PayMo: Payment Channels For Monero

Sri Aravinda Krishnan Thyagarajan [*1], Giulio Malavolta [†2], Fritz Schmidt [‡1], Dominique Schröder [§1]

[1]Friedrich Alexander Universität Erlangen-Nürnberg
[2]Max Planck Institute of Security and Privacy

November 16, 2020

Decentralized cryptocurrencies still suffer from three interrelated weaknesses: Low transaction rates, high transaction fees, and long confirmation times. Payment Channels promise to be a solution to these issues, and many constructions for real-life cryptocurrencies, such as Bitcoin, are known. Somewhat surprisingly, no such solution is known for Monero, the largest privacy-preserving cryptocurrency, without requiring system-wide changes like a hard-fork of its blockchain.

In this work, we close this gap by presenting PayMo, the first payment channel protocol that is fully compatible with Monero. PayMo does not require any modification of Monero and can be readily used to perform off-chain payments. Notably, transactions in PayMo are identical to standard transactions in Monero, therefore not hampering the coins' fungibility. Using PayMo, we also construct the first fully compatible secure atomic-swap protocol for Monero: One can now securely swap a token of Monero with a token of several major cryptocurrencies such as Bitcoin, Ethereum, Ripple, Cardano, etc. Before our work, it was not known how to implement secure atomic swaps protocols for Monero without forcing a hard fork.

Our main technical contribution is a new construction of an efficient verifiable timed linkable ring signature, where signatures can be hidden for a pre-determined amount of time, in a verifiable way. Our scheme is fully compatible with the transaction scheme of Monero and it might be of independent interest.

We implemented PayMo and our results show that, even with high network latency and with a single CPU core, two regular users can perform up to 93500 payments over a span of 2 minutes (the block production rate of Monero). This is approximately five orders of magnitude improvement over the current payment rate of Monero.

[*]E-mail: thyagarajan@cs.fau.de
[†]E-mail: giulio.malavolta@hotmail.it
[‡]E-mail: fritz.schmid@fau.de
[§]E-mail: dominique.schroeder@fau.de

Contents

1 Introduction

Modern cryptocurrencies, such as Bitcoin or Monero, realize the digital analog of a fiat currency without a trusted central authority. They typically consist of two main components: (i) A public ledger that publishes all transactions, and (ii) a transaction scheme that describes the structure and validity of transactions. In comparison to traditional centralized solutions, decentralized cryptocurrencies suffer from three weaknesses: First, they have a relatively low transaction rate, for example, the current transaction rate of Bitcoin is about four transactions per second while it is 0.1 transactions per second in case of Monero [1]. Second, the transaction fees are relatively high, about 0,60$ per transaction in the case of Bitcoin and about 0.25$ in Monero [2]. Third, the confirmation of a transaction takes (on average) one hour in case of Bitcoin and 20 minutes in the case of Monero. Payment Channels (PC) [3] and its generalization Payment Channel Networks (PCN) [3]–[6] have emerged as one of the most promising solutions to mitigate these issues and have been widely deployed to scale payment in major cryptocurrencies, such as Bitcoin [7], Ethereum [8] or Ripple [9]. These solutions are commonly referred to as *layer 2* or *o -chain* solutions.

A PC allows a pair of users to perform multiple payments without committing every intermediate payment to the blockchain. Abstractly, a PC consists of three phases: (i) Two users Alice and Bob open a payment channel by adding a single transaction to the blockchain. This transaction is a promise from Alice that she may pay up to a certain amount of coins to Bob, which he must claim before a certain time T. (ii) Within this time window, Alice and Bob may send coins from the joint address to either of them by sending a corresponding transaction to the other user. (iii) The channel is considered closed when one of those payment transactions is posted on the chain, thus spending coins from the joint address. While realizing PCs for Bitcoin is an established task due to the functionality available in the Bitcoin scripting language, several challenges arise when considering privacy-preserving cryptocurrencies like Monero or Zcash [10]. Bolt [11] is a PC proposal for Zcash while Moreno-Sanchez et al. [12] developed a PC protocol for Monero. However, their proposal has various shortcomings (see below for a detailed discussion) and, as a consequence, is unlikely to be integrated into Monero in the near future. In this work, we aim to close this gap by constructing a PC protocol that is fully compatible with Monero and can be readily used to perform off-chain transactions.

Brief Look into Monero. Monero is the largest privacy-preserving cryptocurrency [13], and the notion of privacy it offers is that: Any external observer cannot learn who the sender or the receiver of a transaction are and the amount of coins being transferred. Monero achieves these properties with Ring Confidential Transactions (RingCT) as its cryptographic bedrock. Briefly, a RingCT is a transaction scheme where the sender of the transaction 'hides' his key in an anonymity set (ring). Comparing with Bitcoin, where transactions have typically one source address, the amount in plain, one or two recipient address(es) and a simple signature (ECDSA), the RingCT based Monero transactions contain a ring of addresses, destination addresses, commitments to amounts, related consistency proofs and a (linkable ring) signature, making the transactions considerably larger in size. Moreover, the size of a Monero transaction grows linearly with the size of the anonymity set. This results in a major setback to the scalability of Monero and often requires users to make tough choices between better privacy (high transaction fee) and smaller transactions (lower transaction fees). There has been a line of research [14]–[17] that proposes new and efficient RingCT constructions that result in smaller transaction sizes. These approaches help to increase privacy because they support larger ring sizes and therefore do not increase the transaction fees. However, the central three issues that PCs are addressing remain open: Increasing the transaction rate, reducing the transaction fees in general, and therefore supporting fast micro-payments, as well as fast verification time. Moreover, all these on-chain solutions require system-wide changes in the Monero protocol, and it is unclear if Monero will fork and adapt to one of these schemes.

Unfortunately, layer 2 solutions (such as PCs) proposed for Bitcoin do not extend to Monero as they crucially rely on some scripting functionalities offered by the Bitcoin blockchain, that are absent in Monero. To the best of our knowledge, all layer 2 solutions that have been proposed for Monero (like that of [12]) are not compatible with the existing implementation and require a hard fork.

1.1 Our Contribution

The contributions of this work can be summarized as follows.

1. We propose PayMo (Section 6), the first payment channel protocol that is fully compatible with today's implementation of the transaction scheme of Monero (Section 4). A notable feature of our solution is that PC transactions are syntactically identical to standard transactions in Monero, thus retaining the *fungibility*[1] of the Monero coins.

2. At the heart of our proposal is a new cryptographic primitive, called *verifiable timed linkable ring signatures* (VTLRS), that we define and construct (Section 5). Our solution relies on well-established cryptographic assumptions and our techniques used in building a VTLRS may be of independent interest.

3. We show that PayMo also enables the first secure atomic-swap protocol for Monero (Section 7). With our solution, one can securely (atomically) exchange Monero tokens with other prominent currency tokens like that of Bitcoin, Ethereum, Ripple, Cardano, etc.

4. We demonstrate the practicality of our approach by benchmarking PayMo (Section 8). Our analysis shows that PayMo can be used on today's hardware by Monero's users. In terms of performance, at its full power, PayMo supports close to 93500 payments over a span of 2 minutes between two regular users with one CPU core each. Here 2 minutes is the block production rate of Monero. This is a significant increase in processing of payments in Monero which currently supports one payment from a single address per 2 minutes.

1.2 Related Work and Discussion

In the following we compare our approach with existing systems and we discuss some of the choices behind practical aspects of our design.

Comparison with [12]. The first PC proposal for Monero was recently put forth by Moreno-Sanchez et al. [12], however their solution requires a hard fork with major changes to the Monero transaction scheme and is *not* backward compatible. Specifically, a PC in their protocol is a joint address comprising two public keys (left and right) and the linkability tag of such a joint address is generated *differently* from the current specification of Monero, to facilitate either keys to spend from the joint address and preserve double-spend linkability. They also require an explicit time-lock functionality for the joint address which allows the left key to spend before the time-lock expires and the right key after the expiry.

We stress that, even assuming a that Monero will fork in the near future to integrate their scheme, the adoption procedure still requires one to solve some challenges: Since the tag generation in [12] is different from the currently used algorithm, one needs to perform massive system-wide changes to the Monero protocol itself, requiring *every* Monero user to spend from their existing unspent keys (with old tag generation) to a new key (with the new tag generation) during a specific time interval. And after this time interval spending is allowed only with the new tag generation algorithm and all spending attempts with the old tag will be rejected. This is highly undesirable as it requires *every* Monero user to be online and make transactions, and any user unable to make this switch during this time interval loses his coins permanently. At the time of writing, it is not clear whether such a proposal will be implemented in Monero [18].

An additional limitation of their proposal in terms of transaction privacy is that, since the timelock information of the payment channel or the payment channel expiry time is publicly available on-chain, it affects the fungibility of the Monero tokens involved in the payment channel. It could also lead to censorship from miners, as a miner could refuse to accept a payment channel transaction if it is too close or too far away from the corresponding channel expiry time.

On the contrary, PayMo does not require any changes to the transaction scheme of Monero, nor it requires to add any functionality to the scripting language. Any interested pair of users can run PayMo without the knowledge of any other user in the Monero system. Furthermore, any PayMo related transaction posted on-chain is *identical* to posting any other regular transaction in Monero. Consequently, our PC protocol is readily usable in Monero today, without making any system-wide modifications. However we note that our PC protocol cannot be extended to a PCN without modifying some features of

[1] Fungibility is a property of a currency whereby two units can be substituted in place of one another. This means that all tokens are essentially the same in value and therefore there is no speciality for some coins due to some transactions operating on them.

the Monero blockchain, while in [12], they have a PCN protocol (still requiring the same modifications to Monero as in their PC protocol).

Time-Lock Puzzles. One of the main challenges that we face in constructing a fully-compatible PC for Monero is the lack of a *time-lock* functionality of its scripting language. To "simulate" this functionality off-chain, our constructions will resort to the usage of time-lock puzzles [19]. This approach translates waiting time into performing some inherently sequential computational task, such as repeated squaring on groups of unknown order. The drawback of this solution is that PC users are required to run a background process for each PC that they are part of.

We believe that the benefits offered by PCs (e.g. the significant increase in transaction volume) largely outweigh the burden of maintaining an additional background computation running. Considering that the typical time-lock duration for channels is in the order of a couple of days, the computational cost associated with it appears to be modest, especially in comparison with the one required to run proof-of-work based consensus. To further mitigate this issue, we also propose an approach to batch the solution of multiple puzzles into a single one (see Section 5.2 for details).

In terms of practicality, the sequential function that we consider (repeated squaring) has been the subject of a large academic [20]–[22] and industrial [23]–[25] effort to study its exact complexity on commodity machines. Functionalities that rely on repeated computation of squares have been integrated in the Chia network [26], [27] and their adoption is currently being considered by Ethereum [25], [28], [29].

It is possible that some parties possess hardware with a more powerful processing power, and could solve time-lock puzzles much earlier than other parties. In a recent work [30], it was shown that this hardware disparity could result in a 60% difference in the time taken to solve a time-lock puzzle. This could affect security for a party who is caught off-guard by another party who ended up solving the puzzle much earlier than expected. One way to deal with this problem is for the potential "victim" party to take action well before a conservative estimate of when the puzzle might be solved. Especially in the case of PayMo where Alice has to solve a time-lock puzzle for time $\mathbf{T}$ to close a payment channel with Bob, Bob must make sure to close the channel well ahead of the time $\mathbf{T}$ that he expects Alice to solve a time-lock puzzle and close the channel.

Uni-directional vs Bi-directional Channels. While the protocol in [12] with the system-wide changes to Monero supports bi-directional PCs, PayMo only supports uni-directional PCs. That is, payments from Alice to Bob and from Bob to Alice require opening two separate channels. Bi-directional are desirable in practice as they increase the capacity of the payment network and achieving such channels in a Monero compatible way is an interesting open problem.

Other Related Work. Payment Channels and Payment Channel Networks [3], [4] have been proposed as solutions to address the problem of high frequency payments in Bitcoin. Typical proposals use a special *Hash Time-Lock Contract (HTLC)* that lets a user get paid if he produces a pre-image of a certain hash value before a specific time, referred to as the time-lock of the payment. Malavolta et al. [5] propose a PCN protocol that does not rely on HTLC and offers better on-chain privacy using a new tool called Anonymous Multi Hop Locks (AMHL). Bolt [11] is a payment channel protocol specially tailored for Zcash [10] which uses zk-SNARKs [31]–[33], and is not compatible with the transaction scheme of Monero, which is the focus of this work. Specifically, BOLT relies on a relative timelock script offered by the blockchain layer (during channel closure) that is not offered by Monero. As a result, it is unclear how to adapt BOLT to Monero. A generalisation of a payment channel with complex conditional payments is a *state channel* [34]–[36] that requires highly expressive scripting functionalities from the underlying blockchain. Since Monero does not offer such expressive scripting functionalities and since our focus is only on fast micro-payments, we focus only on payment channels.

1.3 Organization

The rest of the paper is organized as follows. In Section 2, we give a brief technical overview of our core techniques and intuition into our PayMo protocol. In Section 3 we introduce the building blocks required for our protocols and in Section 4 we formally present the definitions, security and construction of the transaction scheme of Monero. We present the formal definitions and the construction of VTLRS in Section 5. In Section 6 and Section 7 we present the PayMo protocol and our atomic swap protocol, respectively. Finally, we benchmark our protocol and present the experimental results in Section 8.

2 Technical Overview

For ease of presentation, we consider a simplified representation of a transaction in Monero consisting of: A ring of one-time public keys (addresses) $\mathcal{R}$, a linkability tag tag, a signature σ and the target public key (recipient address). We omit other components of a Monero transaction as our tools and techniques only deal with the above components and it can be naturally extended to the current implementation of Monero with all components in place[2].

In the following outline we first introduce the notion of *Verifiable Timed Linkable Ring Signature* (VTLRS) and we show a construction compatible with the transaction scheme in Monero. Then we describe how to leverage VTLRS to construct payment channels (PCs) that are fully compatible with Monero. Finally, we discuss how to extend our protocol to support atomic swaps with tokens from other currencies.

2.1 VTLRS for Monero

We introduce and formalize the notion of *Verifiable Timed Linkable Ring Signature (VTLRS)*. A VTLRS lets a user create a timed commitment of a linkable ring signature on a message (transaction) such that the recipient of the commitment can force the opening of the commitment and learn the signature only after a pre-specified time $\mathbf{T}$. The recipient also receives a proof that convinces him that the force opening would indeed reveal the valid linkable ring signature on the message. Our construction of VTLRS is compatible with the linkable ring signature transaction scheme that is currently implemented in Monero, where the message is now a Monero transaction.

On a high-level, to commit to a VTLRS, the committer takes a (linkable ring) signature on a transaction tx and *encodes* it into a time-lock puzzle [19], which keeps it hidden until time $\mathbf{T}$. To convince the verifier that the puzzle contains a valid signature on the transaction tx, the committer also computes a non-interactive zero-knowledge (NIZK) proof for such a statement. The challenge here is to design an efficient NIZK proof that certifies the validity of the encoded signature. General solutions exist [37], [38] but they are tailored to common signature schemes, like Schnorr, ECDSA [30], [37] and BLS [30].

To design an efficient NIZK, we adopt a cut-and-choose approach, where the signature is redundantly encoded into many puzzles and the validity can be checked by revealing the random coins corresponding to a subset of them. If implemented naively, this would clearly compromise the privacy of the signature. Instead, we harness the structural properties of signature of Monero to reveal only isolated components, while at the same time keeping the signature hidden. More specifically, the committer computes a t-out-of-n secret sharing of a special component of the signature. Given a $t-1$ subset of the shares (which are revealed by the cut-and-choose) the verifier can check whether these opened shares are valid shares of the signature component and that the opened puzzles were indeed valid puzzles (using the random coins supplied by the prover). If the check is successful, then the verifier is convinced that *at least one* of the unopened puzzles contains a well-formed share, which is enough to reconstruct a valid signature. The scheme is made non-interactive using the Fiat-Shamir transformation [39].

We then instantiate the time-lock puzzles with [40] and use the homomorphic properties of such a scheme to combine puzzles in such a way that the computation needed to force open is the same as that to force open *a single puzzle*. We stress that the use of homomorphism is not just crucial for the efficiency of the solver (verifier), but is also important for security. Without homomorphism, a user with $\tilde{n} = n - (t-1)$ processors can solve $\tilde{n}$ puzzles in parallel and in total time $\mathbf{T}$. On the other hand, users with less number of processors will have to solve the puzzles one after another, thereby spending more time than $\mathbf{T}$ time. This could lead to scenarios in PCs where an adversarial party has an unfair advantage with respect to an honest user and could post a valid transaction ahead of time, effectively stealing coins.

2.2 Payment Channels in Monero

Equipped with our efficient VTLRS scheme, we show how Alice and Bob can run a payment channel protocol to make payments (i.e., a joint address where both signatures are needed in order to perform a

[2] A Monero transaction is based on RingCT which additionally consists of Homomorphic commitments to hide the amounts and range proofs to prove that the commitments are well-formed. Precise details of the Monero transaction can be found in [17].

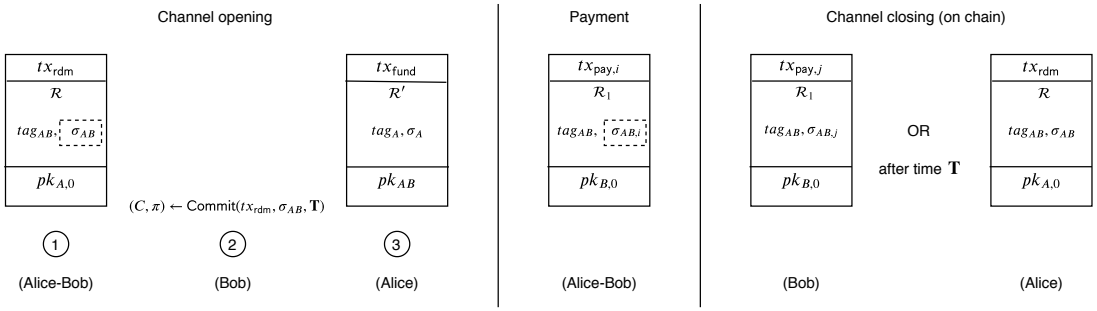

Figure 1: Three phases to a (uni-directional) payment channel protocol between Alice and Bob in PayMo. The phases are split vertically. The channel opening phase has three steps. Steps run individually and jointly through interaction are denoted with (Alice) or (Bob) and (Alice-Bob) respectively. Signature σ_{AB} inside a dotted box indicates that only Bob learns the signature after interaction with Alice.

transaction). A pictorial description of our PC protocol (PayMo) is given in Figure 1 and its three main subroutines (channel opening, payment, and channel closing) are discussed below briefly.

Channel opening. Alice and Bob jointly generate a spending key pk_{AB} and a redeem transaction tx_{rdm} that spends the coins from pk_{AB} (belonging to some ring $\mathcal{R}$) to some address of Alice $pk_{A,0}$. The transaction tx_{rdm} also contains the joint tag tag_{AB} that is needed to prevent double-spending. Note that pk_{AB} is an address that is not yet present on the chain. Bob then generates a VTLRS of the signature σ_{AB} on tx_{rdm} with timing hardness $\mathbf{T}$. Bob gives the VTLRS commitment and proof (generated using $\mathsf{Commit}(tx_{\mathsf{rdm}}, \sigma_{AB}, \mathbf{T})$ in Figure 1) to Alice, who then posts the transaction tx_{fund} on the Monero blockchain. Such a transaction initializes the channel by sending funds from one of her addresses pk_A to the joint key pk_{AB}. The channel is now created and initialized on-chain and its expiry time is set to $\mathbf{T}$. Note that after time $\mathbf{T}$ Alice will be able to recover the signature tx_{rdm} and therefore redeem the remaining funds in the address pk_{AB}, if any.

Payment. When Alice wishes to pay Bob, they jointly generate a transaction $tx_{\mathsf{pay},i}$ for the i-th payment. $tx_{\mathsf{pay},i}$ spends from pk_{AB} (in some ring $\mathcal{R}_1$) and sends it to some address of Bob $pk_{B,0}$. They jointly generate a signature $\sigma_{AB,i}$ on $tx_{\mathsf{pay},i}$, in such a way that only Bob learns the signature.

