work in a single path (see Section 4.5), therefore expanding the domain of cross-currency multi-hop payments.

## 7 Related Work

A recent work [25] shows a protocol to compute an ECDSA signature using multi-party computation. However, it is not as efficient as the Lindell approach [34].

There exists an extensive literature proposing constructions for payment channels [4, 23, 24, 33, 44]. These works focus on a single payment channel and their extension to PCNs remain an open challenge. TumbleBit [29] and Bolt [28] support off-chain payments while achieving payment anonymity guarantees. However, the privacy guarantees of these approaches are restricted to single-hop payments and their extension to support multi-hop payments remain an open challenge.

The LN has emerged as the most promising approach for PCN in practice. Its current description [9] is being followed by several implementations [2, 3, 8, 11]. However, these implementations suffer from the privacy issues with PCNs as described in this work. Instead, we provide several constructions for multi-hop locks that can be leveraged to have secure and privacy-preserving multi-hop payments.

Malavolta et al. [36] propose a protocol for secure and privacy-preserving multi-hop payments compatible with the current LN. Their approach, however, imposes an overhead of around 5 MB for the nodes in the network, therefore, hindering its deployability. Here, we propose several efficient constructions that require only a few bytes of communication.

In the recent literature, we can find proposals for secure and privacy-preserving atomic swaps. Tesseract [16] leverages trusted hardware to perform real time cryptocurrency exchanges. The Merkleized Abstract Syntax Trees (MAST) protocol has been proposed as a privacy solution for atomic swaps [32]. However, MAST relies on scripts that are not available in the major cryptocurrencies today. Moreover, specific contracts for atomic swaps hinder the fungibility of the currency: An observer can easily differentiate between a regular payment and a payment resulting from an atomic swap.

## 8 Conclusion

In this work, we lay the foundations for security, privacy and interoperability guarantees in PCNs, proposing a novel cryptographic construction (multi-hop locks). We instantiate multi-hop locks in two settings: script-based and scriptless. In the script-base setting, as a theoretical insight we demonstrate that multi-hop locks can be realized from any (partially) homomorphic operation. In the scriptless setting, we define two constructions based on Schnorr and ECDSA respectively, thereby catering the vast majority of cryptocurrencies. Our performance evaluation shows that multi-hop locks are practical: in the worst case, running they take less than 100 milliseconds and it conveys a communication overhead of less than 500 bytes.

Moreover, we show that multi-hop locks are of interest in several applications apart from PCNs, such as atomic swaps and interoperable PCNs. The Lightning Network community has picked up our ECDSA construction and initial implementation and tests have started.

In the future, we plan to devise cryptographic instantiations of PCNs for the few cryptocurrencies that are not yet covered, most notably Monero. Furthermore, we intend to support advanced features of PCNs, such as the possibility to split dynamically payments across different paths. Finally, we would like to devise cryptographic instantiations that are post-quantum secure.

# References


[1] Lightning network developers mailing list. https://lists.linuxfoundation.org/pipermail/lightning-dev/2018-May/001244.html.

[2] c-lightning – a lightning network implementation in c. https://github.com/ElementsProject/lightning.

[3] Thunder network. https://github.com/blockchain/thunder.

[4] Bitcoin wiki: Bitcoin contract. https://en.bitcoin.it/wiki/Contract.

[5] Blockchain explorer information. https://blockchain.info/.

[6] Charm: A framework for rapidly prototyping cryptosystems. https://github.com/JHUISI/charm.

[7] Ethereum website. https://www.ethereum.org/.

[8] Lightning network daemon. https://github.com/lightningnetwork/lnd.

[9] Lightning network specifications. https://github.com/lightningnetwork/lightning-rfc.

[10] Raiden network. http://raiden.network/.

[11] A scala implementation of the lightning network. https://github.com/ACINQ/eclair.

[12] Akinyele, J. A., Garman, C., Miers, I., Pagano, M. W., Rushanan, M., Green, M., and Rubin, A. D. Charm: A framework for rapidly prototyping cryptosystems. *Journal of Cryptographic Engineering 3*, 2 (June 2013), 111–128.

[13] Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., Caro, A. D., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y., Muralidharan, S., Murthy, C., Nguyen, B., Sethi, M., Singh, G., Smith, K., Sorniotti, A., Stathakopoulou, C., Vukolic, M., Cocco, S. W., and Yellick, J. Hyperledger fabric: A distributed operating system for permissioned blockchains. *CoRR abs/1801.10228* (2018).

[14] Backes, M., Kate, A., Manoharan, P., Meiser, S., and Mohammadi, E. Anoa: A framework for analyzing anonymous communication protocols. In *Computer Security Foundations Symposium* (2013).

[15] Bellare, M., and Rogaway, P. Random oracles are practical: A paradigm for designing efficient protocols. In *Computer and Communications Security* (1993), pp. 62–73.

[16] Bentov, I., Ji, Y., Zhang, F., Li, Y., Zhao, X., Breidenbach, L., Daian, P., and Juels, A. Tesseract: Real-time cryptocurrency exchange using trusted hardware. *IACR Cryptology ePrint Archive 2017* (2017), 1153.

[17] Blum, M., Feldman, P., and Micali, S. Non-interactive zero-knowledge and its applications. In *Symposium on Theory of Computing* (1988), pp. 103–112.

[18] Bowe, S., and Hopwood, D. Hashed time-locked contract transactions. Bitcoin Improvement Proposal, Accessed May 2018. https://github.com/bitcoin/bips/blob/master/bip-0199.mediawiki.

[19] Camenisch, J., and Lysyanskaya, A. A formal treatment of onion routing. In *Annual International Cryptology Conference* (2005), pp. 169–187.

[20] Canetti, R. Universally composable security: A new paradigm for cryptographic protocols. In *"FOCS'01"*.

[21] Croman, K., Decker, C., Eyal, I., Gencer, A. E., Juels, A., Kosba, A., Miller, A., Saxena, P., Shi, E., Gün Sirer, E., Song, D., and Wattenhofer, R. On Scaling Decentralized Blockchains. In *Financial Cryptography and Data Security* (2016), pp. 106–125.

[22] Damgård, I. On $\sigma$-protocols. *Lecture Notes, University of Aarhus, Department for Computer Science* (2002).

[23] Decker, C., Russel, R., and Osuntokun, O. eltoo: A simple layer2 protocol for bitcoin. `https://blockstream.com/eltoo.pdf`.

[24] Decker, C., and Wattenhofer, R. A fast and scalable payment network with bitcoin duplex micropayment channels. In *Stabilization, Safety, and Security of Distributed Systems* (2015).

[25] Doerner, J., Kondi, Y., Lee, E., and a. shelat. Secure two-party threshold ecdsa from ecdsa assumptions. In *Symposium on Security and Privacy* (2018), pp. 595–612.

[26] Fiat, A., and Shamir, A. How to prove yourself: Practical solutions to identification and signature problems. In *Conference on the Theory and Application of Cryptographic Techniques* (1986), pp. 186–194.

[27] Goldwasser, S., Micali, S., and Rivest, R. L. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing 17*, 2 (1988), 281–308.

[28] Green, M., and Miers, I. Bolt: Anonymous payment channels for decentralized currencies. In *Computer and Communications Security* (2017).

[29] Heilman, E., Alshenibr, L., Baldimtsi, F., Scafuro, A., and Goldberg, S. TumbleBit: An untrusted bitcoin-compatible anonymous payment hub. In *Network and Distributed System Security Symposium* (2017).

[30] Jakobsson, M., and Juels, A. Millimix: Mixing in small batches. Tech. rep., DIMACS Technical report 99-33, 1999.

[31] Khalil, R., and Gervais, A. Revive: Rebalancing off-blockchain payment networks. In *Computer and Communications Security* (2017), pp. 439–453.

[32] Lau, J. Merkelized abstract syntax tree. Bitcoin Improvement Proposal, Accessed May 2018. `https://github.com/bitcoin/bips/blob/master/bip-0114.mediawiki`.

[33] Lind, J., Eyal, I., Pietzuch, P. R., and Sirer, E. G. Teechan: Payment channels using trusted execution environments. `http://arxiv.org/abs/1612.07766`.

[34] Lindell, Y. Fast Secure Two-Party ECDSA Signing. In *CRYPTO* (2017), pp. 613–644.

[35] Luu, L., Narayanan, V., Zheng, C., Baweja, K., Gilbert, S., and Saxena, P. A secure sharding protocol for open blockchains. In *Computer and Communications Security* (2016), pp. 17–30.

[36] Malavolta, G., Moreno-Sanchez, P., Kate, A., Maffei, M., and Ravi, S. Concurrency and privacy with payment-channel networks. In *Computer and Communications Security* (2017).

[37] McCorry, P., Möser, M., Shahandashti, S. F., and Hao, F. Towards bitcoin payment networks. In *Australasian Conference Information Security and Privacy* (2016).

[38] Micali, S., Ohta, K., and Reyzin, L. Accountable-subgroup multisignatures. In *Computer and Communications Security* (2001), pp. 245–254.

[39] Miller, A., Bentov, I., Kumaresan, R., and McCorry, P. Sprites: Payment channels that go faster than lightning. *CoRR abs/1702.05812* (2017).

[40] Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system. `https://bitcoin.org/bitcoin.pdf`, 2008.

[41] Paillier, P. Public-key cryptosystems based on composite degree residuosity classes. In *International Conference on the Theory and Applications of Cryptographic Techniques* (1999), pp. 223–238.

[42] Poelstra, A. Lightning in scriptless scripts. Mailing list post, Accessed May 2018. https://lists.launchpad.net/mimblewimble/msg00086.html.

[43] Poelstra, A. Scriptless scripts. Presentation slides, Accessed May 2018. https://download.wpsoftware.net/bitcoin/wizardry/mw-slides/2017-05-milan-meetup/slides.pdf.

[44] Poon, J., and Dryja, T. The bitcoin lightning network: Scalable off-chain instant payments. Technical Report, Accessed May 2018. https://lightning.network/lightning-network-paper.pdf.

[45] Regev, O. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM) 56*, 6 (2009), 34.

[46] Roos, S., Moreno-Sanchez, P., Kate, A., and Goldberg, I. Settling payments fast and private: Efficient decentralized routing for path-based transactions. In *Network and Distributed System Security Symposium* (2018).

[47] Schnorr, C.-P. Efficient signature generation by smart cards. *Journal of cryptology 4*, 3 (1991), 161–174.

[48] Thomas, S., and Schwartz, E. A Protocol for Interledger Payments. Whitepaper, Accessed May 2018. https://interledger.org/interledger.pdf.

# A Multi-hop Locks Correctness

In this section, we define the notion of correctness for multi-hop locks.

**Definition 3** (Correctness of multi-hop locks)**.** *Let $\mathbb{L}$ be a multi-hop lock, $\lambda \in \mathbb{N}^+$ and $n \in \mathsf{poly}(\lambda)$. Let $(U_0, \ldots, U_n) \in \mathbb{U}^n$ be a vector of users, $(\mathsf{sk}_0, \ldots, \mathsf{sk}_{n-1})$ and $(\mathsf{sk}_1^*, \ldots, \mathsf{sk}_n^*)$ two vectors of private keys and $(\mathsf{pk}_0, \ldots, \mathsf{pk}_{n-1})$ a vector of shared public keys such that for all $0 \leq i < n$, it holds that*

$$\{(\mathsf{sk}_i, \mathsf{pk}_i), (\mathsf{sk}_{i+1}^*, \mathsf{pk}_i)\} \leftarrow \langle \mathsf{KGen}_{U_i}(1^\lambda), \mathsf{KGen}_{U_{i+1}}(1^\lambda) \rangle.$$

*Let $(s_0^I, \ldots, s_n^I)$ be vector of initial states and $k_n$ be a key such that for all $0 \leq i < n$*

$$\{s_0^I, \ldots, (s_n^I, k_n)\} \leftarrow \left\langle \begin{array}{c} \mathsf{Setup}_{U_0}(1^\lambda, U_1, \ldots, U_n) \\ \ldots \\ \mathsf{Setup}_{U_n}(1^\lambda) \end{array} \right\rangle$$

*Furthermore, let $(\ell_0, \ldots, \ell_{n-1})$ be a vector of locks, $(s_1^L, \ldots, s_n^L)$ and $(s_0^R, \ldots, s_{n-1}^R)$ vectors of states, and $(k_0, \ldots, k_{n-1})$ a vector of keys such that for all $0 \leq i < n$, it holds that*

$$\{(\ell_i, s_i^R), (\ell_i, s_{i+1}^L)\} \leftarrow \left\langle \begin{array}{c} \mathsf{Lock}_{U_i}(s_i^I, \mathsf{sk}_i, \mathsf{pk}_i) \\ \mathsf{Lock}_{U_{i+1}}(s_{i+1}^I, \mathsf{sk}_{i+1}^*, \mathsf{pk}_i) \end{array} \right\rangle$$

*and*

$$k_i \leftarrow \mathsf{Rel}(k_{i+1}, (s_{i+1}^I, s_{i+1}^L, s_{i+1}^R))$$

*where $s_n^R$ is $\perp$. We say that $\mathbb{L}$ is correct if there exists a negligible function $\mathsf{negl}$ such that for all $0 \leq i < n$ it holds that*

$$\Pr\left[\mathsf{Vf}(\ell_i, k_i) = 1\right] \geq 1 - \mathsf{negl}(\lambda).$$

# B Security Analysis

Throughout the analysis we denote by $\mathsf{poly}(\lambda)$ any function that is bounded by a polynomial in $\lambda$. We denote any function that is negligible in the security parameter by $\mathsf{negl}(\lambda)$. We say that an algorithm is PPT if it is modelled as a probabilistic Turing machine whose running time is bounded by some function $\mathsf{poly}(\lambda)$. In the following we elaborate on the security analysis of our constructions.

We shall point out that in this analysis we model a very strong variant of anonymous communication, which might not always be reasonable to assume. More realistic privacy guarantees are captured by onion routing [19] functionalities or mix networks [30]. For ease of exposition we stick to our simplistic model, noting that our proof is completely parametric and one can switch to a less idealized functionality in a modular manner.

## B.1 Generic Construction

Here we elaborate the proof of Theorem 1.

*Proof.* We define the following sequence of hybrids, where we gradually modify the initial experiment.

$\mathcal{H}_0$ : Is identical to the protocol as described in Section 4.2.

$\mathcal{H}_1$ : Instead of sending messages through the $\Pi_{\mathsf{anon}}$ channel, the parties communicate in interaction with the ideal functionality $\mathcal{F}_{\mathsf{anon}}$.

**Anon**($m$, $U_i$)

Upon invocation $U_j$ on input $(m, U_i)$:

send $(m, U_i)$ to $U_i$

$\mathcal{H}_2$ : Consider the following ensemble of variables in the interaction with $\mathcal{A}$: A honest user $U_i$, a key pair $(\mathsf{sk}_i, \mathsf{pk})$, a state $s^I$, a tuple $(\ell_i, \ell_{i+1}, s^L, s^R)$ such that

$$\{\cdot, (\ell_i, s^L)\} \leftarrow \langle \cdot, \mathsf{Lock}_{U_i}(s^I, \mathsf{sk}_i, \mathsf{pk})\rangle$$

and

$$\{(\ell_{i+1}, s^R), \cdot\} \leftarrow \langle \mathsf{Lock}_{U_i}(s^I, \mathsf{sk}_i, \mathsf{pk}), \cdot\rangle.$$

If, for any set of these variables, the adversary returns some $k$ such that $\mathsf{Vf}(\ell_{i+1}, k) = 1$ and $\mathsf{Vf}(\ell_i, \mathsf{Rel}(k, (s^I, s^L, s^R))) \neq 1$, then the experiment aborts.