Channel closing. If Bob wishes to close the channel, he takes the last exchanged transaction payment $tx_{\mathsf{pay},j}$ with Alice and posts it along with $\sigma_{AB,j}$ on the Monero blockchain. In case Bob has not posted any such payment and time $\mathbf{T}$ has passed, Alice by then learns σ_{AB} from the VTLRS on tx_{rdm} (that was given by Bob during channel opening). Alice can now post tx_{rdm} and σ_{AB} on the Monero blockchain and redeem the coins from the channel. In either case, once a transaction spending from pk_{AB} is posted on-chain, the payment channel is considered closed.

2.3 Atomic Swaps for Monero

Consider a scenario where Alice has a Monero token that she wants to swap with a token of another currency (Bitcoin, Ethereum, Ripple, etc.) held by Bob, which we denote by

$$B \to^C A \to^M B.$$

The atomicity guarantee in this swap states that Bob transfers a token of the currency to Alice if and only if Alice transfers a token of Monero to Bob.

The first step to build such a protocol is to construct an atomic multi-hop lock (AMHL) [5] protocol compatible with our VTLRS scheme. This allows one to set a chain of conditional payments that are "unlocked" only once the designated receiver is reached. Equipped with this tool, our approach consists in setting up payment channels between the parties in both currencies: On the Monero front the (uni-directional) payment channel is denoted by pk^M_{AB}, and on the other currency, the channel is denoted by pk^C_{BA}. Then, using AMHL, the parties setup payment locks on payment transactions from these channels, which enforces the conditional payment: If Bob posts a transaction paying the coin from pk^M_{AB}, then Alice recovers enough information to compute a valid transaction from pk^C_{BA}, by the security

of the AMHL scheme. On the other hand, if Bob does not post any transaction, eventually the channels expire and the respective parties recover their coins.

One crucial subtlety that we need to address is to ensure that the timelock t_M of the payment channel pk^M_{AB} is less than the timelock t_C of the channel pk^C_{BA}. Precisely we want $t_C = t_M + \delta$ for some conservatively chosen $\delta > 0$. This is because, Alice wants to ensure that after Bob releases the lock of AMHL and gets the Monero token from pk^M_{AB}, she has some time (δ) to release her lock and get the currency token from pk^C_{AB}. This means that, the timing hardness $\mathbf{T}$ of VTLRS that is used in (the PayMo) channel pk^M_{AB} is set such that the associated time t_M satisfies $t_M := t_C - \delta$.

3 Preliminaries

We denote by $\lambda \in \mathbb{N}$ the security parameter and by $x \leftarrow \mathcal{A}(\mathsf{in}; r)$ the output of the algorithm $\mathcal{A}$ on input in using $r \leftarrow \{0,1\}^*$ as its randomness. We omit this randomness and only mention it explicitly when required. We denote the set $\{1, \ldots, n\}$ by $[n]$. We model parallel algorithms as Parallel Random Access Machines (PRAM): In this model multiple processors are attached to a single block of memory and n number of processors can perform independent operations on n number of data in a particular unit of time. We consider *probabilistic polynomial time* (PPT) machines as efficient algorithms. We briefly recall the cryptographic primitives used in our protocols and

Time-Lock Puzzles. Time-lock puzzles [19] allow one to conceal a secret for a certain amount of time $\mathbf{T}$. *Homomorphic Time-Lock Puzzles (HTLPs)* [40] allow one to perform homomorphic computation on honestly generated puzzles. It consists of a setup algorithm (PSetup), that takes as input a time hardness parameter $\mathbf{T}$ and outputs public parameters of the system pp, a puzzle generation algorithm (PGen) that, on input the public parameter and a message, generates the corresponding puzzle. One can then evaluate homomorphically functions over encrypted messages (PEval) and solve the resulting puzzle in time $\mathbf{T}$ (PSolve). The security requirement is that for every PRAM adversary $\mathcal{A}$ of running time $\leq \mathbf{T}^{\epsilon}(\lambda)$ the messages encrypted are computationally hidden.

In [40], Malavolta and Thyagarajan show an efficient construction that is linearly homomorphic over the ring $\mathbb{Z}_{N^s}$, where N is an RSA modulus and s is any positive integer. The scheme is perfectly correct and it is secure against the sequential squaring assumption [19].

Non-Interactive Zero-Knowledge. Let $R : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}$ be an NP relation with corresponding NP-language $\mathcal{L} := \{stmt : \exists wit \ s.t. \ R(stmt, wit) = 1\}$. A non-interactive zero-knowledge proof (NIZK) [41] system for $\mathcal{L}$ is initialized with a setup algorithm $\mathsf{Setup}(1^\lambda)$ that outputs a common reference string crs. A prover can show the validity of a statement $stmt$ with a witness wit by invoking $\mathcal{P}_{\mathsf{NIZK},\mathcal{L}}(crs, stmt, wit)$, which outputs a proof π. The proof π can be efficiently checked by the verification algorithm $\mathcal{V}_{\mathsf{NIZK},\mathcal{L}}(crs, stmt, \pi)$. A NIZK proof for language $\mathcal{L}$ is simulation extractable if one can extract a valid wit from adversarially generated proofs, even if the adversary sees arbitrarily many simulated proofs. A NIZK must also be zero knowledge in the sense that nothing beyond the validity of the statement is leaked to the verifier.

Threshold Secret Sharing. Secret sharing is a method of creating shares of a given secret and later reconstructing the secret itself only if given a threshold number of shares. Shamir [42] proposed a threshold secret sharing scheme where the sharing algorithm takes a secret $s \in \mathbb{Z}_q$ and generates shares $(s_1, \ldots, s_n)$ each in $\mathbb{Z}_q$. The reconstruction algorithm takes as input at least t shares and outputs a secret s. The security of the secret sharing scheme demands that knowing only a set of shares smaller than the threshold size does *not* help in learning any information about the secret s.

Universal composability. To model security and privacy in the presence of concurrent executions we resort to the universal composability framework from Canetti [43] extended to support a global setup [44]. In this framework, parties interact with a trusted ideal functionality through secure authenticated channels. A protocol is executed in the presence of any adversary $\mathcal{A}$ that corrupts any subset of the parties, prior to the beginning of the execution (static corruption). Both honest parties and the adversary receive their input from a special entity, called the environment. Let $\mathsf{EXEC}_{\tau,\mathcal{A},\mathcal{E}}$ be the ensemble of the outputs of the environment $\mathcal{E}$ when interacting with the attacker $\mathcal{A}$ and users running protocol τ (over the random coins of all the involved machines).

Definition 3.1 (Universal Composability). *A protocol τ 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.*

Synchrony and Communication. We assume synchronous communication between users, where the execution of the protocol happens in rounds. We model this via an ideal functionality $\mathcal{F}_{\mathsf{clock}}$ as it is done in [45], [46], where all honest parties are required to indicate that are ready to proceed to the next round before the clock proceeds. The clock functionality that we consider is fully described in [44]. This means that all entities are always aware of the given round. We also assume the existence of secure message transmission channels between users modelled by $\mathcal{F}_{\mathsf{smt}}$.

Blockchain. We assume the existence of a blockchain $\mathbb{B}$ (just as in [4], [5], [47]) that we model as a trusted append-only bulletin board: The corresponding ideal functionality $\mathcal{F}_{\mathbb{B}}$ maintains the chain $\mathbb{B}$ locally and updates it according to the transactions between users. The functionality is also parameterized by a signature scheme that lets any user generate key pairs and can post a signed transaction transferring coins from one user to another. At any point in the execution, any user U can send a distinguished message read to $\mathcal{F}_{\mathbb{B}}$, who sends the whole transcript of $\mathbb{B}$ to U. We denote the number of entries of $\mathbb{B}$ (or the length of the chain) by $|\mathbb{B}|$. We refer the reader to [47] for a formal definition of this functionality. In all of our protocols we model the Monero blockchain as $\mathcal{F}_{\mathbb{B}}$.

4 Transaction Scheme of Monero

We review the basic definitions of Linkable Ring Signatures (LRS) following Lai et al. [17]. In contrast to their work, our definitions do not consider the "confidential transaction" part, meaning that we only focus on the signature of the transaction scheme. The extension to confidential transactions follows naturally and we concentrate on LRS for conceptual simplicity.

4.1 Definition

A ring signature [48] scheme allows to sign messages such that the signer is anonymous within a set a possible signers, called the ring. The members associated to the ring are chosen "on-the-fly" by the signer using their public-keys. Linkability [49] means that anonymity is retained unless the same user signing key is used to sign twice. This is achieved by associating a unique linkability tag to each signing key that is revealed while generating a signature.

In a transaction scheme, we have a block of data referred to as a transaction, that determines the amount of coins transferred from one user address (source) to another user address (target) and it is accompanied by an authentication token (signature) of the sending user. Since the sending user is represented through the source address in the transaction, the signature is checked for validity with respect to the source account. Combining linkable ring signatures and a transaction scheme, we have a linkable ring signature based transaction scheme (LRS-TS), where the message signed is the transaction which consists of: A ring of addresses (LRS public keys) and their associated coins (out of which one of the addresses is the source account), and one or more target addresses. The authentication token of the transaction is a linkable ring signature on the transaction (as message), with the ring of addresses as the ring, and the secret authentication key of the source address as the the signing key of the linkable ring signature scheme. To prevent leakage of the source address it is assumed that each address in the ring of addresses have the same amount of associated coins. This assumption can be relaxed with the use of confidential transactions [50] where an account's associated amount is hidden using commitments.

Definition 4.1. *A* Linkable Ring Signature (LRS) *transaction scheme* Π_{LRS} *consists of the* PPT *algorithms* $(\mathsf{Setup}, \mathsf{OTKGen}, \mathsf{TagGen}, \mathsf{Spend}, \mathsf{Vf})$ *which are defined as follows:*

$\underline{pp \leftarrow \mathsf{Setup}(1^\lambda)}$*: The* setup *algorithm outputs the public parameter* pp*.*

$\underline{(pk, sk) \leftarrow \mathsf{OTKGen}(pp)}$*: The* one-time key generation *algorithm outputs a public-secret key-pair* (pk, sk)*.*

$\underline{tag \leftarrow \mathsf{TagGen}(sk)}$*: The* tag-generation *algorithm takes as input a secret key* sk*. It outputs a tag* tag*.*

$\underline{(tx, \sigma) \leftarrow \mathsf{Spend}(\mathcal{R}, \mathcal{I}, \mathcal{O}, \mu)}$*: The* spend *algorithm inputs the following:*

- $\mathcal{R} = \left\{ pk_i^{\mathcal{R}} \right\}_{i \in [|\mathcal{R}|]}$: *a set of keys* $pk_i^{\mathcal{R}}$ *with each key is associated with* c *coins.*
- $\mathcal{I} = (j, sk, tag)$*: a tuple consisting of an index* j*, a secret key* sk*, and a tag* tag *(of* $pk_j^{\mathcal{R}}$*)*
- $\mathcal{O} = \left\{ pk_i^{\mathcal{O}} \right\}_{i \in [|\mathcal{O}|]}$: *a set consisting of target keys* $pk_i^{\mathcal{O}}$ *to which input coins* c *are distributed to.*
- μ*: metadata*

It outputs a transaction tx *and a signature* σ *where* tx *defined as follows:* $tx := (\mathcal{R}, tag, \mathcal{O}, \mu)$*.*
$b \leftarrow \mathsf{Vf}(tx, \sigma)$*: The* verify *algorithm inputs a transaction* tx *and a signature* σ*. It outputs a bit* b *denoting the validity of* σ*.*

Security. We give intuitions for the security of LRS-TS and refer the reader to Appendix A for the formal definitions. We have three properties of LRS-TS, namely (1) *Privacy*: LRS-TS should ensure privacy of the source account, meaning an adversarial observer on the blockchain should not learn any information about the source address from a transaction other than the fact that it is a member of the ring of one-time addresses, (2) *Non-Slanderability (Unforgeability)*: LRS-TS must ensure that an adversarial user cannot steal the coins of an honest user (unforgeability) or spend coins on behalf of an honest user (non-slanderability), and (3) *Linkability*: LRS-TS must ensure that an adversary cannot double spend his coins. This means that if the adversary tries to spend from the same one-time key twice, then the two resulting transaction must be linkable and thereby detect the double spend attack.

The subtle difference between these two properties is that an adversary stealing funds has to produce a valid authentication token (a signature) for the honest user's one-time key. While an adversary spending on behalf of an honest user (or slandering an honest user) may use only the tag of the honest user's key in his transaction and not necessarily generate the authentication token for the honest user's key. This way of spending makes the honest user's coins unspendable: as the corresponding tag has been used by the adversary already which makes any attempt by the honest user to spend his funds (marked) as a double spend and therefore rejected by the blockchain.

We note that it is enough to prove that a LRS-TS satisfies our privacy, linkability and non-slanderability definitions, as our formal definitions of linkability (Theorem A.3) and non-slanderability (Theorem A.2) actually imply the notion of unforgeability.

4.2 LRS-TS Construction in Monero

We give a formal description of the LRS-TS scheme deployed in Monero, as it will be useful later for our construction. The scheme (Figure 2) is defined over a cyclic group $\mathbb{G}$ of prime order q with generator G and uses two different hash functions $\mathsf{H_P} : \mathbb{G} \to \mathbb{G}, \mathsf{H_S} : \{0,1\}^* \to \mathbb{Z}_q^*$.

The private-public key pair is the tuple $(x, G^x) \in \mathbb{Z}_q^* \times \mathbb{G}$. Each secret key is associated with a unique linkability tag that is set as $tag := \mathsf{H_P}(pk)^{sk}$. The spend algorithm takes as input a ring of public keys, the secret key and the tag of the public key that is a member of the ring. For ease of understanding we make the assumption that the spending public key is always $pk_{|\mathcal{R}|}$. The algorithm samples $(s'_0, s_1, \ldots, s_{|\mathcal{R}|-1}) \leftarrow \mathbb{Z}_q^*$ and computes L_0, R_0, h_0 and L_i, R_i, h_i for each index $i \in [|\mathcal{R}| - 1]$ (as shown in Figure 2). The algorithm finally sets $s_0 := s'_0 - h_{|\mathcal{R}|-1} \cdot sk$ and the signature consists of $\sigma := (s_0, s_1, \ldots, s_{|\mathcal{R}|-1}, h_0)$. Note that s_0 is reminiscent of how signing is done in Schnorr signatures. The verification algorithm runs the same loop as in the spend algorithm (except that it now ranges over the full ring) to obtain $h_{|\mathcal{R}|}$ and it accepts only if $h_0 = h_{|\mathcal{R}|}$.

Using the following theorems, we show that the construction shown in Figure 2 satisfies the security notions of a LRS-TS as defined in Section 4.1. The formal proofs of the theorems are deferred to Appendix A.4.

Theorem 4.2 (Privacy)**.** *If the Decisional Di e-Hellman problem (DDH) is hard over the group* $\mathbb{G}$ *(Definition C.2) then the LRS transaction scheme used in Monero is* private *Theorem A.1 in the ROM.*

Theorem 4.3 (Non-Slanderability)**.** *If the Discrete Logarithm problem (DL) is hard over the group* $\mathbb{G}$ *(Definition C.1), then the LRS transaction scheme used in Monero is* non-slanderable *Theorem A.2 in the ROM.*

Theorem 4.4 (Linkability)**.** *The LRS construction used in Monero is* linkable *Theorem A.3 in the ROM.*

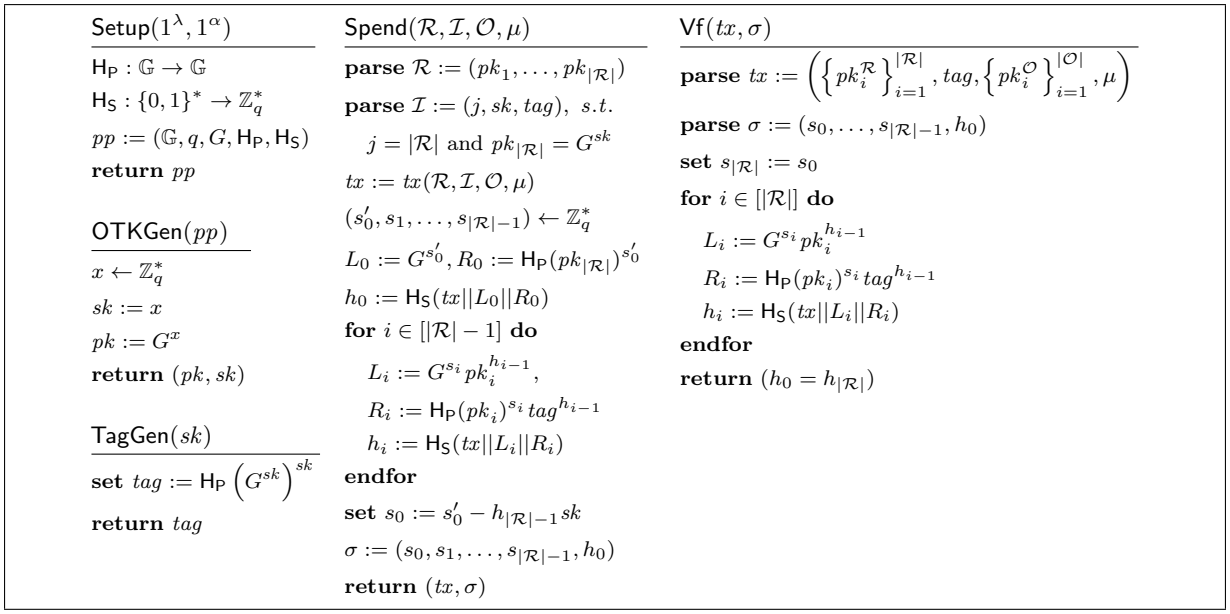

Figure 2: LRS transaction scheme Π_{LRS} used in Monero. Here the spending key G^{sk} is assumed to be the $|\mathcal{R}|$-th element of the ring $\mathcal{R}$.

5 Verifiable Timed Linkable Ring Signature

In the following we defined and construct a *Verifiable Timed Linkable Ring Signature* (VTLRS) transaction scheme.

5.1 Definition

A VTLRS is a linkable ring signature based transaction scheme with the additional property that one can commit to such a signature in a *verifiable* and *extractable* way.

Definition 5.1 (VTLRS)**.** *A Verifiable Timed Linkable Ring Signature Transaction Scheme,* Π_{VTLRS} *for a linkable ring signature transaction scheme* Π_{LRS} *is a tuple of five algorithms* (Setup, Commit, Verify, Open, ForceOp) *where:*

$crs \leftarrow \mathsf{Setup}(1^\lambda)$*: the setup algorithm outputs a common reference string* crs *which is implicitly taken as input in all other algorithms.*

$(C, \pi) \leftarrow \mathsf{Commit}(\sigma, tx, \mathbf{T}; r)$*: the commit algorithm takes as input a signature* σ*, the transaction* tx*, a hiding time* $\mathbf{T}$ *and randomness* r*. It outputs a commitment* C *and a proof* π*.*

$0/1 \leftarrow \mathsf{Verify}(tx, C, \pi)$*: the verify algorithm takes as input a transaction* m*, a commitment* C *of hardness* $\mathbf{T}$ *and a proof* π *and accepts the proof if and only if, the value* σ *embedded in* c *is a valid linkable ring signature on the transaction* tx *(i.e.,* $\mathsf{LRS.Vf}(tx, \sigma) = 1$*). Else it outputs* 0*.*

$(\sigma, r) \leftarrow \mathsf{Open}(C; r)$*: the opening algorithm is run by the committer that as input a commitment* C *and outputs the committed signature* σ *and the randomness* r *used in generating the commitment* C*.*

$\sigma \leftarrow \mathsf{ForceOp}(C)$*: the deterministic* ForceOp *algorithm takes as input the commitment* C *and outputs a signature* σ*.*

The correctness requirement of a VTLRS transaction scheme is formalized in the definitions below.