$\mathcal{H}_3$ : Consider the following ensemble of variables in the interaction with $\mathcal{A}$: A pair of honest users $(U_0, U_i)$ a set of (possibly corrupted) users $(U_1, \ldots, U_n)$, a key pair $(\mathsf{sk}_i, \mathsf{pk})$, a set of initial states

$$(s_0^I \ldots, s_n^I) \leftarrow \langle \mathsf{Setup}_{U_0}(1^\lambda, U_1, \ldots, U_n), \ldots, \mathsf{Setup}_{U_n}(1^\lambda)\rangle,$$

and a pair of locks $(\ell_{i-1}, \ell_i)$ such that

$$\{\cdot, (\ell_{i-1}, \cdot)\} \leftarrow \langle \cdot, \mathsf{Lock}_{U_i}(s_i^I, \mathsf{sk}_i, \mathsf{pk})\rangle$$

and

$$\{(\ell_i, \cdot), \cdot\} \leftarrow \langle \mathsf{Lock}_{U_i}(s_i^I, \mathsf{sk}_i, \mathsf{pk}), \cdot\rangle.$$

If, for any set of these variables, the adversary returns some $k_{i-1}$ such that $\mathsf{Vf}(\ell_{i-1}, k_{i-1}) = 1$ before the user $U_i$ outputs a key $k_i$ such that $\mathsf{Vf}(\ell_i, k_i) = 1$, then the experiment aborts.

$\mathcal{H}_4$ : Let $S = (U_0, \ldots, U_m)$ be an ordered set of (possibly corrupted) users. We say that that an ordered subset $A = (U_1, \ldots, U_j)$ is *adversarial* if $U_i$ is honest and $(U_{i+1}, \ldots, U_j)$ are corrupted. Note that every set of users can be expressed as a concatenation of adversarial subsets, that is $S = (A_1 || \ldots || A_{m'})$, for some $m' \leq m$. Whenever a honest user is requested to setup a lock for a certain set $S = (A_1 || \ldots || A_{m'})$, it initializes an independent lock for each subset $(A_i, A_{i+1}^0)$, where $A_{i+1}^0$ is the first element of the $(i+1)$-th set, if present. Whenever some $A_{i+1}^0$ is requested to release the key for the corresponding lock (recall that all $A_{i+1}^0$ are honest nodes) it releases the key for the fresh lock $(A_i, A_{i+1}^0)$ instead.

$\mathcal{S}$ : The interaction of the simulator is identical to $\mathcal{H}_4$ except that the actions of $\mathcal{S}$ are dictated by the interaction with $\mathcal{F}$. The simulator reads the communication of $\mathcal{A}$ with the honest users via $\mathcal{F}_{\mathsf{anon}}$ and is queried by $\mathcal{F}$ on the following set of inputs.

1. $(\cdot, \cdot, \cdot, \cdot, \mathsf{Init})$: The simulator reconstruct the adversarial set (defined above) from the ids and sets up a fresh lock chain.

2. $(\cdot, \mathsf{Lock})$: The simulator initiates the locking procedure with the adversary and replies with $\perp$ if the execution is not successful.

3. $(\cdot, \mathsf{Rel})$ The simulator releases the key of the corresponding lock and publishes it.

If $\mathcal{A}$ interacts with a honest user (e.g., by releasing a lock) the simulator queries the corresponding interface of $\mathcal{F}$.

Note that the simulator is efficient and interacts in the adversary with the ideal world. Furthermore, the simulation is always consistent with the ideal world, i.e., if the adversary's action is not supported by the interfaces of $\mathcal{F}$ the simulation aborts. What is left to be shown is that the neighbouring hybrids are indistinguishable to the eyes of the environment $\mathcal{E}$.

**Lemma 1.** *For all PPT distinguishers $\mathcal{E}$ it holds that*

$$\mathsf{EXEC}_{\mathcal{H}_0,\mathcal{A},\mathcal{E}} \approx \mathsf{EXEC}_{\mathcal{H}_1,\mathcal{A},\mathcal{E}}.$$

*Proof.* Follows directly from the security of $\Pi_{\mathsf{anon}}$. □

**Lemma 2.** *For all PPT distinguishers $\mathcal{E}$ it holds that*

$$\mathsf{EXEC}_{\mathcal{H}_1,\mathcal{A},\mathcal{E}} \equiv \mathsf{EXEC}_{\mathcal{H}_2,\mathcal{A},\mathcal{E}}.$$

*Proof.* Follows from the homomorphic property of the function $g$: Recall that a key-lock pair $(k, \ell)$ is valid if and only if $g(k) = \ell$. Let $(k_i, \ell_i)$ be the output of $\mathcal{A}$, by construction we have that $\ell_i = \ell_{i-1} + g(y_i)$, for some $(\ell_{i-1}, y_i)$, which is part of the state of the honest node. Since the release algorithm computes $k_i - y_i$ we have that

$$\begin{aligned} g(k_i - y_i) &= g(k_i) - g(y_i) \\ &= \ell_i - g(y_i) \\ &= \ell_{i-1} + g(y_i) - g(y_i) \\ &= \ell_{i-1} \end{aligned}$$

with probability 1, by the homomorphic property of $g$. □

**Lemma 3.** *For all PPT distinguishers $\mathcal{E}$ it holds that*

$$\mathsf{EXEC}_{\mathcal{H}_2,\mathcal{A},\mathcal{E}} \approx \mathsf{EXEC}_{\mathcal{H}_3,\mathcal{A},\mathcal{E}}.$$

*Proof.* Let $q \in \mathsf{poly}(\lambda)$ be a bound on the number of interactions. Recall that $\mathcal{H}_2$ and $\mathcal{H}_3$ differ only for the case where the adversary outputs a key for a honestly generated lock before the trapdoor is released. Assuming towards contradiction that the probability that this event happens is non-negligible, we can construct the following reduction against the one-wayness of $g$: On input some $Y^* \in \mathcal{R}$, the reduction guesses a session $j \in [1, q]$ and some index $i \in [1, n]$. The setup algorithm of the $j$-th session is modified as follows: $Y_i$ is set to be $Y^*$. Then, for all $\iota \in [i-1, 0]$, the setup samples some $y_\iota \in \mathcal{D}$ and returns $(Y_\iota = Y_{\iota+1} - g(y_\iota), Y_{\iota+1}, y_\iota)$. The setup samples a random $y_i \in \mathcal{D}$ and sets $Y_{i+1} = g(y_i)$. Then, for $\iota \in [i+1, n-1]$, the setup samples $y_\iota \in \mathcal{D}$ returns $(Y_\iota, Y_\iota + g(y_\iota), y_\iota)$. The nodes $(U_1, \ldots, U_{n-1})$ are given the corresponding output (except for $U_i$) and $U_n$ is given $(Y_{n-1}, \sum_{j=i}^{n-1} y_j)$. If the node $U_i$ is requested to release the lock, the reduction aborts. At some point of the execution the adversary $\mathcal{A}$ outputs some $y*$, and the reduction returns $y^* + y_{i-1}$.

The reduction is clearly efficient and, whenever $j$ and $i$ are guessed correctly, the reduction does not abort. Since the group defined by $g$ is abelian, the distribution induced by the modified setup algorithm is identical to the original (except for the initial state of $U_1$). Also note that, whenever $j$ and $i$ are guessed correctly, the user $U_i$ is honest and therefore the adversary does not not see the corresponding internal state. It follows that the reduction is identical to $\mathcal{H}_2$, to the eyes of the adversary. Finally, whenever the adversary

outputs some valid $k_{i-1}$ for $\ell_{i-1}$, then it holds that $g(k_{i-1}) = \ell_{i-1}$. Substituting we have that

$$\begin{aligned} g(k_{i-1}) &= \ell_{i-1} \\ g(y^*) &= Y_{i-1} \\ g(y^*) &= Y^* - g(y_{i-1}) \\ g(y^*) + g(y_{i-1}) &= Y^* \\ g(y^* + y_{i-1}) &= Y^*. \end{aligned}$$

It follows that the reduction is successful with probability at least $\frac{1}{q \cdot n \cdot \mathsf{poly}(\lambda)}$. This proves our statement. □

**Lemma 4.** *For all PPT distinguishers $\mathcal{E}$ it holds that*

$$\mathsf{EXEC}_{\mathcal{H}_3,\mathcal{A},\mathcal{E}} \equiv \mathsf{EXEC}_{\mathcal{H}_4,\mathcal{A},\mathcal{E}}.$$

*Proof.* Recall that adversarial sets are always interleaved by a honest node. Therefore in $\mathcal{H}_3$ for each adversarial set starting at index $i$ there exists a $y$ such that $Y_i = Y_{i-1} + g(y)$ and $\mathcal{A}$ is not given $y$. Since $y$ is randomly sampled from $\mathcal{D}$ we have that $Y_{i-1} + g(y) \equiv Y'$, for some $Y'$ sampled uniformly from $\mathcal{R}$, which corresponds to the view of $\mathcal{A}$ in $\mathcal{H}_4$. □

**Lemma 5.** *For all PPT distinguishers $\mathcal{E}$ it holds that*

$$\mathsf{EXEC}_{\mathcal{H}_4,\mathcal{A},\mathcal{E}} \equiv \mathsf{EXEC}_{\mathcal{F},\mathcal{S},\mathcal{E}}.$$

*Proof.* The changes between the two experiments are only conceptual and the equivalence of the views follows. □

This concludes our analysis. □

## B.2 Schnorr-based Construction

Here we prove Theorem 2.

*Proof.* We define the following sequence of hybrids, where we gradually modify the initial experiment.

$\mathcal{H}_0$ : Is identical to the protocol as described in Section 4.3.

$\mathcal{H}_1$ : Instead of sending messages through the $\Pi_{\mathsf{anon}}$ channel, the parties communicate in interaction with the ideal functionality $\mathcal{F}_{\mathsf{anon}}$.

**Anon($m$, $U_i$)**

Upon invocation $U_j$ on input $(m, U_i)$:

send $(m, U_i)$ to $U_i$

$\mathcal{H}_2$ : All the calls to the commitment scheme are replaced with interactions with the ideal functionality $\mathcal{F}_{\mathsf{com}}$, defined in the following.

**Commit($\boldsymbol{sid}, m$)**

Upon invocation by $U_i$ (for $i \in \{0, 1\}$):

record $(sid, i, m)$ and send $(\mathsf{com}, sid)$ to $U_{1-i}$

if some $(sid, \cdot, \cdot)$ is already stored ignore the message

**Decommit($\boldsymbol{sid}$)**

Upon invocation by $U_i$ (for $i \in \{0, 1\}$):

if $(sid, i, m)$ is recorded then send $(\mathsf{decom}, sid, m)$ to $U_{1-i}$

Instead of calling the $\mathsf{Commit}$ algorithm on some message $m$, the parties sent a message of the form **Commit(sid, m)** to the ideal functionality, and the decommitment algorithm is replaced with a call to **Decommit(sid)**. The verifying party simply records messages from $\mathcal{F}_{\mathsf{com}}$.

$\mathcal{H}_3$ : All the calls to the $\mathsf{NIZK}$ scheme are replaced with interactions with the ideal functionality $\mathcal{F}_{\mathsf{NIZK}}$:

**Prove(*sid*, x, w)**

Upon invocation by $U_i$ (for $i \in \{0,1\}$):
if $R(x,w) = 1$ then send $(\mathsf{proof}, sid, x)$ to $U_{1-i}$

Instead of running the proving algorithm in input $(x, w)$, the proving party queries the functionality on **Prove(sid, x, w)**. The verifier records the messages from $\mathcal{F}_{\mathsf{NIZK}}$.

$\mathcal{H}_4, \mathcal{H}_5, \mathcal{H}_6, \mathcal{S}$ : The subsequent hybrids are defined as $\mathcal{H}_2, \mathcal{H}_3, \mathcal{H}_4, \mathcal{S}$, respectively, in Theorem 1.

As argued before, the simulator is efficient and the interaction is consistent with the inputs of the ideal functionality. In the following we prove the indistinguishability of the neighbouring experiments.

**Lemma 6.** *For all PPT distinguishers $\mathcal{E}$ it holds that*

$$\mathsf{EXEC}_{\mathcal{H}_0,\mathcal{A},\mathcal{E}} \approx \mathsf{EXEC}_{\mathcal{H}_1,\mathcal{A},\mathcal{E}}.$$

*Proof.* Follows directly from the security of $\Pi_{\mathsf{anon}}$. □

**Lemma 7.** *For all PPT distinguishers $\mathcal{E}$ it holds that*

$$\mathsf{EXEC}_{\mathcal{H}_1,\mathcal{A},\mathcal{E}} \approx \mathsf{EXEC}_{\mathcal{H}_2,\mathcal{A},\mathcal{E}}.$$

*Proof.* Follows directly from the security of the commitments scheme $\mathsf{COM}$. □

**Lemma 8.** *For all PPT distinguishers $\mathcal{E}$ it holds that*

$$\mathsf{EXEC}_{\mathcal{H}_2,\mathcal{A},\mathcal{E}} \approx \mathsf{EXEC}_{\mathcal{H}_3,\mathcal{A},\mathcal{E}}.$$

*Proof.* Follows directly from the security of the non-interactive zero-knowledge scheme $\mathsf{NIZK}$. □

**Lemma 9.** *For all PPT distinguishers $\mathcal{E}$ it holds that*

$$\mathsf{EXEC}_{\mathcal{H}_3,\mathcal{A},\mathcal{E}} \approx \mathsf{EXEC}_{\mathcal{H}_4,\mathcal{A},\mathcal{E}}.$$

*Proof.* In order to show this claim, we introduce an intermediate experiment.

$\mathcal{H}_3^*$ : The key generation and locking algorithms are substituted with the interaction with the functionality $\mathcal{F}_{\mathsf{schnorr}}$, which provides any two users with the interfaces specified below. Note that the signing interface is called by both parties on input $m$ and $y = \sum_{j=0}^{i} y_j$, where $i$ is the position of the lock in the chains and the $y_j$ are defined as in the original protocol.

**KeyGen($\mathbb{G}$, $G$, $q$)**

Upon invocation by both $U_0$ and $U_1$ on input ($\mathbb{G}$, $G$, $q$):
sample $x \leftarrow\!\!\$\, \mathbb{Z}_q$ and compute $Q = x \cdot G$
set $\mathsf{sk}_{U_0,U_1} = x$
sample $x_0$ and $x_1$ randomly
sample a hash function $H : \{0,1\}^* \to \{0,1\}^{|q|}$
send $(x_0, Q)$ to $U_0$ and $(x_1, Q)$ to $U_1$
ignore future calls by $(U_0, U_1)$

**Sign($m, y$)**

Upon invocation by both $U_0$ and $U_1$ on input $(m, y)$:
compute $(R, s) = \mathsf{Sig}_{\mathsf{schnorr}}(\mathsf{sk}_{U_0,U_1}, m)$
return $(R, s - y)$

We defer the indistinguishability proof to lemma 10. Let $\mathsf{cheat}$ by the event that triggers an abort of the experiment in $\mathcal{H}_4$, that is, the adversary returns some $k$ such that $\mathsf{Vf}(\ell_{i+1}, k) = 1$ and that $\mathsf{Vf}(\ell_i, \mathsf{Rel}(k, (s^I, s^L, s^R))) \neq 1$. Assume towards contradiction that $\mathsf{Pr}[\mathsf{cheat} \mid \mathcal{H}_3^*] \geq \frac{1}{\mathsf{poly}(\lambda)}$, then we can construct the following reduction against the strong-existential unforgeability of Schnorr signatures: The reduction receives as input a public

key $\mathsf{pk}$ and samples an index $j \in [1, q]$, where $q \in \mathsf{poly}(\lambda)$ is a bound on the total amount of interactions. Let $Q$ be the key generated in the $j$-th interaction, the reduction sets $Q = \mathsf{pk}$. All the calls to the signing algorithm are redirected to the signing oracle. If the event $\mathsf{cheat}$ happens, the reduction returns corresponding $(k^*, \ell^*) = (\sigma^*, (m^*, \mathsf{pk}^*))$, otherwise it aborts.