Definition 5.2 (Correctness of VTLRS)**.** *A Verifiable Timed Linkable Ring Signature Transaction Scheme,* $\Pi_{\mathsf{VTLRS}} := (\mathsf{Setup}, \mathsf{Commit}, \mathsf{Verify}, \mathsf{Open}, \mathsf{ForceOp})$ *construction for a linkable ring signature transaction scheme* $\Pi_{\mathsf{LRS}} := (\mathsf{LRS.Setup}, \mathsf{LRS.OTKGen}, \mathsf{LRS.TagGen}, \mathsf{LRS.Spend}, \mathsf{LRS.Vf})$ *is said to satisfy* correctness*, if (i) for all* $\lambda \in \mathbb{N}$*, (ii) all* crs *output by* $\mathsf{Setup}(1^\lambda)$*, (iii) all transactions* tx*, all hiding times* $\mathbf{T} \in \mathbb{N}$*, and (iv) all signature* σ *output by* LRS.Spend*, the following conditions hold*

1. $\mathsf{Verify}(tx, \mathsf{Commit}(\sigma, tx, \mathbf{T})) = 1$,
2. *for all randomness* r *such that* $(C, \pi) \leftarrow \mathsf{Commit}(\sigma, tx, \mathbf{T}; r)$ *we have* $(\sigma, r) \leftarrow \mathsf{Open}(C; r)$, *and* $\sigma \leftarrow \mathsf{ForceOp}(C)$.

Beyond privacy, non-slanderability, and linkability (defined as for LRS), a VTLRS must satisfy the notions of *timed privacy* and *soundness*, defined below.

Timed Privacy. The notion of *timed privacy* requires that all PRAM algorithms whose running time is at most t (where $t < \mathbf{T}$), succeed in extracting σ from the commitment C and π with at most negligible probability. Notice that the adversary is given the spending public key and the tag as input, and gets access to a spending oracle. The challenge for the adversary here is to distinguish (within time $\mathbf{T}$ even with parallel computation power) a commitment from being a commitment to a valid LRS signature with the above attributes, to a simulated commitment.

Definition 5.3 (Timed Privacy)**.** *A Verifiable Timed Linked Ring Signature scheme* $\Pi_{\mathsf{VTLRS}} = (\mathsf{Setup}, \mathsf{Commit}, \mathsf{Verify}, \mathsf{Open}, \mathsf{ForceOp})$ *for a LRS transaction scheme* $\Pi_{\mathsf{LRS}} = (\mathsf{LRS.Setup}, \mathsf{LRS.OTKGen}, \mathsf{LRS.TagGen}, \mathsf{LRS.Spend}, \mathsf{LRS.Vf})$ *is* verifiable timed private *if there exists a PPT simulator* $\mathcal{S}$*, a negligible function* $\mathsf{negl}(\lambda)$*, and a polynomial* $\tilde{\mathbf{T}}$ *such that for all polynomials* $\mathbf{T} > \tilde{\mathbf{T}}$*, all algorithms* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ *where* $\mathcal{A}_1$ *is PPT and* $\mathcal{A}_2$ *is a PRAM whose running time is at most* $t < \mathbf{T}$*, and all* $\lambda \in \mathbb{N}$ *it holds that*

$$\Pr\left[b = b' \;\middle|\; \begin{array}{c} (pk, sk) \leftarrow \mathsf{LRS.OTKGen}(pp) \\ tag \leftarrow \mathsf{TagGen}(sk) \\ (\mathcal{R}, \mathcal{O}, \mu) \leftarrow \mathcal{A}_1^{\mathsf{Spend}\mathcal{O}}(pk, tag, pp) \\ s.t.\ pk = pk_{|\mathcal{R}|} \text{ and } tx := \{\mathcal{R}, tag, \mathcal{O}, \mu\} \\ b \leftarrow \{0,1\},\ b' \leftarrow \mathcal{A}_2^{\mathsf{Spend}\mathcal{O}}(tx, C_b, \pi_b) \end{array} \right] \leq \mathsf{negl}(\lambda)$$

where, $pp \leftarrow \mathsf{LRS.Setup}(1^\lambda), crs \leftarrow \mathsf{Setup}(1^\lambda)$ *and if* $b = 0$*, then* $(C_0, \pi_0) \leftarrow \mathsf{Commit}(\sigma, tx, \mathbf{T})$ *where* $\sigma \leftarrow \mathsf{LRS.Spend}(\mathcal{R}, (|\mathcal{R}|, sk, tag), \mathcal{O}, \mu)$ *and if* $b = 1$*,* $(C_1, \pi_1) \leftarrow \mathcal{S}(pk, tx, \mathbf{T})$.

Soundness. The notion of soundness says that the accepting verifier is convinced that, given C, the $\mathsf{ForceOp}$ algorithm will return a valid signature σ on transaction tx in time $\mathbf{T}$. We say that a VTLRS is *simulation-sound* if it is sound even when the prover has access to simulated proofs for (possibly false) statements of his choice; i.e., the prover must not be able to compute a valid proof for a fresh false statement of his choice.

Definition 5.4 (Soundness)**.** *A Verifiable Timed Linked Ring Signature scheme* $\Pi_{\mathsf{VTLRS}} = (\mathsf{Setup}, \mathsf{Commit}, \mathsf{Verify}, \mathsf{Open}, \mathsf{ForceOp})$ *for a LRS transaction scheme* $\Pi_{\mathsf{LRS}} = (\mathsf{LRS.Setup}, \mathsf{LRS.OTKGen}, \mathsf{LRS.TagGen}, \mathsf{LRS.Spend}, \mathsf{LRS.Vf})$ *is* sound *if there is a negligible function* $\mathsf{negl}(\lambda)$ *such that for all PPT adversaries* $\mathcal{A}$ *and all* $\lambda \in \mathbb{N}$*, we have:*

$$\Pr\left[b_1 = 1 \wedge b_2 = 0 \;\middle|\; \begin{array}{c} crs \leftarrow \mathsf{Setup}(1^\lambda) \\ (tx, C, \pi, \mathbf{T}) \leftarrow \mathcal{A}(crs) \\ (\sigma, r) \leftarrow \mathsf{ForceOp}(C) \\ b_1 := \mathsf{Verify}(tx, C, \pi) \\ b_2 := \mathsf{LRS.Vf}(tx, \sigma) \end{array} \right] \leq \mathsf{negl}(\lambda)\,.$$

5.2 Our VTLRS Construction

We give a construction of VTLRS transaction scheme for the LRS transaction scheme used in Monero. Throughout the following overview, we describe the VTLRS as an interactive protocol between a committer and a verifier, which can be made non-interactive using the Fiat-Shamir transformation [39]. A formal description of our VTLRS is given in Figure 3, where hash function $\mathsf{H}' : \{0,1\}^* \to J$, with J being a set

of indices in $[n]$ such that $|J| = t-1$, is used used to implement the Fiat-Shamir transformation. We now give an intuitive description of the VTLRS construction.

High-Level Overview. The commit algorithm proceeds as follows: Consider a signature $\sigma := (s_0, s_1, \ldots, s_{|\mathcal{R}|-1}, h_0)$ generated by Spend algorithm of Figure 2 on a transaction

$$tx := \left(\left\{ pk_i^{\mathcal{R}} \right\}_{i=1}^{|\mathcal{R}|}, tag, \left\{ pk_i^{\mathcal{O}} \right\}_{i=1}^{|\mathcal{O}|}, \mu \right).$$

Let $pk_{|\mathcal{R}|}$ be the spending key and the committer is privy to this knowledge (which we justify below). The commit algorithm takes as input this transaction tx, signature σ and the hiding time $\mathbf{T}$. To generate a VTLRS on transaction tx, the committer secret shares the values

$$\mathsf{sc} := (s_0, G^{s_0}, \mathsf{H_P}(pk_{|\mathcal{R}|})^{s_0})$$

using a t-out-of-n threshold sharing scheme in the following way:

1. For the first $t-1$ shares, choose $\alpha_i \in \mathbb{Z}_q$ uniformly at random and set $K_i := G^{\alpha_i}$ and $Y_i := \mathsf{H_P}(pk_{|\mathcal{R}|})^{\alpha_i}$, respectively. Note that each share α_i can be publicly verified for consistency by recomputing K_i and Y_i.

2. For the remaining $n-(t-1)$ shares, use Lagrange interpolation in the exponent, i.e., for $i \in \{t, t+1, \ldots, n\}$ set

$$\alpha_i = \left(s_0 - \sum_{j \in [t-1]} \alpha_j^{\ell_j(0)} \right)^{\ell_i(0)^{-1}}$$

$$K_i = \left(\frac{G^{s_0}}{\prod_{j \in [t-1]} K_j^{\ell_j(0)}} \right)^{\ell_i(0)^{-1}} \quad Y_i = \left(\frac{\mathsf{H_P}(pk_{|\mathcal{R}|})^{s_0}}{\prod_{j \in [t-1]} Y_j^{\ell_j(0)}} \right)^{\ell_i(0)^{-1}}$$

   where $\ell_i(\cdot)$ is the i-th Lagrange polynomial basis. Note that here α_i's are integers in $\mathbb{Z}_q$ while $K_i, Y_i \in \mathbb{G}$.

The above steps ensure that we can reconstruct (via Lagrange interpolation) the valid s_0 value that is part of the signature σ from *any* t-sized set of shares of sc.

The committer then computes a time-lock puzzle Z_i with time parameter $\mathbf{T}$ for each share α_i separately. The first message consists of all puzzles $(Z_1, \ldots, Z_n)$ together with $G^{s_0}, \mathsf{H_P}(pk_{|\mathcal{R}|})^{s_0}$ and all (K_i, Y_i) as defined above.

Consistency Proof via Cut-and-Choose. After receiving the above first message, the verifier chooses a random set I of size $(t-1)$ as the challenge set. For this set, the committer opens the time-lock puzzles $\{Z_i\}_{i \in I}$ and reveals the underlying value α_i (together with the corresponding random coins) that it committed to. The verifier wants to ensure that, (i) the puzzles are indeed generated for the correct timing hardness $\mathbf{T}$ and can be successfully solved in that time and (ii) as long as at least one of the shares in the *unopened* puzzles $(\{Z_i\}_{i \in [n]/I})$ is consistent with respect to the corresponding partial commitments (K_i, Y_i), then we can use it to reconstruct s_0 and therefore a valid σ. To do this, the verifier performs the following checks and accepts the commitment as legitimate only if they are all successful:

1. All puzzles $\{Z_i\}_{i \in I}$ are correctly generated using α_i and the corresponding randomness (which was also revealed above) with timing hardness $\mathbf{T}$

2. All $\{\alpha_i\}_{i \in I}$ are consistent with the corresponding K_i, Y_i, i.e., $K_i = G^{\alpha_i}, Y_i = \mathsf{H_P}(pk_{|\mathcal{R}|})^{\alpha_i}$.

3. All K_i, Y_i are valid shares of G^{s_0} and $\mathsf{H_P}(pk_{|\mathcal{R}|})^{s_0}$ respectively, i.e., $K_i^{\ell_i(0)} \cdot \prod_{j \in I} K_j^{\ell_j(0)} = G^{s_0}$ and $Y_i^{\ell_i(0)} \cdot \prod_{j \in I} Y_j^{\ell_j(0)} = \mathsf{H_P}(pk_{|\mathcal{R}|})^{s_0}$.

Consequently, to fool a verifier, a malicious prover has to guess the challenge set I ahead of time to pass the above checks without actually committing a valid s_0 (signature σ). Setting the parameters appropriately, we can guarantee that this happens only with negligible probability.

Signature Recovery via Homomorphic Packing. To recover s_0 and the valid signature, the verifier has to solve $\tilde{n} = (n - t + 1)$ puzzles to force the opening of a VTLRS. To close the gap between honest and malicious verifiers, we would like to reduce his workload to the minimal one of solving a single puzzle. To achieve this goal, we use the linearly homomorphic time-lock puzzle construction [40], combined with standard packing techniques to compress $\tilde{n}$ puzzles into a single one. Concretely, the verifier, on input $(Z_1, \ldots, Z_{\tilde{n}})$ homomorphically evaluates the linear function

$$f(x_1, \ldots, x_{\tilde{n}}) = \sum_{i=1}^{\tilde{n}} 2^{(i-1)\cdot\lambda} \cdot x_i \tag{1}$$

to obtain a single puzzle $\tilde{Z}$, which he can solve in time $\mathbf{T}$. Observe that, once the puzzle is solved, all signatures can be decoded from the bit-representations of the resulting message. However we need to ensure that:

1. The message space of the homomorphic time-lock puzzle must be large enough to accommodate all $\tilde{n}$ signatures.

2. The values α_i encoded in the the input puzzles must not exceed the maximum size of a signature (say λ bits).

Condition (1) can be satisfied instantiating the linearly homomorphic time-lock puzzles with a large enough message space. On the other hand, condition (2) is enforced by including a range NIZK $(\mathcal{P}_{\mathsf{NIZK},\mathcal{L}_{\mathsf{rng}}}, \mathcal{V}_{\mathsf{NIZK},\mathcal{L}_{\mathsf{rng}}})$ for the language $\mathcal{L}_{\mathsf{rng}}$, which certifies that the message of each time-lock puzzles falls into the range $[0, 2^\lambda]$. More formally,

$$\mathcal{L}_{\mathsf{rng}} := \left\{ \begin{array}{l} stmt = (Z, 0, 2^\lambda, \mathbf{T}) : \exists wit = (\alpha, r) \quad s.t. \\ (Z \leftarrow \mathsf{LHTLP.PGen}(pp, \alpha; r)) \ \wedge (\alpha \in [0, 2^\lambda]) \end{array} \right\}.$$

We instantiate the range proof with the recently introduced protocol [30] which we recall for completeness in Appendix E.

Security Analysis. In the following theorems we argue the security of our VTLRS construction. The formal proofs of the theorems are deferred to Appendix D.

Theorem 5.5 (Timed Privacy)**.** *Let* $(\mathsf{Setup}_{\mathsf{NIZK},\mathcal{L}_{\mathsf{rng}}}, \mathcal{P}_{\mathsf{NIZK},\mathcal{L}_{\mathsf{rng}}}, \mathcal{V}_{\mathsf{NIZK},\mathcal{L}_{\mathsf{rng}}})$ *be a NIZK for* $\mathcal{L}_{\mathsf{rng}}$ *and let* LHTLP *be a secure time-lock puzzle. Then the protocol as described in Figure 3 satisfies verifiable timed privacy as in Theorem 5.3 in the ROM.*

Theorem 5.6 (Soundness)**.** *Let* $(\mathsf{Setup}_{\mathsf{NIZK},\mathcal{L}_{\mathsf{rng}}}, \mathcal{P}_{\mathsf{NIZK},\mathcal{L}_{\mathsf{rng}}}, \mathcal{V}_{\mathsf{NIZK},\mathcal{L}_{\mathsf{rng}}})$ *be a NIZK for* $\mathcal{L}_{\mathsf{rng}}$ *and let* LHTLP *be a time-lock puzzle with perfect correctness. Then the protocol as described in Figure 3 satisfies soundness as in Theorem 5.4 in the ROM.*

Knowing the Spending Key. In our VTLRS construction, the committer knows the spending key $pk_{|\mathcal{R}|}$ in the ring $\mathcal{R}$ that was used to generate σ on transaction tx. This is justified because, in our applications (payment channel and atomic swaps), the committer and the verifier jointly generate a signature σ, on the joint public key $pk_{|\mathcal{R}|}$ in such a way that only the committer learns the valid signature. The committer then generates a VTLRS commitment (C, π) over σ and gives it to the verifier. The committer therefore is aware of the spending key which here is the joint key $pk_{|\mathcal{R}|}$.

5.3 Optimizations

In the following we discuss how to deal with the presence of a trusted setup and further optimize the computation needed for solving puzzles.

On The Setup Assumption. Our VTLRS protocol requires a one-time setup that is computed by a trusted party. The output of this setup procedure consists of the common reference string crs_{rng} for the range proof and the public parameters pp of the homomorphic time-lock puzzles. Specifically, crs_{rng} consists of sampling a random oracle and pp is a (uniformly sampled) RSA modulus $N = p \cdot q$. For

$\mathsf{Setup}(1^\lambda)$

$crs_{\mathsf{rng}} \leftarrow \mathsf{ZK.Setup}(1^\lambda)$

$pp_{\mathsf{LRS}} \leftarrow \mathsf{LRS.Setup}(1^\lambda, 1^\alpha)$

$pp_{\mathsf{LHTLP}} \leftarrow \mathsf{LHTLP.Setup}(1^\lambda, \mathbf{T})$

$crs := (crs_{\mathsf{rng}}, pp_{\mathsf{LRS}}, pp_{\mathsf{LHTLP}})$

return crs

$\mathsf{Commit}(\sigma, tx, \mathbf{T})$

parse $crs := (crs_{\mathsf{rng}}, pp_{\mathsf{LRS}}, pp_{\mathsf{LHTLP}})$

parse $tx := \left(\mathcal{R}, tag, \left\{pk_i^{\mathcal{O}}\right\}_{i=1}^{|\mathcal{O}|}, \mu\right)$

parse $\sigma := (s_0, s_1, \ldots, s_{|\mathcal{R}|-1}, h_0)$

parse $\mathcal{R} := (pk_1, \ldots, pk_{|\mathcal{R}|})$

$\forall i \in [t-1]\ \alpha_i \leftarrow \mathbb{Z}_q^*, K_i := G^{\alpha_i}, Y_i := \mathsf{H_P}(pk_{|\mathcal{R}|})^{\alpha_i}$

for $i \in \{t, \ldots, n\}$ **do**

$$\alpha_i = \left(s_0 - \sum_{j \in [t-1]} \alpha_j \cdot \ell_j(0)\right) \cdot \ell_i(0)^{-1}$$

$$K_i = \left(\frac{G^{s_0}}{\prod_{j\in[t-1]} K_j^{\ell_j(0)}}\right)^{\ell_i(0)^{-1}}$$

$$Y_i = \left(\frac{\mathsf{H_P}(pk_{|\mathcal{R}|})^{s_0}}{\prod_{j\in[t-1]} Y_j^{\ell_j(0)}}\right)^{\ell_i(0)^{-1}}$$

endfor

for $i \in [n]$ **do**

$r_i \leftarrow \{0,1\}^\lambda$

$Z_i \leftarrow \mathsf{LHTLP.PGen}(pp, \alpha_i; r_i)$

$\pi_{\mathsf{rng},i} \leftarrow \mathcal{P}_{\mathsf{NIZK},\mathcal{L}_{\mathsf{rng}}}(crs_{\mathsf{rng}}, (Z_i, 0, 2^\lambda, \mathbf{T}), (\alpha_i, r_i))$

endfor

$I \leftarrow \mathsf{H}'\left(G^{s_0}, \mathsf{H_P}(pk_{|\mathcal{R}|})^{s_0}, \{(K_i, Y_i, Z_i, \pi_{\mathsf{rng},i})\}_{i\in[n]}\right)$

set $C := (G^{s_0}, \mathsf{H_P}(pk_{|\mathcal{R}|})^{s_0}, \{s_i\}_{i\in[|\mathcal{R}|-1]}, h_0, \{Z_i\}_{i\in[n]}, \mathbf{T})$

set $\pi := (\{K_i, Y_i, \pi_{\mathsf{rng},i}\}_{i\in[n]}, I, \{\alpha_i, r_i\}_{i\in I})$

return (C, π)