The reduction is clearly efficient. Assume for the moment that $j$ is the index of the interaction where $\mathsf{cheat}$ happens, and let $i+1$ be the index that identifies the lock $\ell^*$ in the corresponding chain. Note that in case the guess of the reduction is correct we have that $\mathsf{pk}^* = \mathsf{pk}$. Since $\mathsf{cheat}$ happens we have that $\mathsf{Vf}_{\mathsf{schnorr}}(\mathsf{pk}^*, m^*, \sigma^*) = 1$ and the release fails, i.e., $\mathsf{Vf}(\ell_i, \mathsf{Rel}(k, (s_i^I, s_i^L, s_i^R)) \neq 1$ (where $\ell_i$ is the lock in the previous position as $\ell^*$ in the same chain). Recall that the release algorithm parses $s_i^L$ as $(W_{i,0}, w_{i,1})$ and $\sigma^*$ as $(R^*, s^*)$ and returns $(W_{i,0}, w_{i,1} + s^* - (s_i^R + y_i))$. Substituting with the corresponding values

$$\begin{aligned}
&\left(W_{i,0}, w_{i,1} + s^* - (s_i^R + y_i)\right) \\
&= \left(R_i, \left(s_i - \sum_{j=0}^{i-1} y_j\right) + s^* - \left(\left(s_j - \sum_{j=0}^{i} y_j\right) + y_i\right)\right) \\
&= \left(R_i, s_i + s^* - s_j\right),
\end{aligned}$$

where $s_j$ is the answer of the oracle on the $j$-th session on input $m_j$. This implies that $s^* \neq s_j$, otherwise $(R_i, s_i)$ would be a valid signature since it is an output of the signing oracle. Since each message uniquely identifies a session (the same message is never queried twice to the interface **Sign($m$,$y$)**) this implies that $(\sigma^*, (m^*, \mathsf{pk}^*))$ is a valid forgery. By assumption this happens with probability at leasts $\frac{1}{q \cdot \mathsf{poly}(\lambda)}$, which is a contradiction and proves that $\mathsf{Pr}\left[\mathsf{cheat} \mid \mathcal{H}_3^*\right] \leq \mathsf{negl}(\lambda)$. Since the experiments $\mathcal{H}_3$ and $\mathcal{H}_4$ differ only when $\mathsf{cheat}$ happens (and $\mathcal{H}_4$ aborts), we are only left with showing the indistinguishability of $\mathcal{H}_3$ and $\mathcal{H}_3^*$.

**Lemma 10.** *For all PPT distinguishers $\mathcal{E}$ it holds that*

$$\mathsf{EXEC}_{\mathcal{H}_3, \mathcal{A}, \mathcal{E}} \approx \mathsf{EXEC}_{\mathcal{H}_3^*, \mathcal{A}, \mathcal{E}}.$$

*Proof.* The proof consists of the description of the simulator for the interactive lock algorithm. The simulator for the key generation phase is trivial and therefore it is omitted. We describe two simulators depending on whether the honest adversary is playing the role of the "left" or "right" party. For each proof, both the simulators implicitly check that the given witness is valid and abort if this is not the case.

1. Left corrupted: Prior to the interaction the simulator is sent $(Y, y, (\mathsf{prove}, \{\exists y^* \text{ s.t } y^* \cdot G = Y\}, y^*))$, which is the state corresponding to the execution of the lock. After agreeing on a message $m$, the simulator sends $(\mathsf{com}, sid)$ to $\mathcal{A}$, for a random $sid$. The simulator also queries the interface **Sign** on input $m, y^*$ and receives a signature $\sigma = (R, s)$. At some point of the execution $\mathcal{A}$ sends $(R_0, (\mathsf{prove}, \{\exists r_0 \text{ s.t } r_0 \cdot G = R_0\}, r_0))$. The simulator replies with

$$\left(\begin{array}{l} \left(\mathsf{decom}, sid, \left(\begin{array}{l} R^* = R - (R_0 + Y), \\ \left(\begin{array}{l} \mathsf{proof}, sid, \\ \{\exists r^* \text{ s.t } r^* \cdot G = R^*\} \end{array}\right) \end{array}\right)\right), \\ R^*, (s - r_0 - e \cdot x_0) \end{array}\right),$$

   where $e = H(\mathsf{pk} \| R^* \| m)$ and $x_0$ is the value returned by the key generation to $\mathcal{A}$. The rest of the execution is unchanged.

2. Right corrupted: Prior to the interaction the simulator is sent $(Y, y, (\mathsf{prove}, \{\exists y^* \text{ s.t } y^* \cdot G = Y\}, y^*))$, which is the state corresponding to the execution of the lock. After agreeing on a message $m$, the simulator is given

$$\left(\mathsf{com}, sid, \left(R_1, \left(\begin{array}{l} \mathsf{prove}, sid, \\ \{\exists r_1 \text{ s.t } r_1 \cdot G = R_1\}, r_1 \end{array}\right)\right)\right)$$

   by $\mathcal{A}$. The simulator then queries the interface **Sign** on input $m, y^*$ and receives a signature $\sigma = (R, s)$. The simulator sends $(R^* = R - (R_1 + Y), (\mathsf{proof}, sid, \{\exists r^* \text{ s.t } r^* \cdot G = R^*\}))$ to $\mathcal{A}$ and receives $((\mathsf{decom}, sid), s^*)$ in response. The simulator checks whether $s^* = r_1 + e \cdot x_1$, where $e = H(\mathsf{pk} \| R^* \| m)$, and returns $s$ if this is the case.

Both simulators are obviously efficient and the distributions induced by the simulated views are identical to the ones of the original protocol. □

This concludes the proof of lemma 9. □

**Lemma 11.** *For all PPT distinguishers $\mathcal{E}$ it holds that*

$$\mathsf{EXEC}_{\mathcal{H}_4,\mathcal{A},\mathcal{E}} \approx \mathsf{EXEC}_{\mathcal{H}_5,\mathcal{A},\mathcal{E}}.$$

*Proof.* Let $q \in \mathsf{poly}(\lambda)$ be a bound on the number of interactions. Let $\mathsf{cheat}$ denote the events that triggers an abort in $\mathcal{H}_5$ but not in $\mathcal{H}_4$. In the following we are going to show that $\mathsf{Pr}[\mathsf{cheat} \mid \mathcal{H}_4] \leq \mathsf{negl}(\lambda)$, thus proving the indistinguishability of $\mathcal{H}_4$ and $\mathcal{H}_5$. Assume that the converse is true, then we can construct the following reduction against the discrete logarithm problem (which is implied by the sEUF of Schnorr): On input some $Y^* \in \mathbb{G}$, the reduction guesses a session $j \in [1, q]$ and some index $i \in [1, n]$. The setup algorithm of the $j$-th session is modified as follows: $Y_i$ is set to be $Y^*$. Then, for all $\iota \in [i-1, 0]$, the setup samples some $y_\iota \in \mathbb{Z}_q$ and returns $(Y_\iota = Y_{\iota+1} - y_\iota \cdot G, Y_{\iota+1}, y_\iota)$. The setup samples a random $y_i \in \mathbb{Z}_q$ and sets $Y_{i+1} = y_i \cdot G$. Then, for $\iota \in [i+1, n-1]$, the setup samples $y_\iota \in \mathbb{Z}_q$ returns $(Y_\iota, Y_\iota + y_\iota \cdot G, y_\iota)$. The nodes $(U_1, \ldots, U_{n-1})$ are given the corresponding output (except for $U_i$) and $U_n$ is given $(Y_{n-1}, \sum_{j=i}^{n-1} y_j)$. If the node $U_i$ is requested to release the lock, the reduction aborts. At some point of the execution the adversary $\mathcal{A}$ outputs some $k^* = (R^*, s^*)$. The reduction parses $s^R$ as the updated state of $U_i$ and returns $s^* + y_{i-1} - s^R$.

The reduction is clearly efficient and, whenever $j$ and $i$ are guessed correctly, the reduction does not abort. Since the group $\mathbb{G}$ is abelian and the $U_i$ is honest, the distribution induced by the modified setup algorithm is identical to the original to the eyes of the adversary. Recall that $\mathsf{cheat}$ happens only in the case where $k^*$ is a valid opening for $\ell_i$ and the release algorithm is successful on input $k^*$ (if the last condition is not satisfied both $\mathcal{H}_4$ and $\mathcal{H}_5$ abort). Substituting, we have that $s^R$ is of the form $r_0 + r_1 + e \cdot (x0 + x_1) - y = s' - y$, for some $y \in \mathbb{Z}_q$. Since the release is successful, then it must be the case that $(R' = (r_0 + r_1) \cdot G + Y_{i-1}, s')$ is a valid Schnorr signature on the message $m_{i-1}$ (agreed by the two parties in the locking algorithm for $\ell_{i-1}$), which implies that $y \cdot G = Y_{i-1}$. As argued in the proof of lemma 9, if $s^* \neq s'$, then we have an attacker against the strong unforgeability of the signature scheme. It follows that $s^* = s'$ with all but negligible probability. Substituting we have

$$\begin{aligned}
(s^* + y_{i-1} - s^R) \cdot G &= (s^* + y_{i-1} - s' + y) \cdot G \\
&= (y_{i-1} + y) \cdot G \\
&= y_{i-1} \cdot G + y \cdot G \\
&= y_{i-1} \cdot G + Y_{i-1} \\
&= y_{i-1} \cdot G + (Y^* - y_{i-1} \cdot G) \\
&= Y^*
\end{aligned}$$

as expected. Since, by assumption, this happens with probability at least $\frac{1}{q \cdot n \cdot \mathsf{poly}(\lambda)}$ we have a successful attacker against the discrete logarithm problem. This proves our statement.

□

**Lemma 12.** *For all PPT distinguishers $\mathcal{E}$ it holds that*

$$\mathsf{EXEC}_{\mathcal{H}_5,\mathcal{A},\mathcal{E}} \equiv \mathsf{EXEC}_{\mathcal{H}_6,\mathcal{A},\mathcal{E}}.$$

*Proof.* Recall that adversarial sets are always interleaved by a honest node. Therefore in $\mathcal{H}_5$ for each adversarial set starting at index $i$ there exists a $y$ such that $Y_i = Y_{i-1} + y \cdot G$ and $\mathcal{A}$ is not given $y$. Since $y$ is randomly sampled from $\mathbb{Z}_q$ we have that $Y + i - 1 + y \cdot G \equiv Y'$, for some $Y'$ sampled uniformly from $\mathbb{G}$, which corresponds to the view of $\mathcal{A}$ in $\mathcal{H}_6$. □

**Lemma 13.** *For all PPT distinguishers $\mathcal{E}$ it holds that*

$$\mathsf{EXEC}_{\mathcal{H}_6,\mathcal{A},\mathcal{E}} \equiv \mathsf{EXEC}_{\mathcal{F},\mathcal{S},\mathcal{E}}.$$

*Proof.* The change is only syntactical and the indistinguishability follows. □

This concludes our analysis. □

## B.3 ECDSA-based Construction

In the following we prove Theorem 3.

*Proof.* The sequence of hybrids that we define is identical to the one described in the proof of Theorem 2. In the following we prove the indistinguishability of neighbouring experiments only for the cases where the argument needs to be modified. If the argument is identical, the proof is omitted.

**Lemma 14.** *For all PPT distinguishers $\mathcal{E}$ it holds that*

$$\mathsf{EXEC}_{\mathcal{H}_3,\mathcal{A},\mathcal{E}} \approx \mathsf{EXEC}_{\mathcal{H}_4,\mathcal{A},\mathcal{E}}.$$

*Proof.* In order to show this claim, we introduce an intermediate experiment.

$\mathcal{H}_3^*$ : The key generation and locking algorithms are substituted with the interaction with the functionality $\mathcal{F}_{\mathsf{ECDSA}}$, which provides any pair of users with the interfaces specified below. Note that the locking algorithm is called by both parties on input $m$ and $y = \sum_{j=0}^{i} y_j$, where $i$ is the position of the lock in the chains and the $y_j$ are defined as in the original protocol.

**KeyGen($\mathbb{G}$, $G$, $q$)**

Upon invocation by both $U_0$ and $U_1$ on input ($\mathbb{G}$, $G$, $q$):
sample $x \leftarrow\!\!\$\, \mathbb{Z}_q$ and compute $Q = x \cdot G$
sample $x_0$ and $x_1$ randomly
sample a hash function $H : \{0,1\}^* \to \{0,1\}^{|q|}$
sample a key pair $(\mathsf{sk}_{U_0,U_1}, \mathsf{pk}_{U_0,U_1}) \leftarrow \mathsf{KGen}_{\mathsf{HE}}(1^\lambda)$
compute $c \leftarrow \mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}, \tilde{r})$ for a random $\tilde{r}$
send $(x_0, Q, H, \mathsf{sk})$ to $U_0$ and $(x_1, Q, H, c)$ to $U_1$
ignore future calls by $(U_0, U_1)$

**Sign($m, y$)**

Upon invocation by both $U_0$ and $U_1$ on input $(m, y)$:
compute $(r, s) = \mathsf{Sig}_{\mathsf{ECDSA}}(\mathsf{sk}_{U_0,U_1}, m)$
return $(r, \mathsf{min}(s \cdot y, -s \cdot y))$

The indistinguishability proof of $\mathcal{H}_3$ and $\mathcal{H}_3^*$ is formally shown in lemma 15. Let $\mathsf{cheat}$ by the event that triggers an abort of the experiment in $\mathcal{H}_4$, that is, the adversary returns some $k$ such that $\mathsf{Vf}(\ell_{i+1}, k) = 1$ and $\mathsf{Vf}(\ell_i, \mathsf{Rel}(k, (s^I, s^L, s^R))) \neq 1$. Assume towards contradiction that $\mathsf{Pr}[\mathsf{cheat} \mid \mathcal{H}_3^*] \geq \frac{1}{\mathsf{poly}(\lambda)}$, then we can construct the following reduction against the strong-existential unforgeability of ECDSA signatures: The reduction receives as input a public key $\mathsf{pk}$ and samples an index $j \in [1, q]$, where $q \in \mathsf{poly}(\lambda)$ is a bound on the total amount of interactions. Let $Q$ be the key generated in the $j$-th interaction, the reduction sets $Q = \mathsf{pk}$. All the calls to the signing algorithm are redirected to the signing oracle. If the event $\mathsf{cheat}$ happens, the reduction returns corresponding $(k^*, \ell^*) = (\sigma^*, (m^*, \mathsf{pk}^*))$, otherwise it aborts.