$\mathsf{Open}(C, \{r_i\}_{i\in[n]})$

return $(\sigma, \{r_i\}_{i\in[n]})$

$\mathsf{Verify}(tx, C, \pi)$

parse $tx := \left(\left\{pk_i^{\mathcal{R}}\right\}_{i=1}^{|\mathcal{R}|}, tag, \left\{pk_i^{\mathcal{O}}\right\}_{i=1}^{|\mathcal{O}|}, \mu\right)$

parse $crs := (crs_{\mathsf{rng}}, pp_{\mathsf{LRS}}, pp_{\mathsf{LHTLP}})$

parse $C := (\tilde{G}, \tilde{H}, \{s_i\}_{i\in[|\mathcal{R}|-1]}, h_0, \{Z_i\}_{i\in[n]}, \mathbf{T})$

parse $\pi := (\{K_i, Y_i, \pi_{\mathsf{rng},i}\}_{i\in[n]}, I, \{\alpha_i, r_i\}_{i\in I})$

for $i \in [|\mathcal{R}| - 1]$ **do**

$L_i := G^{s_i} pk_i^{h_{i-1}}$

$R_i := \mathsf{H_P}(pk_i)^{s_i} tag^{h_{i-1}}$

$h_i := \mathsf{H_S}(tx||L_i||R_i)$

endfor

$L_{|\mathcal{R}|} := \tilde{G} \cdot pk_{|\mathcal{R}|}^{h_{|\mathcal{R}|-1}}$

$R_{|\mathcal{R}|} := \tilde{H} \cdot tag^{h_{|\mathcal{R}|-1}}$

$h_{|\mathcal{R}|} := \mathsf{H_S}(tx||L_{|\mathcal{R}|}||R_{|\mathcal{R}|})$

$b_1 := (h_0 \neq h_{|\mathcal{R}|})$

$$b_2 := \exists j \notin I \left(K_j^{\ell_j(0)} \cdot \prod_{i\in I} K_i^{\ell_i(0)} \neq \tilde{G}\right)$$

$$b_3 := \exists j \notin I \left(Y_j^{\ell_j(0)} \cdot \prod_{i\in I} Y_i^{\ell_i(0)} \neq \tilde{H}\right)$$

$b_4 := \exists i \in [n] \left(\mathcal{V}_{\mathsf{NIZK},\mathcal{L}_{\mathsf{rng}}}(crs_{\mathsf{rng}}, (Z_i, 0, 2^\lambda, \mathbf{T}), \pi_{\mathsf{rng},i}) \neq 1\right)$

$b_5 := \exists i \in I\,(Z_i \neq \mathsf{LHTLP.PGen}(pp, \alpha_i; r_i))$

$b_6 := \exists i \in I(K_i \neq G^{\alpha_i})$

$b_7 := \exists i \in I \left(Y_i \neq \mathsf{H_P}(pk_{|\mathcal{R}|})^{\alpha_i}\right)$

$b_8 := \left(I \neq \mathsf{H}'\left(\tilde{G}, \tilde{H}, \{(K_i, Y_i, Z_i, \pi_{\mathsf{rng},i})\}_{i\in[n]}\right)\right)$

if $\bigvee_{i\in[8]} b_i = 1$ **then**

return 0

else

return 1

$\mathsf{ForceOp}(C)$

parse $C := (\tilde{G}, \tilde{H}, \{s_i\}_{i\in[|\mathcal{R}|-1]}, h_0, \{Z_i\}_{i\in[n]}, \mathbf{T})$

$\forall_{i\in[n]} \alpha_i \leftarrow \mathsf{LHTLP.PSolve}(pp, Z_i)$

$s_0 := \sum_{j\in[t]} (\alpha_j) \cdot \ell_j(0)$ // assuming first t shares are valid

$\sigma := (s_0, \ldots, s_{|\mathcal{R}|-1}, h_0)$

return σ

Figure 3: Verifiable Timed Linkable Ring Signature-Transaction Scheme

our applications, the VTLRS is generated by one user and given to one other user (verifier/solver), and therefore it suffices that the solver does not learn the factorization of N. This implies that the VTLRS commitment generator can sample N himself and make it part of his public key. That is, our VTLRS can be implemented *without a trusted setup* (in the ROM). However, there are some advantages in assuming a global modulus N (with unknown factors) which is shared across all users, as discussed below.

Batched Force-Opening of VTLRS Commitments. Looking ahead to our PC application, the VTLRS will substitute the time-lock functionality of a blockchain, i.e. the channel can be closed once

the signature is recovered. This however means that a user has to continuously solve as many puzzles as currently open channels. This might limit the number of channels that one can keep open at the same time. To mitigate this issue, we observe that assuming (i) a large enough message space of the time-lock puzzles and (ii) global public parameters pp, one can batch the solution of different puzzle into a single one. This can be done by homomorphically packing the messages in each puzzle into a single puzzle with a linear function (see Equation (1)) as discussed before. This is efficiently implementable using known constructions [40]. Under these assumptions, each user will have to solve *at most* a single puzzles at all times, regardless of how many channels are open.

6 PayMo: Payment Channels For Monero

In the following we describe our protocol for (uni-directional) payment channels in Monero.

6.1 Payment Channel Formalism

We formally define the notion of a payment channel as an ideal functionality $\mathcal{F}_{\mathsf{PC}}$ in Figure 4. Our model closely follows (a simplified version of) the functionality proposed in [4].

Payment channels in the Blockchain $\mathbb{B}$ are of the form $(c_{\langle u_0,u_1\rangle}, v, t)$, where $c_{\langle u_0,u_1\rangle}$ is a unique channel identifier for the channel between users u_0 and u_1, v is the capacity of the channel, and t is the expiration time of the channel. Note that any two users may have multiple channels open simultaneously. The functionality maintains two additional internal lists $\mathcal{C}$ and $\mathcal{P}$. The former is used to keep track of the closed channels, while the latter records the the off-chain payments in a open channel. Entries in $\mathcal{P}$ are of the form $(c_{\langle u_0,u_1\rangle}, v, t, h)$, where $c_{\langle u_0,u_1\rangle}$ is the corresponding identifier, v is the amount of credit used, t is the expiration time of the channel, and h is the identifier for this entry.

The functionality has access to the current time via the global clock. It provides the users with interfaces `open`, `close` and `pay` using which the user can open a channel, close the channel and make payments using the channel, respectively. $\mathcal{F}_{\mathsf{PC}}$ initializes a pair of empty lists $\mathcal{P}, \mathcal{C}$. Users can query $\mathcal{F}_{\mathsf{PC}}$ to open channels and close them to any valid state in $\mathcal{P}$. On input a value μ and a channel $c_{\langle u_0,u_1\rangle}$ from some user u_0, $\mathcal{F}_{\mathsf{PC}}$ checks whether the channel is an open one and if the value μ is less than the channel capacity. If so, the functionality updates the channel capacity and adds the new state d_i (after the i-th payment) to $\mathcal{P}$.

To see why the ideal functionality captures the security notion of interest of a PC, observe that a payment in the channel or it can be closed only by one of the users involved in the channel. If the time exceeds the channel expiry time and there is a close channel request, the functionality ignores all the payments and everything returns to the state as it were before any payments. Notice that payment amounts cannot exceed the channel's latest capacity, thereby ensuring users cannot payments with outdated state of balance or even mint new coins out of thin air.

6.2 Auxiliary Interfaces

To ease the presentation of our main PC protocol, we introduce two interfaces that allow a pair of users to generate a joint key and to jointly sign a transaction using the LRS of Monero. We define the corresponding ideal functionality $\mathcal{F}_{\mathsf{J-LRS}}$ in Figure 5. Our PC protocol will then be described in a hybrid world where parties have oracle access to the these ideal functionalities.

More concretely, the functionality $\mathcal{F}_{\mathsf{J-LRS}}$ consists of two interfaces: (i) The KTGen interface enables two users to generate joint one-time public keys and corresponding tags of the LRS transaction scheme in such a way that neither user learns the corresponding secret key of the one-time joint public key. (ii) The JSpend interface enables two users to spend from a joint one-time key by generating a signature on a transaction of the users' choice and letting users choose the $\{s_i\}_{i\in[|\mathcal{R}|-1]}$ values of the signature (refer to Figure 2). The interface returns the valid signature to one of the users.

Instantiations. Due to space constraints, we only give a brief overview of the protocols that we design to realize these ideal interfaces and we refer the reader to Appendix F for formal descriptions and for the security analysis.

Open Channel: On input $(\texttt{open}, c_{\langle u,u'\rangle}, v, u', t)$ from a user u, do the following:

- Check whether $c_{\langle u,u'\rangle}$ is well-formed (contains valid identifiers and it is not a duplicate).
- Send $(c_{\langle u,u'\rangle}, v, t)$ to u', who can either abort or authorize the operation.
- In the latter case, append the tuple $(c_{\langle u,u'\rangle}, v, t)$ to $\mathbb{B}$ and the tuple $(c_{\langle u,u'\rangle}, v, t, h)$ to $\mathcal{P}$, for some random h.
- Return h to u and u'.

Close Channel: On input $(\texttt{close}, c_{\langle u,u'\rangle}, h)$ from either user u or u', do the following:

- Parse $\mathbb{B}$ for an entry $(c_{\langle u,u'\rangle}, v, t)$ and $\mathcal{P}$ for an entry $(c_{\langle u,u'\rangle}, v', t, h)$, for $h \neq \perp$.
- If $c_{\langle u,u'\rangle} \in \mathcal{C}$ and the current time T (according to the global clock) is greater than t, abort.
- Otherwise add $(c_{\langle u,u'\rangle}, u', v', t)$ to $\mathbb{B}$ and add $c_{\langle u,u'\rangle}$ to $\mathcal{C}$.
- Notify the other user with the message $(c_{\langle u,u'\rangle}, \perp, h)$.

Pay: On input $(\texttt{pay}, \mu, c_{\langle u_0,u_1\rangle}, t_0)$ from a user u_0, do the following:

- Sample a random h.
- Parse $\mathbb{B}$ for an entry of the form $(c_{\langle u_0,u_1\rangle}, v, t)$.
- If such an entry does exist, send the tuple $(h, c_{\langle u_0,u_1\rangle}, \mu)$ to the users u_1.
- Check whether for all entries of the form $(c, v', \cdot, \cdot) \in \mathcal{P}$ it holds that $v' \geq \mu$ and that $t_0 > T$ (where T is the current time).
- If this is the case add $d_i = (c_{\langle u_0,u_1\rangle}, v' - \mu, t_0, \perp)$ to $\mathcal{P}$, where $(c_{\langle u_0,u_1\rangle}, v', \cdot, \cdot) \in \mathcal{P}$ is the entry with the lowest v'.
- If any of the conditions above is not met, remove from $\mathcal{P}$ all the entries d_i added in this phase and abort.

Figure 4: Ideal functionality $\mathcal{F}_{\mathsf{PC}}$ for Payment Channels (PC).

$\mathsf{KTGen}(\mathbb{G}, G, q)$

Upon invocation by both users U_0 and U_1:
sample $x \leftarrow \mathbb{Z}_q$ and compute $pk := G^x$
set $sk_{U_0U_1} := x$
Sample hash functions $\mathsf{H_S} : \{0,1\}^* \to \mathbb{Z}_q^*$
and $\mathsf{H_P} : \mathbb{G} \to \mathbb{G}$
set $tag_{U_0U_1} \leftarrow \mathbb{G}$
choose random x_0, x_1 such that $x_0 + x_1 = x$
record $(pk, tag, sk_{U_0U_1}, x_0, x_1)$
send $x_0, pk, tag, \mathsf{H_S}, \mathsf{H_P}$ to U_0
send $x_1, pk, tag, \mathsf{H_S}, \mathsf{H_P}$ to U_1
ignore future calls from (U_0, U_1)

$\mathsf{JSpend}(U_b, (s_1, \ldots, s_{|\mathcal{R}|-1}), U_0, U_1, tx)$

Upon invocation by both users U_0 and U_1:
where U_b $(b \in \{0,1\})$ gives inputs $(s_1, \ldots, s_{|\mathcal{R}|-1})$
Retrieve $pk_{U_0U_1}$ that was generated
parse $tx := (\mathcal{R}, tag_{U_0,U_1}, pk^{\mathcal{O}}, \mu)$
parse $\mathcal{R} := (pk_1, \ldots, pk_{|\mathcal{R}|}),\ s.t.\ pk_{|\mathcal{R}|} := pk_{U_0U_1}$
choose $s_0' \leftarrow \mathbb{Z}_q^*$
Compute
$L_0 := G^{s_0'}, R_0 := \mathsf{H_P}(pk_{|\mathcal{R}|})^{s_0'}$
$h_0 := \mathsf{H_S}(tx||L_0||R_0)$
for $i \in [|\mathcal{R}| - 1]$ **do**
$L_i := G^{s_i} pk_i^{h_{i-1}}$,
$R_i := \mathsf{H_P}(pk_i)^{s_i} tag^{h_{i-1}}$
$h_i := \mathsf{H_S}(tx||L_i||R_i)$
endfor
set $s_0 := s_0' - h_{|\mathcal{R}|-1} sk_{U_0U_1}$
$\sigma := (s_0, s_1, \ldots, s_{|\mathcal{R}|-1}, h_0)$
return σ to U_b

Figure 5: Ideal functionality $\mathcal{F}_{\mathsf{J-LRS}}$

The joint key and tag generation (Figure 16) is standard extension of the protocol in [51]. The interaction is between Alice and Bob, where Alice and Bob generate the joint public key pk_{AB} similar to a Diffie-Hellman key exchange. The parties then jointly generate the joint tag tag_{AB} for pk_{AB}, with the exception that now the parties have to additionally prove in zero-knowledge that their messages are

consistent with the key exchange protocol. At the end of the protocol Alice and Bob learn x_A and x_B, respectively, which are the shares of the joint secret key $sk_{AB} := x_A + x_B$. And both Alice and Bob obtain the joint key $pk_{AB} := G^{x_A+x_B}$, the joint tag $tag_{AB} := \mathsf{H_P}(pk_{AB})^{x_A+x_B}$.

In the joint spending protocol (Figure 17), Alice and Bob generate a transaction tx that they wish to sign on. The transaction contains tag_{AB} as the spending tag and the ring $\mathcal{R}$ in the transaction consists of the joint one-time key pk_{AB}. Parties exchange messages in a consistent manner and run the spending algorithm from Figure 2 in a joint manner. The interaction is scheduled such that Alice sends the last message to Bob with which Bob can obtain a valid signature σ on tx with pk_{AB} as the spending key and tag_{AB} as the spending tag. Note that Alice does not obtain σ at the end of the interaction, which is crucial for our PC protocol.

6.3 PayMo Protocol

With these tools we describe our main protocol. Our construction consists of three phases: Channel opening, payment, and channel closing. We present a formal description of PayMo in Figure 6. Throughout the following overview, we consider an example of a uni-directional channel from Alice to Bob, where only Alice makes payments to Bob via the channel.

Channel Opening. The first step is for Alice and Bob to open a channel. This is done by generating a joint address and tag via the KTGen interface of $\mathcal{F}_{\mathsf{J-LRS}}$, which returns pk_{AB} and tag_{AB} to both parties. Alice and Bob then call the JSpend interface of $\mathcal{F}_{\mathsf{J-LRS}}$ on a transaction tx_{rdm} (that spends the coins from pk_{AB} to some address of Alice). The functionality returns the signature σ to Bob. Bob then generates a VTLRS on a transaction tx_{rdm} on σ and hands it over to Alice. The VTLRS is generated with timing hardness parameter $\mathbf{T}$, meaning that Alice learns the valid signature σ on tx_{rdm} only after time corresponding to the timing parameter $\mathbf{T}$. This means that after time $\mathbf{T}$, Alice will learn a the signature σ and therefore will be able to redeem any remaining coin in the address (if any). The channel is considered opened after Alice's last step, which consists in posting a transaction tx' that sends v coins to pk_{AB}. It is crucial for security that Alice performs this step only after she obtains a valid VTLRS on tx_{rdm}.

Payments. For subsequent payments (i-th payment) Alice and Bob generate joint signatures on transactions $tx_{\mathsf{rdm},i}$ (that spend from pk_{AB} to some key of Bob) using JSpend interface of $\mathcal{F}_{\mathsf{J-LRS}}$. The transaction and the corresponding signature are stored as $L_{\mathsf{pay}} := (tx_{\mathsf{rdm},i}, \sigma_{\mathsf{rdm},i})$.

Channel Closing. Whenever (before time $\mathbf{T}$) Bob wishes to close the channel he can simply post the most recent $tx_{\mathsf{rdm},i}$ and the corresponding valid signature $tx_{\mathsf{rdm},i}$ retrieved from L_{pay}. As it is standard with payment channels, it is imperative that Bob posts a closing transaction before time $\mathbf{T}$ has elapsed. After time $\mathbf{T}$, Alice has had enough time to solve the VTLRS on tx_{rdm} and consequently recover a valid signature on it. Thus, she can redeem all unspent coins from the channel by posting tx_{rdm} and σ on the blockchain.

Below is the theorem arguing the security of PayMo for Monero. The proof can be found in Appendix G.

Theorem 6.1. *Let Π_{VTLRS} be a VTLRS transaction scheme with privacy and soundness as defined in Theorems 5.3 and 5.4 and Π_{LRS} be a private, non-slanderable and linkable, transaction scheme. Then, the payment channel protocol* PayMo *described in Figure 6 with access to $(\mathcal{F}_{\mathsf{J-LRS}}, \mathcal{F}_{\mathbb{B}}, \mathcal{F}_{\mathsf{smt}}, \mathcal{F}_{\mathsf{clock}})$ UC realizes the functionality $\mathcal{F}_{\mathsf{PC}}$ Figure 4.*

The savvy reader might notice that VTLRS hides the signature only for a bounded (polynomial) amount of time, which seems to be at odds with the standard UC setting, where the environment is an arbitrary PPT machine of potentially unbounded depth. We however stress that our simulator does not make any assumption on the depth of the distinguisher and the security of VTLRS is called only in intermediate hybrids. We refer the reader to Appendix G for more details.

7 Atomic Swap with Monero

Our main technical tool to build atomic swaps for Monero is a construction of an Anonymous Multi Hop Lock (AMHL) for the transaction scheme of Monero. Using this, parties can setup payment locks that enforce the atomicity of the atomic swap. Once the appropriate AMHL is built, the atomic swaps

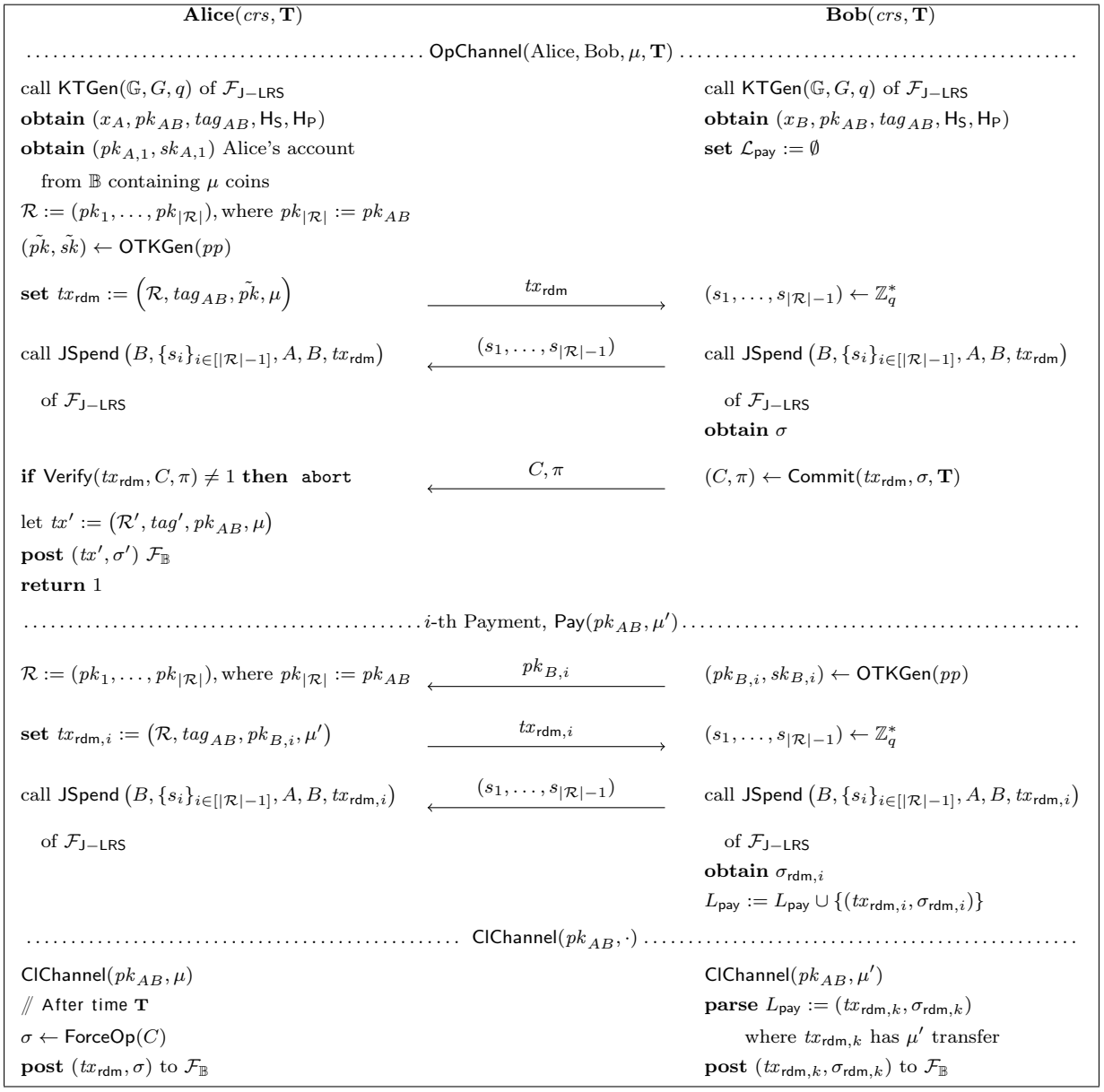

Figure 6: PayMo protocol in Monero using VTLRS

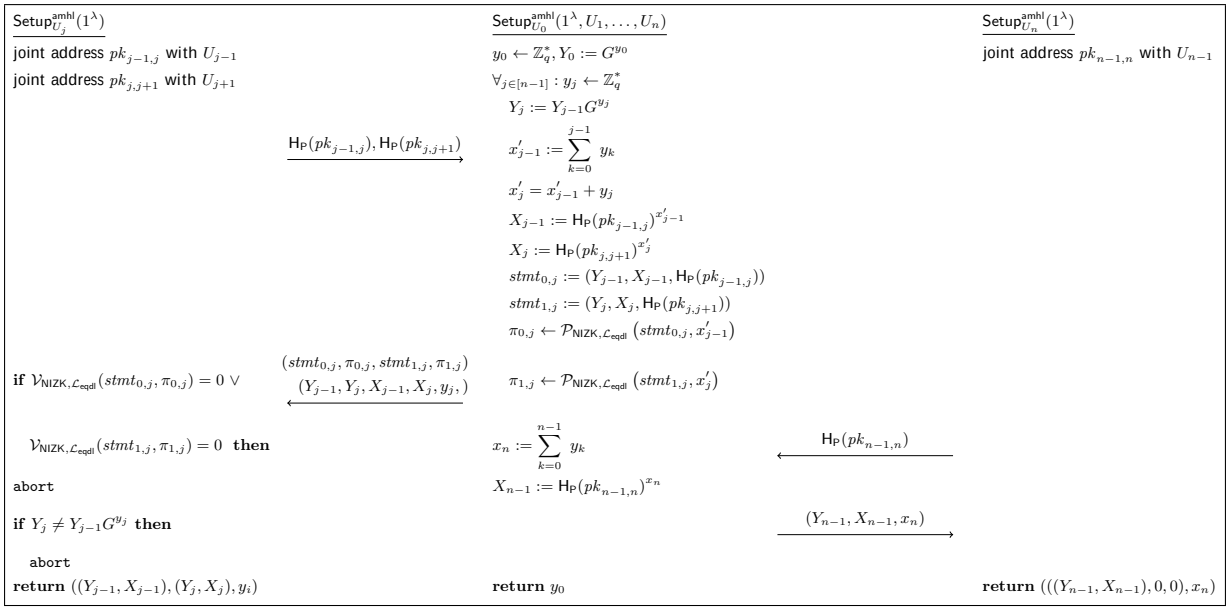

Figure 7: Setup procedure for AMHL in Monero

protocol follows immediately, as discussed in [5] (see also Section 2.3 for an informal overview). Note that in [5], the authors instead required AMHL to build a more powerful primitive of Payment Channel Networks (PCNs), whereas we are only interested in building an atomic swap protocol.

AMHL in Monero. An AMHL allows several parties to establish $n+1$ locks in a path, denoted by $\ell_0, \ell_1, \ldots, \ell_n$. On a high level, AMHL guarantees that the lock ℓ_i can be "unlocked" if and only if ℓ_{i+1} is also released. Furthermore, locks are associated with signatures over transactions and unlocking ℓ_i implies that the i-th party learns the corresponding signature σ_i. Given an AMHL, one can setup a multi-hop payment by locking transactions over all the intermediaries and release the last lock only once the receiver is reached. This triggers a cascade reaction where each intermediary unlocks the corresponding signature σ_i and can therefore redeem its payment.

Thus, the challenge that we face is to design an AMHL scheme that supports the LRS transaction scheme of Monero. Our approach is largely based on the scheme presented in [5] and we give a brief overview below.

7.1 Formal Description of Our AMHL Protocol for Monero

As in Section 6, we make use of the hash function $\mathsf{H} : \{0,1\}^* \to \{0,1\}^\lambda$, and a NIZK proof system $(\mathcal{P}_{\mathsf{NIZK}}, \mathcal{V}_{\mathsf{NIZK}})$ for the language $\mathcal{L}_{\mathsf{eqdl}}$ (as defined in Appendix F).

Overview. In the setup procedure, the first user in the path (sender) provides each intermediary U_j with a tuple of the form $((Y_{j-1}, X_{j-1}), (Y_j, X_j), y_j)$ such that $Y_j = Y_{j-1}G^{y_j}$. Additionally, the sender sends a NIZK proof π_j that certifies that the descrete logarithm of Y_{j-1} with respect to G and X_{j-1} with respect to $\mathsf{H_P}(pk_{j-1,j})$ (the tag of the address) are equal. The same statement is also proven for Y_j and X_j.

The lock procedure is run between two adjoint users U_j and U_{j+1} and ensures that each user receives the respective lock (s^R and s^L) for a transaction tx and joint public key pk. Here $s^R := s'_0$ and $s^L := (s'_0, \ldots, s_{|\mathcal{R}|-1}, h_0)$. The values are generated in such a way that $\sigma := (s_0, s_1, \ldots, s_{|\mathcal{R}|-1}, h_0)$ is a valid signature on tx under pk and that $s'_0 = s_0 - \sum_{k=0}^{j} y_k$.

Once U_{j+1} releases a valid signature (and thus revels s_0), U_j can then compute $\sum_{k=0}^{j} y_k$ by computing $s_0 - s'_0$. Since he knows y_j, user U_j can compute $\sum_{k=0}^{j-1} y_k$ which is what he requires to release his left lock with user U_{j-1}. The lock release verification is simply the LRS verification for transaction tx.

Security of the AMHL protocol described above is stated in Theorem 7.1 and its formal proof is deferred to Appendix H.

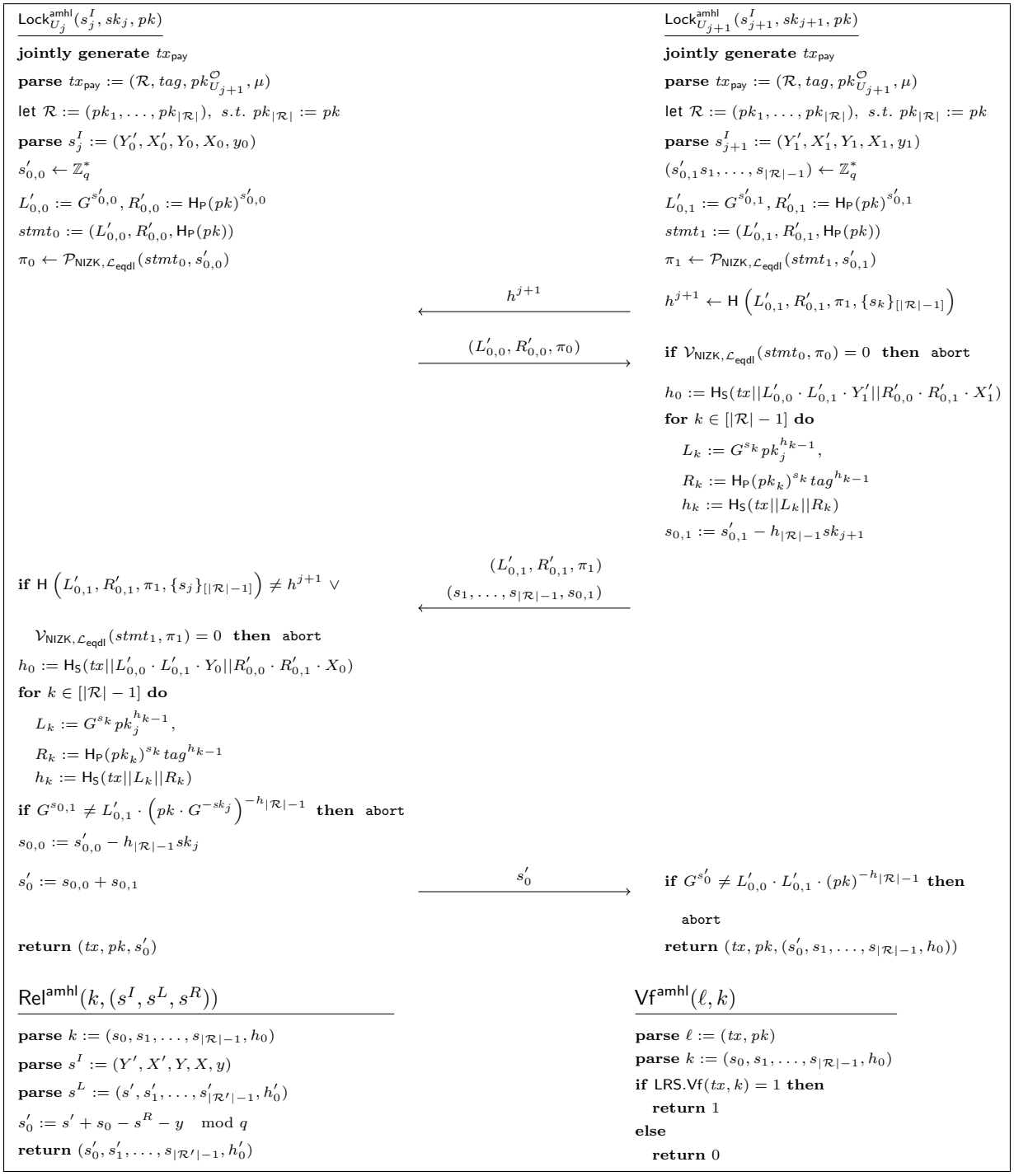

Figure 8: Lock generation, lock release and release verification procedures for AMHL in Monero

Theorem 7.1 (AMHL)**.** *Let* H *be modelled as a random oracle,* LRS *of Monero be a private, non-slanderable and linkable, transaction scheme. Let* $(\mathcal{P}_{\mathsf{NIZK},\mathcal{L}_{\mathsf{eqdl}}}, \mathcal{V}_{\mathsf{NIZK},\mathcal{L}_{\mathsf{eqdl}}})$ *be a NIZK proof of knowledge for the language* $\mathcal{L}_{\mathsf{eqdl}}$ *and let* $\mathsf{H}, \mathsf{H_P}, \mathsf{H_S}$ *be modeled as random oracles. Then the construction in Figures 7 and 8 UC-realizes the ideal functionality* $\mathcal{F}_{\mathsf{AMHL}}$ *in the* $(\mathcal{F}_{\mathsf{J-LRS}}, \mathcal{F}_{\mathsf{sync}}, \mathcal{F}_{\mathsf{smt}})$ *hybrid model.*

Interoperability. Our AMHL protocol is compatible with both the LRS-based transaction scheme of Monero (that uses ed25519 curve) and Schnorr signatures (that uses the ed25519 curve) transaction schemes of [5]. In case of atomic swaps, this means that we can swap a Monero token with any currency

that uses Schnorr signatures with ed25519 curve, for example, Ripple [52] or Cardano [53]. To atomically swap with signatures implemented over different curves, one has to establish an HTLC in the other currency without changing the payment lock in Monero. This requires including a NIZK proof that ensures that the HTLC and the AMHL are computed "consistently", i.e. the pre-image of the HTLC can be recovered by unlocking the lock of the AMHL. For a more comprehensive overview of atomic swaps over different curves, we refer the reader to [12].

8 Benchmarking

We implement prototypes of our VTLRS construction as described in Figure 3 and the PayMo protocol as shown in Figure 6. We build our VTLRS prototype with rust using the curve25519-dalek [54] library and primitives (LRS transaction scheme, and NIZK proof for $\mathcal{L}_{\mathsf{eqdl}}$) built over this curve have the security parameter $\lambda = 128$.

Setup And System Configurations. All measurements were done on on a single CPU core of an AWS t2 micro instance for easier comparison with the following specifications: 1 core of a Intel Xeon E5-2676 v3 @ 2.40Ghz, 1GB of RAM, Ubuntu Linux 18.04.2 LTS (4.15.0-1045-aws) and rust 1.41.

8.1 Cryptographic Blocks

Our evaluation covers several cryptographic primitives that are used as tools in the paper and they are listed below. The measurements of the primitives were executed for 1000 times and the median of all executions is reported.

Hash functions. All hashing operations are implemented using SHA-512 or the Keccak variant used in Monero, with different variants for different hash functions, namely, $\mathsf{H_S}$ (scalar output), $\mathsf{H_P}$ (elliptic curve point output), H (bitstring output), and H' (set of indices of size $t-1$).

Elliptic Curve. For all Elliptic Curve Operations the Curve25519 implementation from curve25519-dalek [54] in Ristretto form [55] was used. This curve was selected to be comparable in terms of speed to Monero.

NIZK Proofs. We implemented the NIZK proof from [56] for $\mathcal{L}_{\mathsf{eqdl}}$ (Appendix F) and the NIZK range proof described in Figure 15 for $\mathcal{L}_{\mathsf{rng}}$. The prover and verifier times for the NIZK proof for $\mathcal{L}_{\mathsf{eqdl}}$ is 0.079ms and 0.143ms, respectively. For the NIZK range proof Figure 15, we have to set an additional statistical security parameter k that influences the soundness of the proof. We report the measurements in Table 1 for two different choices of k. For better security, in all our experiments below, we use $k = 64$ for the range proofs. Note that the choice of k only affects the channel opening time of PayMo and *not* the channel's payment throughput.

Table 1: Measurements for NIZK proof for $\mathcal{L}_{\mathsf{rng}}$ (Figure 15) for different choice of statistical security parameter k.

Parameter k	**Soundness error**	**Prover** (in ms)	**Verifier** (in ms)
32	2.32×10^{-10}	129.33	145.47
64	5.42×10^{-20}	258.66	289.56

Linearly Homomorphic Time-Lock Puzzles (LHTLP). We implemented the LHTLP construction [40] in our prototypes with a 1024 bit RSA modulus N. In our benchmark, the time taken for (one-time) puzzle setup PSetup (including prime generation) is 730.43ms, the time taken for puzzle generation PGen is 3.557ms. And the time taken by PSolve for solving a LHTLP puzzle of timing hardness $\mathbf{T} := 1024, 2048$ and 4096 is 2.708 ms, 4.070ms and 6.795ms, respectively.

VTLRS. We evaluated our VTLRS construction (Figure 3) by setting the cut-and-choose parameters as $n = 80$ and $t = 40$ (probability of adversary breaking soundness is 9.3×10^{-24}). We observe that the time needed for committing and verification in the VTLRS transaction scheme is dominated by PGen. This is because during the generation of the range proofs, the committer needs to generate $k + n = 144$ puzzles and the verifier needs to recompute $(t-1)+1 = 40$ puzzles during verification. Our results show

that Commit and Verify algorithms of our VTLRS construction take 586.76ms and 467.84ms in CPU time, respectively.

8.2 Evaluation of PayMo

For the purposes of benchmarking, we consider two different measurements: (i) Only the computation operations and not the cost of serialisation and network transmission in PayMo. This was done to be able to report a theoretical throughput for one node and show the performance of the protocol on the sender and receiver side of a PayMo channel. (ii) Total time taken by operations including network operations and latency. To show the impact of network transmission in this measurement, two settings with different network latency are considered.

We consider Alice and Bob who share a payment channel. To evaluate the performance of PayMo, we measure the computation time of both users during the channel opening and payment phase. Specifically, we measure the CPU time required by either users individually. Our results are summarised in Table 2. In Table 3, we measure the total time taken for PayMo operations that includes network latency between parties. Our results from Table 3 show that the time taken for finishing a single payment is less a third of second even under high latency scenarios.

Table 2: Evaluation of PayMo: Time taken for PC operations excluding network overhead.

	Alice (in ms)	**Bob** (in ms)
Joint key/tag (Figure 16)	0.13	0.31
Channel opening (Figure 6)	468.1	588.4
Channel payment (Figure 6)	1.30	1.28

Table 3: Evaluation of PayMo: Total time taken for PC operations including network latency. We consider low latency setup **S1** and high latency setup **S2** with Round Trip Times between the two users of 0.3ms and 144 ms, respectively.

	Setup **S1** (in ms)	Setup **S2** (in ms)
Joint key/tag (Figure 16)	1.85	440.7
Channel opening (Figure 6)	1060	1351
Channel payment (Figure 6)	3.61	297.9

Interpretation. Our results from Table 2 show that by exploiting parallel request processing, the receiver of one or more channel(s) can process around 780 payments per second per CPU core, while the sender of one of more channel(s) can process around 770 payments per second per CPU core. The parties can easily scale up their processing power if they spawn more PC nodes (or cores) as done in the Lightning Network.

For instance, from the perspective of a payment service provider (as the receiver), it can accept more than 93600 payments per CPU core over a span of 2 minutes (average block production rate in Monero), from users with PayMo channels with the service provider. In this case, only the receiver's CPU time for payments is considered, excluding the overhead for serialization and network.

To showcase the power of PayMo, in case of just payments from Alice to Bob and assuming a round trip latency time of 144ms per message, Alice and Bob can process close to 93500 payments per CPU core (with acknowledgement of payment) over a span of 2 minutes. This is because during message transmission, parties do not stay idle but instead spawn new payments in parallel. In case the parties do not exploit such parallelism and only make sequential payments, Alice can still make more than 400 payments over the span of 2 minutes.

9 Conclusions

We presented PayMo, the first payment channel protocol that is fully compatible with Monero, the largest privacy-preserving cryptocurrency. Our results show an increase in the transaction throughput of several orders of magnitudes when compared with the current implementation of Monero. As an exciting next step, we plan to test the large scale adoption of our approach for real transactions in Monero.

Acknowledgments

The work was also partially supported by the German research foundation (DFG) through the collaborative research center 1223, and by the state of Bavaria at the Nuremberg Campus of Technology (NCT). NCT is a research cooperation between FAU and the Technische Hochschule Nürnberg Georg Simon Ohm (THN).

References

[1] [Online]. Available: https://www.blockchain.com/en/charts/transactions-per-second.

[2] [Online]. Available: https://bitinfocharts.com/comparison/transactionfees-btc-xmr.html.

[3] J. Poon and T. Dryja, *The bitcoin lightning network: Scalable o -chain instant payments*, 2016.

[4] G. Malavolta, P. Moreno-Sanchez, A. Kate, M. Maffei, and S. Ravi, "Concurrency and privacy with payment-channel networks," in *ACM CCS 17*, ACM Press, 2017, pp. 455–471. DOI: 10.1145/3133956.3134096.

[5] G. Malavolta, P. Moreno-Sanchez, C. Schneidewind, A. Kate, and M. Maffei, "Anonymous multi-hop locks for blockchain scalability and interoperability," in *NDSS 2019*, The Internet Society, 2019.