The reduction runs in polynomial time. Assume for the moment that $j$ is the index of the interaction where $\mathsf{cheat}$ happens, and let $i+1$ be the index that identifies the lock $\ell^*$ in the corresponding chain. Note that in case the guess of the reduction is correct we have that $\mathsf{pk}^* = \mathsf{pk}$. Since $\mathsf{cheat}$ happens we have that $\mathsf{Vf}_{\mathsf{ECDSA}}(\mathsf{pk}^*, m^*, \sigma^*) = 1$ and the release fails, i.e., $\mathsf{Vf}(\ell_i, \mathsf{Rel}(k^*, k^*, (s_i^I, s_i^L, s_i^R))) \neq 1$ (where $\ell_i$ is the lock in the previous position as $\ell^*$ in the same chain). Recall that the release algorithm parses $s_i^L$ as $(w_{i,0}, w_{i,1})$, $\sigma^*$ as $(r^*, s^*)$, and $s_i^R$ as $(s', m, \mathsf{pk})$ and computes $t = w_1 \cdot (\frac{s'}{s^*} - y)^{-1}$ and $t' = w_1 \cdot (-\frac{s'}{s^*} - y)^{-1}$. Then it returns either $(w_{i,0}, \mathsf{min}(t, -t))$ or $(w_{i,0}, \mathsf{min}(t', -t'))$ depending on which verifies as a valid signature on $m$ under $\mathsf{pk}$. Substituting with the corresponding values (for the case $t$ is the lower term)

$$\begin{aligned}(w_{i,0}, t) &= \left(r_i, w_{i,1} \cdot \left(\frac{s'}{s^*} - y\right)^{-1}\right) \\ &= \left(r_i, s_i \cdot \sum_{j=0}^{i-1} y_j \cdot \left(\frac{s_j \cdot \sum_{j=0}^{i} y_j}{s^*} - y_i\right)^{-1}\right)\end{aligned}$$

where $s_j$ is the answer of the oracle on the $j$-th session on input the corresponding message $m_j$. If we set $s^* = s_j$ then we have

$$\begin{aligned}(w_{i,0}, t) &= \left( r_i, s_i \cdot \sum_{j=0}^{i-1} y_j \cdot \left( \sum_{j=0}^{i} y_j - y_i \right)^{-1} \right) \\ &= (r_i, s_i)\end{aligned}$$

which is a valid signature on $m_i$ (since it is the output of the signing oracle) and the release would be successful. So this cannot happen and we can assume that $s^* \neq s_j$. A similar argument (substituting $t$ with $t'$) can be used to show that it must be the case that $s^* \neq -s_j$. Since each message uniquely identifies a session (the same message is never queried twice to the interface **Sign(*m*,*y*)**) this implies that $(\sigma^*, (m^*, \mathsf{pk}^*))$ is a valid forgery. By assumption this happens with probability at leasts $\frac{1}{q \cdot \mathsf{poly}(\lambda)}$, which is a contradiction and proves that $\mathsf{Pr}\left[\mathsf{cheat} \mid \mathcal{H}_3^*\right] \leq \mathsf{negl}(\lambda)$. Since the experiments $\mathcal{H}_3$ and $\mathcal{H}_4$ differ only when $\mathsf{cheat}$ happens (and $\mathcal{H}_4$ aborts), we are only left with showing the indistinguishability of $\mathcal{H}_3$ and $\mathcal{H}_3^*$.

**Lemma 15.** *For all PPT distinguishers $\mathcal{E}$ it holds that*

$$\mathsf{EXEC}_{\mathcal{H}_3, \mathcal{A}, \mathcal{E}} \approx \mathsf{EXEC}_{\mathcal{H}_3^*, \mathcal{A}, \mathcal{E}}.$$

*Proof.* The proof consists of the description of the simulator for the interactive lock algorithm. The simulator for the key generation phase is identical as the one described in the work of Lindell [34]. In the following we describe the two simulators for the locking protocol depending on whether the honest adversary is playing the role of the "left" or "right" party. For each zero-knowledge proof, both the simulators implicitly check that the given witness is valid and abort if this is not the case.

1. Left corrupted: Prior to the interaction the simulator is sent $(Y, y, (\mathsf{prove}, \{\exists y^* \text{ s.t } y^* \cdot G = Y\}, y^*))$, which is the state corresponding to the execution of the lock. After agreeing on a message $m$, the simulator sends $(\mathsf{com}, \mathit{sid})$ to $\mathcal{A}$, for a random *sid*. The simulator also queries the interface **Sign** on input $m, y^*$ and receives a signature $\sigma = (r, s)$. The simulator sets $R = \frac{H(m)}{s} \cdot G + \frac{r}{s} \cdot \mathsf{pk}$. At some point of the execution $\mathcal{A}$ sends $(R_0, R_0', (\mathsf{prove}, \{\exists r_0 \text{ s.t } r_0 \cdot G = R_0 \text{ and } r_0 \cdot Y = R_0'\}, r_0))$. Then the simulator samples a $\rho \leftarrow \mathbb{Z}_{q^2}$ and computes $c' \leftarrow \mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}, s \cdot r_0 + \rho q)$. Then it provides the attacker with
$$\left( \left( \mathsf{decom}, \mathit{sid}, \left( R^* = (r_0)^{-1} \cdot R, R_1 = y^{-1} \cdot R^*, \left( \mathsf{proof}, \mathit{sid}, \left\{ \begin{array}{c} \exists r^* \text{ s.t } r^* \cdot G = R_1 \\ \text{and } r^* \cdot Y = R^* \end{array} \right\} \right) \right) \right), \right. \\ \left. R_1, R^*, c' \right).$$
The rest of the execution is unchanged.

The executions are identical except for the way $c'$ is computed. In order to show the statistical proximity we invoke a the following helping lemma.

**Lemma 16.** *[34] For all $(r, s, p) \in \mathbb{Z}_q$ and for a random $\rho \in \mathbb{Z}_{q^2}$, the distributions $\mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}, r \cdot s \bmod q + pq + \rho q)$ and $\mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}, r \cdot s \bmod q + \rho q)$ are statistically close.*

In the real world $c'$ is computed as $\mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}, r \cdot s \bmod q + pq + \rho q)$, for some $p$ which is bounded by $q$ since the only operation performed without modular reduction are one multiplication and one addition, which cannot increase the result by more than $q^2$. Since the distribution $\mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}, r \cdot s \bmod q + \rho q)$ is identical to the simulation, the indistinguishability follows.

2. Right corrupted: Prior to the interaction the simulator is sent $(Y, y, (\mathsf{prove}, \{\exists y^* \text{ s.t } y^* \cdot G = Y\}, y^*))$, which is the state corresponding to the execution of the lock. After agreeing on a message $m$, the simulator is given
$$\left( \mathsf{com}, \mathit{sid}, \left( R_1, R_1', \left( \mathsf{prove}, \mathit{sid}, \left\{ \begin{array}{c} \exists r_1 \text{ s.t } r_1 \cdot G = R_1 \text{ and} \\ r_1 \cdot Y = R_1' \end{array} \right\}, r_1 \right) \right) \right)$$

by $\mathcal{A}$. The simulator then queries the interface **Sign** on input $m, y^*$ and receives a signature $\sigma = (r, s)$. Then it sets $R = \frac{H(m)}{s} \cdot G + \frac{r}{s} \cdot \mathsf{pk}$ and $R^* = R - (R_1 + Y)$ and sends $(R_0 = y^{-1} \cdot R^*, R^*, (\mathsf{proof}, sid, \{\exists r^* \text{ s.t } r^* \cdot G = R_0 \text{ and } r^* \cdot Y = R^*\}))$ to $\mathcal{A}$. The attacker sends $((\mathsf{decom}, sid), c')$ in response. The simulator checks

$$\mathsf{Dec}_{\mathsf{HE}}(\mathsf{sk}, c') = \tilde{r} \cdot r \cdot (r_1)^{-1} + H(m) \cdot r_1^{-1} \bmod q,$$

where $\tilde{r}$ was sampled in the key generation algorithm. If the check holds true, the simulator sends $s$ to $\mathcal{A}$.