[6] C. Egger, P. Moreno-Sanchez, and M. Maffei, "Atomic multi-channel updates with constant collateral in bitcoin-compatible payment-channel networks," in *ACM CCS 19*, ACM Press, 2019, pp. 801–815. DOI: 10.1145/3319535.3345666.

[7] *Lightning network*, https://lightning.network/.

[8] *Raiden network*, https://raiden.network/.

[9] *Payment channels in ripple*, https://xrpl.org/use-payment-channels.html.

[10] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from bitcoin," in *2014 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, May 2014, pp. 459–474. DOI: 10.1109/SP.2014.36.

[11] M. Green and I. Miers, "Bolt: Anonymous payment channels for decentralized currencies," in *ACM CCS 17*, ACM Press, 2017, pp. 473–489. DOI: 10.1145/3133956.3134093.

[12] P. Moreno-Sanchez, D. V. Le, S. Noether, B. Goodell, and A. Kate, "Dlsag: Non-interactive refund transactions for interoperable payment channels in monero," Cryptology ePrint Archive, Report 2019/595, 2019, https://eprint. iacr. org ..., Tech. Rep., 2019.

[13] *Market capitalisation*. [Online]. Available: https://coinmarketcap.com.

[14] S.-F. Sun, M. H. Au, J. K. Liu, and T. H. Yuen, "RingCT 2.0: A compact accumulator-based (linkable ring signature) protocol for blockchain cryptocurrency monero," in *ESORICS 2017, Part II*, S. N. Foley, D. Gollmann, and E. Snekkenes, Eds., ser. LNCS, vol. 10493, Springer, Heidelberg, Sep. 2017, pp. 456–474. DOI: 10.1007/978-3-319-66399-9_25.

[15] W. A. A. Torres, V. Kuchta, R. Steinfeld, A. Sakzad, J. K. Liu, and J. Cheng, "Lattice RingCT V2.0 with multiple input and multiple output wallets," in *ACISP 19*, ser. LNCS, Springer, Heidelberg, 2019, pp. 156–175. DOI: 10.1007/978-3-030-21548-4_9.

[16] T. H. Yuen, S. feng Sun, J. K. Liu, M. H. Au, M. F. Esgin, Q. Zhang, and D. Gu, *RingCT 3.0 for blockchain confidential transaction: Shorter size and stronger security*, Cryptology ePrint Archive, Report 2019/508, https://eprint.iacr.org/2019/508, 2019.

[17] R. W. F. Lai, V. Ronge, T. Ruffing, D. Schröder, S. A. K. Thyagarajan, and J. Wang, "Omniring: Scaling private payments without trusted setup," in *ACM CCS 19*, ACM Press, 2019, pp. 31–48. DOI: 10.1145/3319535.3345655.

[18] *What can we expect from monero in 2020?* https://www.reddit.com/r/Monero/comments/eehfp2/what_can_we_expect_about_monero_in_2020/.

[19] R. L. Rivest, A. Shamir, and D. A. Wagner, "Time-lock puzzles and timed-release crypto," Cambridge, MA, USA, Tech. Rep., 1996.

[20] K. Pietrzak, "Simple verifiable delay functions," in *ITCS 2019*, Jan. 2019, 60:1–60:15. DOI: 10.4230/LIPIcs.ITCS.2019.60.

[21] B. Wesolowski, "Efficient verifiable delay functions," pp. 379–407. DOI: 10.1007/978-3-030-17659-4_13.

[22] N. Ephraim, C. Freitag, I. Komargodski, and R. Pass, "Continuous verifiable delay functions," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2020, pp. 125–154.

[23] *Chia network*, https://www.chia.net/.

[24] *Vdf research*, https://vdfresearch.org/.

[25] *Protocol labs and ethereum foundation team up to research verifiable delay functions*, https://cointelegraph.com/news/protocol-labs-and-ethereum-foundation-team-up-to-research-verifiable-delay-functions.

[26] B. Cohen and K. Pietrzak, *The chia network blockchain*, 2019.

[27] *Chia network competition*, https://github.com/Chia-Network/vdf-competition, https://medium.com/@chia.net/chia-vdf-competition-guide-5382e1f4bd39.

[28] *Verifiable delay functions and attacks*, https://ethresear.ch/t/verifiable-delay-functions-and-attacks/2365.

[29] B. Bünz, S. Goldfeder, and J. Bonneau, "Proofs-of-delay and randomness beacons in ethereum," *IEEE Security and Privacy on the blockchain (IEEE S&B)*, 2017.

[30] S. A. K. Thyagarajan, A. Bhat, G. Malavolta, N. Döttling, A. Kate, and D. Schröder, "Verifiable timed signatures made practical," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '20, Virtual Event, USA: Association for Computing Machinery, 2020, 1733–1750, ISBN: 9781450370899. DOI: 10.1145/3372297.3417263. [Online]. Available: https://doi.org/10.1145/3372297.3417263.

[31] J. Kilian, "A note on efficient zero-knowledge proofs and arguments (extended abstract)," in *24th ACM STOC*, ACM Press, May 1992, pp. 723–732. DOI: 10.1145/129712.129782.

[32] S. Micali, "CS proofs (extended abstracts)," in *35th FOCS*, IEEE Computer Society Press, Nov. 1994, pp. 436–453. DOI: 10.1109/SFCS.1994.365746.

[33] R. Gennaro, C. Gentry, B. Parno, and M. Raykova, "Quadratic span programs and succinct NIZKs without PCPs," in *EUROCRYPT 2013*, T. Johansson and P. Q. Nguyen, Eds., ser. LNCS, vol. 7881, Springer, Heidelberg, May 2013, pp. 626–645. DOI: 10.1007/978-3-642-38348-9_37.

[34] S. Dziembowski, L. Eckey, S. Faust, and D. Malinowski, "Perun: Virtual payment hubs over cryptocurrencies," in *2019 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, 2019, pp. 106–123. DOI: 10.1109/SP.2019.00020.

[35] S. Dziembowski, S. Faust, and K. Hostáková, "General state channel networks," in *ACM CCS 18*, ACM Press, 2018, pp. 949–966. DOI: 10.1145/3243734.3243856.

[36] A. Miller, I. Bentov, S. Bakshi, R. Kumaresan, and P. McCorry, "Sprites and state channels: Payment networks that go faster than lightning," in *FC 2019*, ser. LNCS, Springer, Heidelberg, 2019, pp. 508–526. DOI: 10.1007/978-3-030-32101-7_30.

[37] J. A. Garay and M. Jakobsson, "Timed release of standard digital signatures," in *FC 2002*, M. Blaze, Ed., ser. LNCS, vol. 2357, Springer, Heidelberg, Mar. 2003, pp. 168–182.

[38] J. A. Garay and C. Pomerance, "Timed fair exchange of standard signatures: [extended abstract]," in *FC 2003*, R. Wright, Ed., ser. LNCS, vol. 2742, Springer, Heidelberg, Jan. 2003, pp. 190–207.

[39] A. Fiat and A. Shamir, "How to prove yourself: Practical solutions to identification and signature problems," in *CRYPTO'86*, A. M. Odlyzko, Ed., ser. LNCS, vol. 263, Springer, Heidelberg, Aug. 1987, pp. 186–194. DOI: 10.1007/3-540-47721-7_12.

[40] G. Malavolta and S. A. K. Thyagarajan, "Homomorphic time-lock puzzles and applications," pp. 620–649. DOI: 10.1007/978-3-030-26948-7_22.

[41] A. De Santis, S. Micali, and G. Persiano, "Non-interactive zero-knowledge proof systems," in *Conference on the Theory and Application of Cryptographic Techniques*, Springer, 1987, pp. 52–72.

[42] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[43] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," in *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, IEEE, 2001, pp. 136–145.

[44] R. Canetti, Y. Dodis, R. Pass, and S. Walfish, "Universally composable security with global setup," in *TCC 2007*, S. P. Vadhan, Ed., ser. LNCS, vol. 4392, Springer, Heidelberg, Feb. 2007, pp. 61–85. DOI: 10.1007/978-3-540-70936-7_4.

[45] J. Katz, U. Maurer, B. Tackmann, and V. Zikas, "Universally composable synchronous computation," in *TCC 2013*, A. Sahai, Ed., ser. LNCS, vol. 7785, Springer, Heidelberg, Mar. 2013, pp. 477–498. DOI: 10.1007/978-3-642-36594-2_27.

[46] S. Dziembowski, L. Eckey, S. Faust, J. Hesse, and K. Hostáková, "Multi-party virtual state channels," pp. 625–656. DOI: 10.1007/978-3-030-17653-2_21.

[47] L. Aumayr, O. Ersoy, A. Erwig, S. Faust, K. Hostakova, M. Maffei, P. Moreno-Sanchez, and S. Riahi, "Generalized bitcoin-compatible channels.," *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 476, 2020.

[48] R. L. Rivest, A. Shamir, and Y. Tauman, "How to leak a secret," in *International Conference on the Theory and Application of Cryptology and Information Security*, Springer, 2001, pp. 552–565.

[49] J. K. Liu, V. K. Wei, and D. S. Wong, "Linkable spontaneous anonymous group signature for ad hoc groups (extended abstract)," in *ACISP 04*, H. Wang, J. Pieprzyk, and V. Varadharajan, Eds., ser. LNCS, vol. 3108, Springer, Heidelberg, Jul. 2004, pp. 325–335. DOI: 10.1007/978-3-540-27800-9_28.

[50] G. Maxwell, "Confidential transactions," Available at https://people.xiph.org/~greg/confidential_values.txt, 2015.

[51] A. Nicolosi, M. N. Krohn, Y. Dodis, and D. Mazières, "Proactive two-party signatures for user authentication," in *NDSS 2003*, The Internet Society, Feb. 2003.

[52] *Ripple ledger cryptographic keys*, https://xrpl.org/cryptographic-keys.html.

[53] *Cryptocurrencies, signing algorithms and curves*, https://www.susanka.eu/coins-crypto/.

[54] "Curve25519-dalek," 2019. [Online]. Available: https://github.com/dalek-cryptography/curve25519-dalek.

[55] d. V. H. L. I. Arcieri T., "The ristretto group," 2019. [Online]. Available: https://ristretto.group/ristretto.html.

[56] J. Camenisch and M. Stadler, "Proof systems for general statements about discrete logarithms," *Technical report/Dept. of Computer Science, ETH Zürich*, vol. 260, 1997.

[57] M. Bellare and G. Neven, "Multi-signatures in the plain public-key model and a general forking lemma," in *ACM CCS 06*, A. Juels, R. N. Wright, and S. Vimercati, Eds., ACM Press, 2006, pp. 390–399. DOI: 10.1145/1180405.1180453.

[58] *ACM CCS 19*, ACM Press, 2019.

[59] *ACM CCS 17*, ACM Press, 2017.

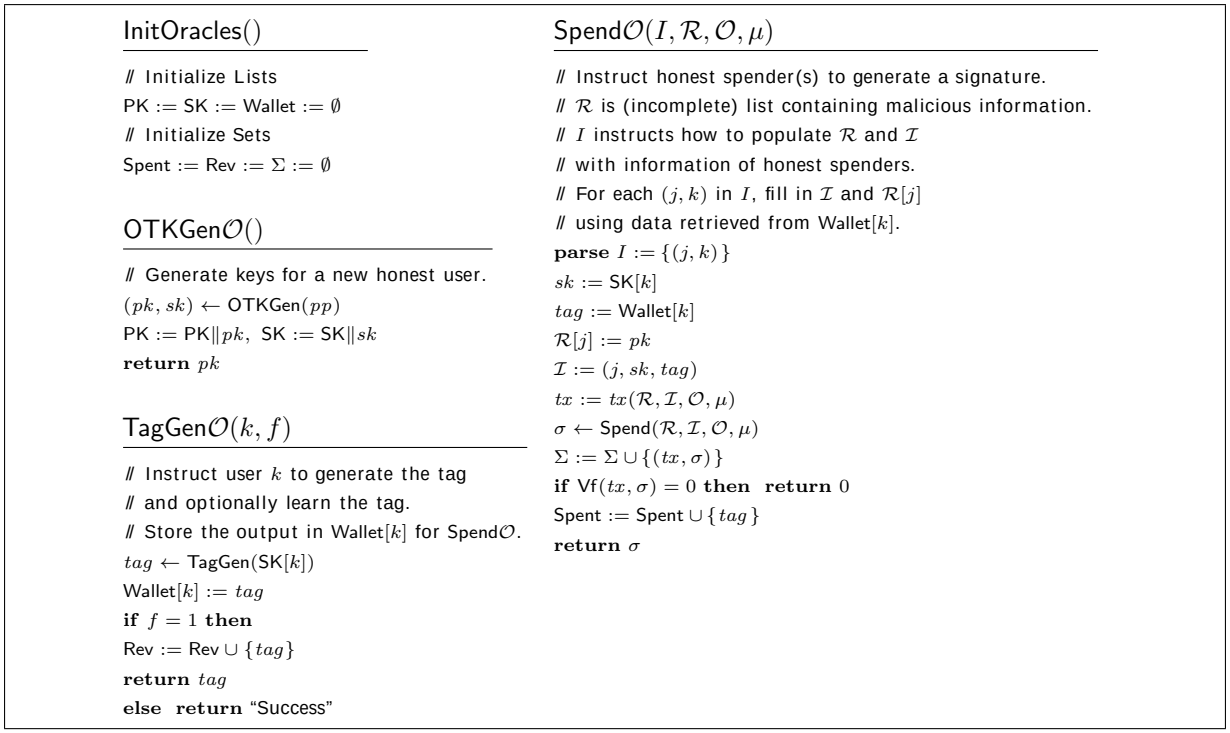

Figure 9: Oracles for Security Experiments

A Formal Definitions of LRS-TS

For the formalization of our security notions, we need to define the oracles shown in Figure 9. The one-time key generation oracle OTAccGen$\mathcal{O}$ generates a fresh one-time key pair and returns the one-time public key as the response to the oracle. The spend oracle Spend$\mathcal{O}$ takes as input the index of the source one-time public key, the ring, a set of target public keys and metadata. The oracle retrieves the corresponding one-time secret key and the tag previously generated and then runs the Spend algorithm to generate a signature σ. It returns σ as the oracle reply. The tag generation oracle TagGen$\mathcal{O}$ takes as input the index k and a flag value f. It retrieves the one-time secret key generated before and generates the corresponding tag tag using TagGen algorithm. It additionally checks if the flag f is set to 1. If so, it adds tag returns tag as the reply of the oracle. Otherwise, it just returns a simple success message and does not reveal the tag.

A.1 Privacy

In order to define privacy, we consider an adversary that takes part in the PrivExp experiment. The adversary is given access to the one-time key generation oracle, the spending oracle and the tag generation oracle. It then outputs two challenge indices representing two one-time public keys previously generated during some oracle query. It also outputs a ring, a set of target accounts and some metadata. Both the one-time public keys are included in the ring The secret keys and the tags of both the one-time keys are retrieved The experiment runs the spend algorithm and generates a signature using one of the secret keys. The signature and the transaction are given to the adversary. The adversary finally outputs a bit guessing which of the one-time keys was the spending key. The adversary is said to win the experiment if the guess is correct, and the tags have not been revealed to the adversary in some oracle query. The formal specification is given in Figure 10 and the definition of the privacy property is given in Theorem A.1.

Definition A.1 (LRS-TS Privacy)**.** *A LRS transaction scheme is* private *if for all* PPT *adversaries* $\mathcal{A}$ *and all positive integers* $\alpha \in \mathsf{poly}(\lambda)$*,*

$$Pr\left[\mathsf{PrivExp}_{\mathsf{LRS},\mathcal{A}}(1^\lambda) = 1\right] \leq \tfrac{1}{2} + \mathsf{negl}(\lambda)$$

where $\mathsf{PrivExp}^{b}_{\mathsf{LRS},\mathcal{A}}$ *is defined in Figure 10.*

$\mathsf{PrivExp}_{\mathsf{LRS},\mathcal{A}}(1^\lambda)$

$pp \leftarrow \mathsf{Setup}(1^\lambda),\ \mathsf{InitOracles}()$
$\mathbb{O} := \{\mathsf{OTKGen}\mathcal{O}, \mathsf{Spend}\mathcal{O}, \mathsf{TagGen}\mathcal{O}\}$
$(I, \mathcal{R}, \mathcal{O}, \mu) \leftarrow \mathcal{A}^{\mathbb{O}}(pp)$
$\mathcal{I}_0 := \mathcal{I}_1 := \emptyset$ // Preparing honest spenders as instructed by adversary.
parse I **as** $\{j_t, k_t^{\mathcal{I}}\}_{t=0}^{1}$
for $t \in \{0, 1\}$
  $sk_t^{\mathcal{I}} := \mathsf{SK}[k_t^{\mathcal{I}}]$
  $tag_t := \mathsf{Wallet}[k_t^{\mathcal{I}}]$
  $\mathcal{R}[j_t] := pk_t^{\mathcal{I}}$
endfor
$b \leftarrow \{0, 1\}$
$\mathcal{I}_b := (j_b, sk_b^{\mathcal{I}}, tag_b)$
$tx_b := tx(\mathcal{R}, \mathcal{I}_b, \mathcal{O}, \mu)$
$\sigma_b \leftarrow \mathsf{Spend}(\mathcal{R}, \mathcal{I}_b, \mathcal{O}, \mu)$
if $\mathsf{Vf}(tx_t, \sigma_t) = 0$ **then return** 0
$b' \leftarrow \mathcal{A}^{\mathbb{O}}(tx_b, \sigma_b)$
$b_0 := b = b'$
$b_1 := tag_0 \neq tag_1$
$b_2 := \{tag_0, tag_1\} \cap \mathsf{Spent} = \emptyset$
$b_3 := \{tag_0, tag_1\} \cap \mathsf{Rev} = \emptyset$
return $b_0 \wedge b_1 \wedge b_2 \wedge b_3$

Figure 10: Privacy Experiment

A.2 Non-Slanderability (and Unforgeability)

As explained above, slandering is the act of producing a valid signature on a trasnaction on behalf of another user. Formally, we model non-slanderability (Theorem A.2) by defining a security experiment in which the adversary produces a transaction-signature tuple, after several queries to the oracles (Figure 9). The adversary is successful if the tuple is valid, not produced by the spend oracle, and the tag specified in the slandering transaction was previously obtained in a spend oracle query or a tag generation oracle query. Since a tag is computationally bound to a unique one-time key (as required by the linkability property Figure 12), non-slanderability (which states that no adversary can forge under a tag of a honest user key), naturally implies that no adversary can forge signatures for honest keys. As a consequence, we do not need to define an unforgeability property explicitly.

Definition A.2 (LRS-TS Non-slanderability)**.** *A LRS transaction scheme is* non-slanderable *if for all* PPT *adversaries* $\mathcal{A}$ *and all* $\alpha \in \mathsf{poly}(\lambda)$*,*

$$Pr\left[\mathsf{NSlandExp}_{\mathsf{LRS},\mathcal{A}}(1^\lambda) = 1\right] \leq \mathsf{negl}(\lambda)$$

where the experiment $\mathsf{NSlandExp}_{\mathsf{LRS},\mathcal{A}}$ *is defined in Figure 11.*

A.3 Linkability

Linkability (Theorem A.3) roughly means that a user cannot double-spend coins from an account. In more detail, we say that a LRS transaction scheme is balanced if the following two properties are satisfied. First, the predicate CheckTag is required to be “binding" in a sense similar to a commitment scheme. The binding property of CheckTag ensures that a tag is computationally bound to a source one-time key (i.e., it is hard to come up with two tags for the same one-time public key), which in turn ensures that checking for duplicate tags is sufficient to prevent double-spending. The second property requires that,

$\mathsf{NSlandExp}_{\mathsf{LRS},\mathcal{A}}(1^\lambda)$

$pp \leftarrow \mathsf{Setup}(1^\lambda),\ \mathsf{InitOracles}()$

$(tx^*, \sigma^*) \leftarrow \mathcal{A}^{\mathsf{OTKGen}_{\mathcal{O}},\mathsf{TagGen}_{\mathcal{O}},\mathsf{Spend}_{\mathcal{O}}}(pp)$

$\mathbf{parse}\ tx^*\ \mathbf{as}\ \left(\left\{pk_i^{\mathcal{R}}\right\}_{i=1}^{|\mathcal{R}|}, tag, \left\{pk_i^{\mathcal{O}}\right\}_{i=1}^{|\mathcal{O}|}, \mu\right)$

$b_0 := \mathsf{Vf}(tx^*, \sigma^*)$

$b_1 := ((tx^*, \sigma^*) \notin \Sigma)$

$b_2 := (tag \in \mathsf{Spent} \vee tag \in \mathsf{Rev})$

$\mathbf{return}\ b_0 \wedge b_1 \wedge b_2$

Figure 11: Non-slanderability Experiment

for any efficient adversary $\mathcal{A}$ which produces a transaction with a signature, there exists an extractor $\mathcal{E}_{\mathcal{A}}$ such that, if the signature is valid (Event 0), then the probability of the following inconsistency occurring is negligible. (Event 1), the extractor $\mathcal{E}_{\mathcal{A}}$ extracted an index j and a secret key sk, yet sk is inconsistent with the j-th ring key $pk_j^{\mathcal{R}}$ the tag tag according to the predicate ChkTag, i.e., $\mathsf{ChkTag}(pk_j^{\mathcal{R}}, sk, tag) = 0$. The two properties can be interpreted in the following way. If the spender attempts to spend from the same account twice by producing different tags for the account, then with the spender and the extractor $\mathcal{E}_{\mathcal{A}}$ one can break the binding property of ChkTag. Therefore double-spending is infeasible.

Definition A.3 (LRS-TS Linkability)**.** *A LRS transaction scheme is* linkable *if:*

1. ChkTag *is binding. That is, for any* PPT *adversary* $\mathcal{A}$*, for all positive integers* $\alpha \in \mathsf{poly}(\lambda)$*,*

$$Pr\left[\begin{array}{c|c} \mathsf{ChkTag}(pk, sk, tag) = 1 & pp \leftarrow \mathsf{Setup}(1^\lambda) \\ \mathsf{ChkTag}(pk, sk', tag') = 1 & \left(\begin{array}{c} pk, \\ \end{array}\begin{array}{c} sk, tag, \\ sk', tag' \end{array}\right) \leftarrow \mathcal{A}(pp) \\ (sk, \mathsf{tag}) \neq (sk', \mathsf{tag}') & \end{array}\right] \leq \mathsf{negl}(\lambda)$$

2. *For all* PPT *adversaries* $\mathcal{A}$*, and all positive integers* $\alpha \in \mathsf{poly}(\lambda)$*, there exists a* PPT *extractor* $\mathcal{E}_{\mathcal{A}}$ *such that*

$$Pr\left[\mathsf{LinkExp}_{\mathsf{LRS},\mathcal{A},\mathcal{E}_{\mathcal{A}}}(1^\lambda, 1^\alpha) = 1\right] \leq \mathsf{negl}(\lambda)$$

where $\mathsf{LinkExp}_{\mathsf{LRS},\mathcal{A},\mathcal{E}_{\mathcal{A}}}(1^\lambda, 1^\alpha)$ *is defined in Figure 12.*

$\mathsf{LinkExp}_{\mathsf{LRS},\mathcal{A},\mathcal{E}_{\mathcal{A}}}(1^\lambda, 1^\alpha)$

$pp \leftarrow \mathsf{Setup}(1^\lambda)$

$(tx, \sigma) \leftarrow \mathcal{A}(pp)$

$(\mathcal{R}, \mathcal{I}, \mathcal{O}, \mu) \leftarrow \mathcal{E}_{\mathcal{A}}(pp, tx, \sigma)$

$\mathbf{parse}\ \mathcal{R}\ \mathbf{as}\ \left\{pk_i^{\mathcal{R}}\right\}_{i=1}^{|\mathcal{R}|}$

$\mathbf{parse}\ \mathcal{I}\ \mathbf{as}\ (j, sk, tag)$

$b_0 := \mathsf{Vf}(tx, \sigma)$

$b_1 := \mathsf{ChkTag}(pk_j^{\mathcal{R}}, sk, tag) = 0$

$\mathbf{return}\ b_0 \wedge b_1$

Figure 12: Linkability Experiment

A.4 Security Analysis of Monero's LRS (Figure 2)

A.4.1 Privacy

Intuition. The adversary is challenged in distinguishing a signature that was generated by one of the two challenge indices. The challenge indices and the challenge ring are of the adversary's choice. If the

adversary identifies the signer correctly then the signature is assumed to be correctly generated with a valid DDH instance. If the adversary fails, then the DDH instance is assumed to be an invalid one.

Privacy. Consider an adversary $\mathcal{A}$ that violates the privacy of the LRS scheme of Monero. This means that we have

$$\Pr\left[\mathsf{PrivExp}_{\mathsf{LRS},\mathcal{A}}(1^\lambda, 1^\alpha) = 1\right] > \frac{1}{2} + \mathsf{negl}(\lambda)$$

We construct a reduction $\mathcal{S}$ that simulates the privacy experiment for $\mathcal{A}$ and based on its output solve the DDH problem. $\mathcal{S}$ answers the oracle queries made by $\mathcal{A}$ that it is given access to in the privacy experiment. Specifically, let us denote the total queries made by $\mathcal{A}$ to $\mathsf{OTKGen}\mathcal{O}$ as q_K, to $\mathsf{Spend}\mathcal{O}$ as q_S and to $\mathsf{TagGen}\mathcal{O}$ as q_T. Additionally, $\mathcal{A}$ can makes $q_{\mathsf{H_P}}$ and $q_{\mathsf{H_S}}$ oracle queries to $\mathsf{H_P}$ and $\mathsf{H_S}$ respectively. We assume w.l.o.g that the adversary $\mathcal{A}$ does not make redundant queries to any of the oracles. The total number of queries made by $\mathcal{A}$ to all oracles are upper bounded by some polynomial $p(\lambda)$.

We now describe the reduction procedure. $\mathcal{S}$ first receives as input $(\mathbb{G}, q, G, G^a, G^b, G^c)$ as the DDH challenge. It runs the setup for LRS by running $\mathsf{Setup}(1^\lambda, 1^\alpha)$ and obtains pp. It then invokes $\mathcal{A}$ by giving it pp as input.

Oracle Queries $\mathcal{S}$ guesses some $i \in [q_K]$ such that in the i-th query to the $\mathsf{OTKGen}\mathcal{O}$ it sets $pk := G^a$. It adds pk as the i-th entry to PK and set the i-th entry of SK as $\perp$. It then returns pk.

For the spend oracle queries made by $\mathcal{A}$ of the form $(\{j,k\}, \mathcal{R}, \mathcal{O}, \mu)$ if $k = i$, then $\mathcal{S}$ aborts the execution. For all other cases of $k \neq i$, the reduction $\mathcal{S}$ runs the spend oracle as described in Figure 9 where it generates the signature honestly according to the algorithm in Figure 2.

For the tag generation queries from $\mathcal{A}$ of the form (k, f), if $k \neq i$, the reduction retrieves the secret key from SK and generates the tag as specified in Figure 9. This involves checking if $\mathsf{H_P}(G^{\mathsf{SK}[k]})$ is already set, if not, it calls the $\mathsf{H_P}$ oracle and retrieves the response. If additionally, $f = 1$, the reduction reveals the tag tag. If $k = i$ and $f = 0$, the reduction adds G^c to $\mathsf{Wallet}[k]$.

For oracle queries to $\mathsf{H_P}$ of the form pk, the reduction checks if $pk = G^a$, and if so, sets the oracle reply as G^b. For other cases, the reduction samples fresh random coins $r_j \leftarrow \mathbb{Z}_q^*$ for the j-th query and sets the response as G^{r_j}. It records the oracle response (pk, g^{r_j}) into the list $\mathcal{L}_{\mathsf{H_P}}$.

For oracle queries to $\mathsf{H_S}$ of the form $(tx||L||R)$, the reduction samples random coins $r'_j \leftarrow \mathbb{Z}_q^*$ and responds to the adversary with r'_j.

Challenge phase $\mathcal{S}$ obtains the response from $\mathcal{A}$ in the form of $(I, \mathcal{R}, \mathcal{O}, \mu)$, where $|\mathcal{R}| = |\mathcal{R}|$. Parse $I := \{j_0, k_0, j_1, k_1\}$. If $k_0 \neq i$ and $k_1 \neq i$, the reduction aborts by outputting $\mathtt{abort}_1$. For ease of understanding w.l.o.g, let us assume that $k_0 = i$ and that $j_0 = |\mathcal{R}|$. The reduction sets $\mathcal{R}[j_0] = G^a$ and retrieves G^c from $\mathsf{Wallet}[k_0]$. It then sets $\mathcal{R}[j_1] = \mathsf{PK}[k_1]$ and parse $\mathcal{R} := (pk_1, \ldots, pk_{|\mathcal{R}|})$. In order to generate the challenge signature, the reduction does the following steps:

- Set $tx^* := \{\mathcal{R}, G^c, \mathcal{O}, \mu\}$
- Sample $h_{|\mathcal{R}|}, s_{|\mathcal{R}|} \leftarrow \mathbb{Z}_q^*$ and set $h_0 = h_{|\mathcal{R}|}$, for $j \in [|\mathcal{R}| - 1]$,
  - Check if $(pk_j, G^{e_j}) \in \mathcal{L}_{\mathsf{H_P}}$, and if not set $\mathsf{H_P}(pk_j) := G^{e_j}$ for random $e_j \leftarrow \mathbb{Z}_q^*$
  - Sample $s_j, h_j \leftarrow \mathbb{Z}_q^*$ and set $\mathsf{H_S}(tx^*||G^{s_j} pk_j^{h_{j-1}}||G^{e_j s_j}(G^{ch_{j-1}})) = h_j$
- Set $\mathsf{H_S}(tx^*||G^{s_{|\mathcal{R}|}}(G^a)^{h_{|\mathcal{R}|-1}}||(G^b)^{s_{|\mathcal{R}|}}(G^c)^{h_{|\mathcal{R}|-1}}) = h_{|\mathcal{R}|}$
- Set $\sigma^* := (s_{|\mathcal{R}|}, s_1, \ldots, s_{|\mathcal{R}|-1}, h_0)$ and return (tx^*, σ^*) to $\mathcal{A}$.

Note that the random oracle queries set in this challenge phase are not queried by the adversary before this phase except with negligible probability.

For any query made by the adversary to any of the oracles, the reduction answers by ensuring consistency with the query-response set during the previous phases.

Adversary $\mathcal{A}$ outputs a bit b' as its guess of the signer being $\mathcal{R}[j_{b'}]$.

Analysis Let us denote b'' to be the challenge bit for the reduction $\mathcal{S}$: if $b'' = 1$ then the DDH challenge given to $\mathcal{S}$ is a valid DDH tuple and if $b'' = 0$ then the DDH challenge is a random tuple. We have the following:

$$\begin{aligned}\Pr\left[\mathcal{S}(G^a, G^b, G^c) = b''\right] = \\ \Pr\left[\mathcal{S}(G^a, G^b, G^c) = b'' \wedge \mathtt{abort}_1\right] \\ + \Pr\left[\mathcal{S}(G^a, G^b, G^c) = b'' \wedge \neg\mathtt{abort}_1\right]\end{aligned}$$

We have $\Pr\left[\mathcal{S}(\mathbb{G}, q, G, G^a, G^b, G^c) = b'' \wedge \mathtt{abort}_1\right] = 0$ because, $\mathcal{S}$ outputs 0 or 1 only if it does not abort the execution. We now rewrite,

$$\begin{aligned}\Pr\left[\mathcal{S}(G^a, G^b, G^c) = b'' \wedge \neg\mathtt{abort}_1\right] = \\ \Pr\left[\neg\mathtt{abort}_1\right] \cdot \Pr\left[\mathcal{S}(G^a, G^b, G^c) = b'' | \neg\mathtt{abort}_1\right]\end{aligned}$$

We calculate the above probability as follows:

$$\begin{aligned}&\Pr\Big[\mathcal{S}(G^a, G^b, G^c) = b'' \,|\, b'' = 1 \wedge \neg\mathtt{abort}_1\Big] \\ &\quad = \Pr\Big[\mathcal{S}(G^a, G^b, G^c) = b'' \,|\, b'' = 1 \wedge \neg\mathtt{abort}_1 \\ &\qquad \wedge \mathsf{PrivExp}_{\mathsf{LRS},\mathcal{A}}(1^\lambda, 1^\alpha) = 1\Big] \\ &\quad + \Pr\Big[\mathcal{S}(G^a, G^b, G^c) = b'' \,|\, b'' = 1 \wedge \neg\mathtt{abort}_1 \\ &\qquad \wedge \mathsf{PrivExp}_{\mathsf{LRS},\mathcal{A}}(1^\lambda, 1^\alpha) = 0\Big] \\ &\quad \geq 1 \cdot \left(\frac{1}{2} + \mathsf{negl}(\lambda)\right) + \frac{1}{2} \cdot \left(1 - \frac{1}{2} - \mathsf{negl}(\lambda)\right) \\ &\quad = \frac{3}{4} + \frac{\mathsf{negl}(\lambda)}{2}\end{aligned}$$

and

$$\begin{aligned}&\Pr\Big[\mathcal{S}(G^a, G^b, G^c) = b'' \,|\, b'' = 0 \wedge \neg\mathtt{abort}_1\Big] \\ &\quad = \Pr\Big[\mathcal{S}(G^a, G^b, G^c) = b'' \,|\, b'' = 0 \wedge \neg\mathtt{abort}_1 \\ &\qquad \wedge \mathsf{PrivExp}_{\mathsf{LRS},\mathcal{A}}(1^\lambda, 1^\alpha) = 1\Big] \\ &\quad + \Pr\Big[\mathcal{S}(G^a, G^b, G^c) = b'' \,|\, b'' = 0 \wedge \neg\mathtt{abort}_1 \\ &\qquad \wedge \mathsf{PrivExp}_{\mathsf{LRS},\mathcal{A}}(1^\lambda, 1^\alpha) = 0\Big] \\ &\quad \geq 0 \cdot \frac{1}{2} + \frac{1}{2} \cdot \left(1 - \frac{1}{2}\right) = \frac{1}{4}.\end{aligned}$$

The probabilities for $b'' = 1$ and $b'' = 0$ are both $\frac{1}{2}$ as it is a coin toss. We have the factor $\frac{1}{q_K}$ lower bounded by $\frac{1}{p(\lambda)}$ which is the probability with which $\mathcal{S}$ does not abort with output abort_1 (meaning that its guess of the challenge key was correct). This leads to

$$
\begin{aligned}
&\Pr\left[\mathcal{S}(G^a, G^b, G^c) = b''\right] \\
&= \frac{1}{2}\Pr\left[\neg\mathtt{abort}_1\right]\Pr\Big[\mathcal{S}(G^a, G^b, G^c) = b'' \mid \neg\mathtt{abort}_1 \\
&\qquad \wedge (b'' = 1 \vee b'' = 0)\Big] \\
&\geq \frac{2}{p(\lambda)}\frac{1}{2}\left(\frac{3}{4} + \frac{\mathsf{negl}(\lambda)}{2} + \frac{1}{4}\right) \\
&= \frac{2}{p(\lambda)}\left(\frac{1}{2} + \frac{\mathsf{negl}(\lambda)}{4}\right).
\end{aligned}
$$

For the case where $\mathcal{S}$ aborts the simulation by outputting $\mathtt{abort}_1$, it can do no better than guessing to solve the DDH problem instance. From this we learn that $\mathcal{S}$ solves the DDH instance with probability greater than $\left(\frac{1}{2} + \frac{1}{p(\lambda)} + \frac{\mathsf{negl}(\lambda)}{4}\right)$ which is non-negligibly larger than $\frac{1}{2}$. This proves the contradiction and hence privacy follows. □

A.4.2 Non-Slanderability

Intuition. We make use of the forking lemma that is discussed by Bellare et al. [57]. The interaction with the adversary breaking the non-slanderability of our scheme is recorded on a transcript tape. After a successful slander, we rewind this tape to a forking point and repeat the interaction with independent coins except maintaining consistency with the previous exchanges. This forking point is chosen based on the queries to the random oracle and the slander. In the end, we solve the DLP in relation to the forking point.

Proof. Consider an adversary $\mathcal{A}$ that violates the non-slanderability of the LRS scheme of Monero. This means that we have

$$\Pr\left[\mathsf{NSlandExp}_{\mathsf{LRS},\mathcal{A}}(1^\lambda, 1^\alpha) = 1\right] > \epsilon(\lambda)$$

where ϵ is some polynomial. We construct a reduction $\mathcal{S}$ that simulates the non-slanderability experiment for $\mathcal{A}$ and based on its output solve the DL problem. $\mathcal{S}$ answers the oracle queries made by $\mathcal{A}$ that it is given access to in the non-slanderability experiment. Specifically, let us denote the total queries made by $\mathcal{A}$ to $\mathsf{OTKGen}\mathcal{O}$ as q_K, to $\mathsf{Spend}\mathcal{O}$ as q_S and to $\mathsf{TagGen}\mathcal{O}$ as q_T. Additionally, $\mathcal{A}$ can makes $q_{\mathsf{H_P}}$ and $q_{\mathsf{H_S}}$ oracle queries to $\mathsf{H_P}$ and $\mathsf{H_S}$ respectively. We assume w.l.o.g that the adversary $\mathcal{A}$ does not make redundant queries to any of the oracles. The total number of queries made by $\mathcal{A}$ to all oracles are upper bounded by some polynomial $p(\lambda)$.

We now describe the reduction procedure. $\mathcal{S}$ first receives as input $(\mathbb{G}, q, G, G^a)$ as the DL challenge. It runs the setup for LRS by running $\mathsf{Setup}(1^\lambda, 1^\alpha)$ and obtains pp. It then invokes $\mathcal{A}$ by giving it pp as input.

Oracle Queries $\mathcal{S}$ guesses some $i \in [q_K]$ such that in the i-th query to the $\mathsf{OTKGen}\mathcal{O}$. It sets $pk = G^a$ and $sk = \perp$. For all other queries, it generates $(pk, sk) \leftarrow \mathsf{OTKGen}(pp)$. It then adds pk to the list PK and sk to the list SK. It then returns pk to the adversary $\mathcal{A}$.

For oracle queries to $\mathsf{H_P}$ of the form pk, the reduction samples fresh random coins $r_j \leftarrow \mathbb{Z}_q^*$ for the j-th query and sets the response as G^{r_j}. It records the oracle response (pk, G^{r_j}, r_j) into the list $\mathcal{L}_{\mathsf{H_P}}$.

For the tag generation queries from $\mathcal{A}$ of the form (k, f), if $k \neq i$, the reduction retrieves the secret key from SK and generates the tag as specified in Figure 9. If additionally, $f = 1$, the reduction reveals the tag tag. If $k = i$, the reduction retrieves $(\mathsf{PK}[i], G^r, r)$ from $\mathcal{L}_{\mathsf{H_P}}$ and adds $(G^a)^r$ to $\mathsf{Wallet}[k]$ and return $(G^a)^r$ if $f = 1$.