The distributions induced by the simulator is identical to the real experiment except for the way $c$ is computed. Towards showing indistinguishability, consider the following modified simulator, that is given the oracle $\mathcal{O}(c', a, b)$ as defined in the following security experiment of the Paillier encryption scheme.

$\mathsf{Exp} - \mathsf{ecCPA}^{\mathcal{A}}_{\mathsf{HE}}(\lambda)$ :

$(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KGen}_{\mathsf{HE}}(1^\lambda)$

$(w_0, w_1) \leftarrow_\$ \mathbb{Z}_q$

$Q = w_0 \cdot G$

$b \leftarrow_\$ \{0, 1\}$

$c \leftarrow \mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}, w_b)$

$b' \leftarrow \mathcal{A}(\mathsf{pk}, c, Q)^{\mathcal{O}(\cdot,\cdot,\cdot)}$

where $\mathcal{O}(c', a, b)$ returns 1 iff $\mathsf{Dec}_{\mathsf{HE}}(\mathsf{sk}, c') = a + b \cdot w_b$

return 1 iff $b = b'$

Instead of performing the last check, the simulator queries the oracle on input $(c', a = H(m) \cdot r_1^{-1}, b = r \cdot (r_1)^{-1})$. It is clear that the modified simulator accepts if and only if the simulator described above accepts. Assume towards contradiction that the modified simulator can be efficiently distinguished from the real world experiment. Then we can reduce to the security of Paillier as follows: On input $(\mathsf{pk}, c, Q)$, the reduction simulates the inputs of $\mathcal{A}$ as described in the modified simulator using the input $\mathsf{pk}$, $Q$, and $c$ as the corresponding variables. It is easy to see that the reduction is efficient. Note that if $b = 0$ then we have that $c = \mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}, w_0)$ and $Q = w_0 \cdot G$, which is identical to the real world execution. On the other hand if $b = 1$ then it holds that $c = \mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}, w_1)$ and $Q = w_0 \cdot G$, where $w_1$ is uniformly distributed in $\mathbb{Z} - q$, which is identical to the (modified) simulated experiment. This implies that the modified simulation is computationally indistinguishable from the real world experiment. Since the modified simulation and the simulation (as described above) are identical to the eyes of the adversary, the validity of the lemma follows. □

This concludes the proof of lemma 14. □

**Lemma 17.** *For all PPT distinguishers $\mathcal{E}$ it holds that*

$$\mathsf{EXEC}_{\mathcal{H}_4, \mathcal{A}, \mathcal{E}} \approx \mathsf{EXEC}_{\mathcal{H}_5, \mathcal{A}, \mathcal{E}}.$$

*Proof.* Let $q \in \mathsf{poly}(\lambda)$ be a bound on the number of interactions. Let $\mathsf{cheat}$ denote the event that triggers an abort in $\mathcal{H}_5$ but not in $\mathcal{H}_4$. In the following we are going to show that $\mathsf{Pr}[\mathsf{cheat} \mid \mathcal{H}_4] \leq \mathsf{negl}(\lambda)$, thus proving the indistinguishability of $\mathcal{H}_4$ and $\mathcal{H}_5$. Assume that the converse is true, then we can construct the following reduction against the discrete logarithm problem (which is implied by the sEUF of ECDSA): On input some $Y^* \in \mathbb{G}$, the reduction guesses a session $j \in [1, q]$ and some index $i \in [1, n]$. The setup algorithm of the $j$-th session is modified as follows: $Y_i$ is set to be $Y^*$. Then, for all $\iota \in [i-1, 0]$, the setup samples some $y_\iota \in \mathbb{Z}_q$ and returns $(Y_\iota = Y_{\iota+1} - (y_\iota) \cdot G, Y_{\iota+1}, y_\iota)$. The setup samples a random $y_i \in \mathbb{Z}_q$ and sets $Y_{i+1} = y_i \cdot G$. Then, for $\iota \in [i+1, n-1]$, the setup samples $y_\iota \in \mathbb{Z}_q$ and returns $(Y_\iota, Y_\iota + y_\iota \cdot G, y_\iota)$. The nodes $(U_1, \ldots, U_{n-1})$ are given the corresponding output (except for $U_i$) and $U_n$ is given $(Y_{n-1}, \sum_{j=i}^{n-1} y_j)$. If the node $U_i$ is requested to release the lock, the reduction aborts. At some point of the execution the adversary $\mathcal{A}$ outputs some $k^* = (r^*, s^*)$. The reduction parses $s^R = (s', m, \mathsf{pk})$ as the updated state of $U_i$ then checks the following:

1. $\left(\frac{s}{s^*} + y_{i-1}\right) \cdot G = Y^*$
2. $-\left(\frac{s'}{s^*} + y_{i-1}\right) \cdot G = Y^*$

and returns the LHS term of the equation that satisfies the relation.

The reduction is clearly efficient and, whenever $j$ and $i$ are guessed correctly, the reduction does not abort. Since the $\mathbb{G}$ is abelian and the $U_i$ is honest, the distribution induced by the modified setup algorithm is identical to the original to the eyes of the adversary. Recall that $\mathsf{cheat}$ happens only in the case where $k^*$ is a valid opening for $\ell_i$ and the release algorithm is successful on input $k^*$ (if the last condition is not satisfied both $\mathcal{H}_4$ and $\mathcal{H}_5$ abort). Substituting, we have that $s'$ is of the form $\frac{x_0 \cdot x_1 \cdot r_x + H(m)}{r_0 \cdot r_1} = \tilde{s} \cdot y$, where $R' = r_0 \cdot r_1 \cdot Y_{i-1} = (r_x, r_y)$, for some $y \in \mathbb{Z}_q$. Since the release is successful, then it must be the case that $(r_x, \tilde{s})$ is a valid ECDSA signature on the message $m_{i-1}$ (agreed by the two parties in the locking algorithm for $\ell_{i-1}$). This implies that $y \cdot G = Y_{i-1}$. As argued in the proof of lemma 14, if $s^* \neq \tilde{s}$ and $s^* \neq \tilde{s}$, then we have an attacker against the strong unforgeability of the signature scheme. It follows that $s^* = \tilde{s}$ or $s^* = -\tilde{s}$ with all but negligible probability. Substituting we have

$$\begin{aligned}
\left(\frac{s'}{s^*} + y_{i-1}\right) \cdot G &= \left(\frac{\tilde{s} \cdot y}{s^*} + y_{i-1}\right) \cdot G \\
&= \frac{\tilde{s} \cdot y}{s^*} \cdot G + y_{i-1} \cdot G \\
&= y \cdot G + y_{i-1} \cdot G \\
&= Y_{i-1} + y_{i-1} \cdot G \\
&= (Y^* - y_{i-1} \cdot G) + y_{i-1} \cdot G \\
&= Y^*
\end{aligned}$$

which implies that condition (1) holds if $s^* = \tilde{s}$. For the other case

$$\begin{aligned}
-\left(\frac{s'}{s^*} + y_{i-1}\right) \cdot G &= -\left(\frac{\tilde{s} \cdot y}{s^*} + y_{i-1}\right) \cdot G \\
&= -\frac{\tilde{s} \cdot y}{s^*} \cdot G + y_{i-1} \cdot G \\
&= y \cdot G + y_{i-1} \cdot G \\
&= Y_{i-1} + y_{i-1} \cdot G \\
&= (Y^* - y_{i-1} \cdot G) + y_{i-1} \cdot G \\
&= Y^*
\end{aligned}$$

which means that condition (2) is satisfied if $s^* = -\tilde{s}$. Since, by assumption, this happens with probability at least $\frac{1}{q \cdot n \cdot \mathsf{poly}(\lambda)}$ we have a successful attacker against the discrete logarithm problem. This proves our statement. □

This concludes our proof. □