For the spend oracle queries made by $\mathcal{A}$ of the form $(\{j, k\}, \mathcal{R}, \mathcal{O}, \mu)$, where $|\mathcal{R}| = |\mathcal{R}|$ and (w.l.o.g.) $j = |\mathcal{R}|$, if $k = i$, then $\mathcal{S}$ does the following:

- Parse $\mathcal{R} := (pk_1, \ldots, pk_{|\mathcal{R}|})$, notice that $pk_{|\mathcal{R}|} = G^a$
- Retrieve $(G^a)^r$ from $\mathsf{Wallet}[k]$

- Retrieve $(\mathsf{PK}[k], G^r, r)$ from $\mathcal{L}_{\mathsf{H_P}}$
- Set $tx := \{\mathcal{R}, (G^a)^r, \mathcal{O}, \mu\}$
- Sample $h_{|\mathcal{R}|}, s_{|\mathcal{R}|} \leftarrow \mathbb{Z}_q^*$ and set $h_0 = h_{|\mathcal{R}|}$, for $j \in [|\mathcal{R}| - 1]$,
  - Check if $(pk_j, G^{e_j}) \in \mathcal{L}_{\mathsf{H_P}}$, and if not set $\mathsf{H_P}(pk_j) := G^{e_j}$ for random $e_j \leftarrow \mathbb{Z}_q^*$
  - Sample $s_j, h_j \leftarrow \mathbb{Z}_q^*$ and set $\mathsf{H_S}(tx^*||G^{s_j} pk_j^{h_{j-1}}||G^{e_j s_j}(G^{ch_{j-1}})) = h_j$
- Set $\mathsf{H_S}(tx||G^{s_{|\mathcal{R}|}}(G^a)^{h_{|\mathcal{R}|-1}}||(G^r)^{s_{|\mathcal{R}|}}(G^{ar})^{h_{|\mathcal{R}|-1}}) = h_{|\mathcal{R}|}$
- Set $\sigma := (s_{|\mathcal{R}|}, s_1, \ldots, s_{|\mathcal{R}|-1}, h_0)$ and return (tx, σ) to $\mathcal{A}$.

For spend oracle queries where $k \neq i$, the reduction simulates as described in Figure 9 checking for query response for other oracles namely, $\mathsf{H_P}$ and $\mathsf{H_S}$. The reduction sets new responses for the oracles if any query was not made previously.

Finally, $\mathcal{A}$ outputs its slander (tx^*, σ^*). The reduction aborts the execution if it is not a valid signature, or if the pair was previously obtained through a spend oracle query. Let $tx^* := \{\mathcal{R}, tag, \mathcal{O}, \mu\}$ and the reduction also aborts if $tag \neq (G^a)^r$.

Operations of $\mathcal{S}$ Note that $\mathsf{H_S}$ and $\mathsf{H_P}$ queries are made by $\mathcal{S}$ during the verification process of the slander. To be more precise, a total of $|\mathcal{R}|$ number of '$\mathsf{H_S}$ operations' are needed to verify the signature (in this case the slander). We therefore have two events. Firstly, the event $\mathbf{E}$ where each of the $|\mathcal{R}|$ queries corresponding to the $|\mathcal{R}|$ verification queries (to verify the slander) have already been included in the $q_{\mathsf{H_S}}$ number of hash queries (made by $\mathcal{A}$) or in the q_S number of spending oracle queries. Secondly, the event $\neg\mathbf{E}$ which denotes the complement of the above event, where $\mathcal{S}$ aborts the execution by outputting $\mathtt{abort}_1$.

If $\mathbf{E}$ happens, consider the set of $|\mathcal{R}|$ queries made by $\mathcal{A}$ to $\mathsf{H_S}$ that match the $|\mathcal{R}|$ queries made in the verification process ($|\mathcal{R}|$ queries per slander) by $\mathcal{S}$. We let $X_{i_1}, X_{i_2}, \ldots, X_{i_{|\mathcal{R}|}}$ denote the first appearance on the transcript T of each of the queries to $\mathsf{H_S}$ used by $\mathcal{S}$ for verification of the slander where $1 \leq i_1 \leq \ldots \leq i_{|\mathcal{R}|}$. (This is to consider the case of repetition of queries)

Let ℓ be such that

$$X_{i_{|\mathcal{R}|}} = \mathsf{H_S}(tx^*||G^{s_\ell} pk_\ell^{h_{\ell-1}}||(G^{r_\ell})^{s_\ell}(tag)^{h_{\ell-1}})$$

in the verification process by $\mathcal{S}$. We call this ℓ as the gap of σ^*.

We annotate a successful slander σ^* by $\mathcal{S}$ as a (w, ℓ)-slander if $i_1 = w$ i.e, the first appearance of all verification related queries is the w-th query to the $\mathsf{H_S}$ oracle and ℓ is the gap. Queries made during the spending oracle simulation to the random oracles are counted. $\mathcal{S}$ aborts by outputting $\mathtt{abort}_2$ if $tag \neq (G^a)^r$ and continues to perform a rewind simulation otherwise.

$\mathcal{S}$ has recorded the transcript T for the (w, ℓ) slander output by $\mathcal{A}$. Given this successful (w, ℓ) slander, $\mathcal{S}$ now rewinds the transcript T to the w-th query and gives it to $\mathcal{A}$ as a rewind simulation to obtain another transcript T' which is a successful (w, ℓ) slander again. If T' is not a successful (w, ℓ) slander again then $\mathcal{S}$ aborts by outputting $\mathtt{abort}_3$, and if not, it continues. New coin flips that are independent to T are made for all queries subsequent to the w-th query (where w is learnt from the (w, ℓ) slander from T). Note that consistency is maintained with the previous queries and also that both T and T' use the same program in $\mathcal{A}$.

Let the w-th query common to T and T' be denoted by $\mathsf{H_S}(tx^*, G^u, G^v)$.

Here, $\mathcal{S}$ knows G^u and G^v but not u and v at the time of rewind. After $\mathcal{A}$ returns the output from the rewind simulation, $\mathcal{S}$ proceeds to compute the discrete log a of G^a.

The transcript T and the rewind simulation transcript T' contain two (w, ℓ)-slander signatures such that the following pairs of equalities hold.

$$\begin{aligned} G^u &= G^{s_\ell} pk_\ell^{h_{\ell-1}} = G^{s_\ell + x_\ell h_{\ell-1}} \\ G^u &= G^{s'_\ell} pk_\ell^{h'_{\ell-1}} = G^{s'_\ell + x_\ell h'_{\ell-1}} \end{aligned} \tag{2}$$

$$\begin{aligned} G^v &= G^{r_\ell s_\ell}(G^{ar})^{h_\ell} = G^{r_\ell s_\pi + arh_{\ell-1}} \\ G^v &= G^{r_\ell s'_\ell}(G^{ar})^{h'_{\ell-1}} = G^{r_\ell s'_\ell + arh'_{\ell-1}} \end{aligned} \tag{3}$$

Notice that in equation (1) the public key pk_ℓ may be of adversary's choice. Therefore x_ℓ is not known to the reduction. Therefore the reduction uses the equation (2) to solve the DL problem and retrieve a. As the reduction knows $r_\ell, s_\ell, s'_\ell, r, h_{\ell-1}, h'_{\ell-1}$, it can do the following:

$$a = \frac{r_\ell s_\ell - r_\ell s'_\ell}{r h'_{\ell-1} - r h_{\ell-1}} \mod q$$

Analysis We first analyze the probability of $\mathcal{S}$ outputting $\texttt{abort}_1$. We can see that in the case of event $\neg\mathbf{E}$ the conditional probability of h_0 in the forged signature σ^* satisfying the final equation in the verification process is at most $\frac{1}{t-q_H-|\mathcal{R}|q_S}$ (where t denotes all possible hash values for an input) which is negligible. For the given adversary $\mathcal{A}$ we have

$$\begin{aligned}\epsilon(\lambda) &< \Pr[\mathbf{E}]\Pr[\mathcal{A} \text{ slanders}|\mathbf{E}] + \Pr[\neg\mathbf{E}]\Pr[\mathcal{A} \text{ slanders}|\neg\mathbf{E}] \\ &\leq \Pr[\mathbf{E}]\Pr[\mathcal{A} \text{ slanders}|\mathbf{E}] + 1\left(\frac{1}{t-q_H-|\mathcal{R}|q_S}\right)\end{aligned}$$

The probability of $\mathcal{A}$ returning a slander and having already queried for all $|\mathcal{R}|$ queries used in verification is greater than $\epsilon(\lambda)$ as $\left(\frac{1}{t-q_H-|\mathcal{R}|q_S}\right)$ is negligible.

For analyzing the probabilities $\Pr[\neg\texttt{abort}_2]$ and $\Pr[\neg\texttt{abort}_3]$, we do the following. We refer to the run of $\mathcal{S}$ resulting in transcript T as the first and T' as the second. The probability of a slander in the first run is $\left(\frac{1}{q_K}\cdot\epsilon(\lambda)\right)$. This is because the $\Pr[\neg\texttt{abort}_2]$ which is the probability of $\mathcal{S}$ correctly guessing the slander tag is equal to $\left(\frac{1}{q_K}\right)$. We can compute $\Pr[\neg\texttt{abort}_3]$ which is the probability of the second run of $\mathcal{S}$ also resulting in a (w,ℓ) slander as $\left(\frac{\epsilon(\lambda)}{q_K(q_H+q_Kq_S)}\right)$.

To bound the success of reduction $\mathcal{S}$ we refer to the forking lemma proposed by Bellare et al. [57]. By the forking lemma we have that $\mathcal{S}$ solves the DL problem with probability

$$\Pr[\mathcal{S} \text{ succeeds}] \geq \frac{1}{q_K}\epsilon(\lambda)\cdot\left(\frac{\epsilon(\lambda)}{q_K(q_H+q_Kq_S)} - \frac{1}{h}\right).$$

Here $\frac{1}{h}$ refers to the probability with which the randomness used in the second run is the same as that in the first run and this is negligible.

We can see that the complexity of $\mathcal{S}$ is no more than $q_K(q_H+q_Kq_S)$ times that of $\mathcal{A}$ and the probability of success of $\mathcal{S}$ against the DL problem is at least $\frac{1}{q_H+q_Kq_S}\cdot\left(\frac{\epsilon(\lambda)}{q_K}\right)^2$ which is non-negligible. Therefore we arrive at the contradiction, thereby proving non-slanderability.

□

A.4.3 Linkability

Proof. We first argue that the tag generated in the LRS construction is binding to the secret key. Consider an adversary that outputs $(pk, sk_1, sk_2, tag_1, tag_2)$ such that $\mathsf{ChkTag}(pk, sk_i, tag_i) = 1$ for $i \in [2]$.

The ChkTag procedure for the LRS construction of Monero checks the following:

- $pk = G^{sk_i}$
- $tag_i = \mathsf{H_P}(pk)^{sk_i}$

We therefore have

$$pk = G^{sk_1} = G^{sk_2}$$

which implies $sk_1 = sk_2$. Since $\mathsf{H_P}$ is modeled as a random oracle, it is deterministic and therefore we have

$$\mathsf{H_P}(pk) = \mathsf{H_P}(G^{sk_1}) = \mathsf{H_P}(G^{sk_2})$$

$$tag_1 = \mathsf{H_P}(G^{sk_1})^{sk_1} = \mathsf{H_P}(G^{sk_2})^{sk_1} = \mathsf{H_P}(G^{sk_2})^{sk_2} = tag_2$$

We now have $(sk_1, tag_1) = (sk_2, tag_2)$. Therefore we conclude that the tag in the LRS construction of Monero is indeed binding.

We now proceed to the $\mathsf{LinkExp}_{\mathsf{LRS},\mathcal{A},\mathcal{E}_\mathcal{A}}(1^\lambda, 1^\alpha)$ experiment. If the adversary $\mathcal{A}$ wins the experiment it has to output a tuple (tx, σ) such that the signature is valid and the tag is not consistent with the secret key and public key used for generating the signature.

We construct the extractor $\mathcal{E}_\mathcal{A}$ similar to the technique used in the non-slanderability proof. Note that now the adversary $\mathcal{A}$ only queries for the random oracles $\mathsf{H_S}$ and $\mathsf{H_P}$ (Figure 9). The extractor picks random coins and sets the oracle responses for both $\mathsf{H_P}$ and $\mathsf{H_S}$. In both cases, the extractor locally maintains a list of query-response pair. Let the total number of queries made to the oracles be denoted by $q_{\mathsf{H_P}}$ and $q_{\mathsf{H_S}}$ respectively.

When $\mathcal{A}$ outputs a tuple (tx, σ) where $tx := \left\{\{pk_i^\mathcal{R}\}_{i\in[|\mathcal{R}|]}, tag, \mathcal{O}, \mu\right\}$ and $\sigma := (s_0, s_1, \ldots, s_{|\mathcal{R}|-1}, h_0)$, the extractor verifies if the signature is valid. During this signature verification, the extractor makes its own queries to the random oracles. Similar to what we saw in the non-slanderability proof, we have the event $\mathbf{E}$ where each of the $|\mathcal{R}|$ queries corresponding to the $|\mathcal{R}|$ verification queries (to verify the signature) have already been included in the $q_{\mathsf{H_S}}$ number of hash queries (made by $\mathcal{A}$). Secondly, the event $\neg\mathbf{E}$ which denotes the complement of the above event, where $\mathcal{E}_\mathcal{A}$ aborts the execution by outputting $\mathtt{abort}_1$.

If $\mathbf{E}$ happens, consider the set of $|\mathcal{R}|$ queries made by $\mathcal{A}$ to $\mathsf{H_S}$ that match the $|\mathcal{R}|$ queries made in the verification process by $\mathcal{E}_\mathcal{A}$. We let $X_{i_1}, X_{i_2}, \ldots, X_{i_{|\mathcal{R}|}}$ denote the first appearance on the transcript T of each of the queries to $\mathsf{H_S}$ used by $\mathcal{E}_\mathcal{A}$ for verification of the signature where $1 \le i_1 \le \ldots \le i_{|\mathcal{R}|}$. (This is to consider the case of repetition of queries)

Let ℓ be such that

$$X_{i_{|\mathcal{R}|}} = \mathsf{H_S}(tx||G^{s_\ell} {pk_\ell}^{h_{\ell-1}}||(G^{r_\ell})^{s_\ell}(tag)^{h_{\ell-1}})$$

in the verification process by $\mathcal{E}_\mathcal{A}$. We call this ℓ as the gap of σ.

We annotate a successful σ by $\mathcal{A}$ as a (w, ℓ)-signature if $i_1 = w$ i.e, the first appearance of all verification related queries is the w-th query to the $\mathsf{H_S}$ oracle and ℓ is the gap.

$\mathcal{E}_\mathcal{A}$ has recorded the transcript T for the (w, ℓ)-signature output by $\mathcal{A}$. Given this successful (w, ℓ)-signature, $\mathcal{E}_\mathcal{A}$ now rewinds the transcript T to the w-th query and gives it to $\mathcal{A}$ as a rewind simulation to obtain another transcript T' which is a successful (w, ℓ)-signature again. The second such successful signature for tx is denoted by $\sigma' := (s'_0, s'_1, \ldots, s'_{|\mathcal{R}|-1}, h'_0)$.

New coin flips that are independent to T are made for all queries subsequent to the w-th query (where w is learnt from the (w, ℓ)-signature from T). Note that consistency is maintained with the previous queries and also that both T and T' use the same program in $\mathcal{A}$. If T' is not a successful (w, ℓ)-signature again then $\mathcal{E}_\mathcal{A}$ aborts by outputting $\mathtt{abort}_3$, and if not, it continues.

Let the w-th query common to T and T' be denoted by $\mathsf{H_S}(tx, G^u, G^v)$.

Here, $\mathcal{E}_\mathcal{A}$ knows G^u and G^v but not u and v at the time of rewind. The transcript T and the rewind simulation transcript T' contain two (w, ℓ)-signatures such that the following pairs of equalities hold, where $(G^{r_\ell})^y = tag$.

$$\begin{aligned} G^u &= G^{s_\ell} {pk_\ell}^{h_{\ell-1}} = G^{s_\ell + x_\ell h_{\ell-1}} \\ G^u &= G^{s'_\ell} {pk_\ell}^{h'_{\ell-1}} = G^{s'_\ell + x_\ell h'_{\ell-1}} \end{aligned} \tag{4}$$

$$\begin{aligned} G^v &= G^{r_\ell s_\ell}(G^{r_\ell})^{y h_\ell} = G^{r_\ell s_\pi + y r_\ell h_{\ell-1}} \\ G^v &= G^{r_\ell s'_\ell}(G^{r_\ell})^{y h'_{\ell-1}} = G^{r_\ell s'_\ell + r_\ell y h'_{\ell-1}} \end{aligned} \tag{5}$$

Notice that in the first equation the public key pk_ℓ is of adversary's choice. Since the extractor knows $s_\ell, s'_\ell, h_{\ell-1}, h'_{\ell-1}$, it retrieves the secret key by doing the following:

$$x_\ell := \frac{s_\ell - s'_\ell}{h'_{\ell-1} - h_{\ell-1}} \mod q$$

The extractor $\mathcal{E}_\mathcal{A}$ outputs $(\{pk_i\}_{i\in[|\mathcal{R}|]}, (\ell, x_\ell, tag), \mathcal{O}, \mu)$. For the adversary to win the $\mathsf{LinkExp}_{\mathsf{LRS},\mathcal{A},\mathcal{E}_\mathcal{A}}(1^\lambda, 1^\alpha)$ experiment it has to be that $\mathsf{ChkTag}(pk_\ell, x_\ell, tag) = 0$. Since we have $pk_\ell := G^{x_\ell}$ it must be the case that $tag \neq \mathsf{H_P}(pk_\ell)^{x_\ell}$.

Notice that using equation (2) we can do the following:

$$y := \frac{r_\ell s_\ell - r_\ell s'_\ell}{r_\ell h'_{\ell-1} - r_\ell h_{\ell-1}} \mod q$$

$$y := \frac{s_\ell - s'_\ell}{h'_{\ell-1} - h_{\ell-1}} \mod q$$

$$y := x_\ell$$

Since $tag \neq \mathsf{H_P}(pk_\ell)^{x_\ell}$, we have $(G^{r_\ell})^y = \mathsf{H_P}(pk_\ell)^y \neq \mathsf{H_P}(pk_\ell)^{x_\ell}$. This implies that we have $\mathsf{ChkTag}(pk_\ell, y, tag) = 1$, $\mathsf{ChkTag}(pk_\ell, x_\ell, \mathsf{H_P}(pk_\ell)^{x_\ell}) = 1$ and $(y, tag) \neq (x_\ell, \mathsf{H_P}(pk_\ell)^{x_\ell})$ which is contradiction to the binding property of the tag generation in the LRS construction of Monero.

Therefore we have that $\mathsf{LinkExp}_{\mathsf{LRS},\mathcal{A},\mathcal{E}_\mathcal{A}}(1^\lambda, 1^\alpha) \leq \mathsf{negl}(\lambda)$.

The analysis of the probability of success of $\mathcal{E}_\mathcal{A}$ is similar to what we saw in the non-slanderability proof except now there are no spending oracle and key generation oracle queries to account for.

We have thus showed that LRS construction in Monero satisfies the two notions required for linkability and this concludes the proof.

□

B Formal Definition of Payment Channel

Definition B.1 (Payment Channel [4]). *A PC is defined with respect to a blockchain $\mathbb{B}$ and is equipped with three operations* (OpChannel, ClChannel, Pay) *described below:*

$\{0,1\} \leftarrow \mathsf{OpChannel}(u_1, u_2, \beta, t)$*: On input two users u_1, u_2, an initial channel capacity β, and a timeout t, if the operation is authorized by u_1, and u_1 owns at least β coins,* OpChannel *creates a new payment channel $(c_{\langle u_1,u_2\rangle}, \beta, t)$, where $c_{\langle u_1,u_2\rangle}$ is a fresh channel identifier and adds to a list $\mathcal{L}$. Then it uploads it to $\mathbb{B}$ and returns* 1*. Otherwise, it returns* 0*.*

$\{0,1\} \leftarrow \mathsf{ClChannel}(c_{\langle u_1,u_2\rangle}, v)$*: On input a channel identifier $c_{\langle u_1,u_2\rangle}$ and a balance v (i.e., the distribution of coins locked in the channel between u_1 and u_2), if the operation is authorized by both u_1 and u_2,* ClChannel *removes the corresponding channel from $\mathcal{L}$, includes the balance v in $\mathbb{B}$ and returns* 1*. Otherwise, it returns* 0*.*

$\{0,1\} \leftarrow \mathsf{Pay}(c_{\langle u_1,u_2\rangle}, v)$*: On input a channel identifier $c_{\langle u_1,u_2\rangle}$ and a payment value v, and if the payment channel has at least a current balance $\gamma \geq v$, the pay operation decreases the current balance of the payment channel by v and returns* 1*. In any other case,* Pay *returns* 0*.*

C Assumptions

Definition C.1 (Discrete Logarithm (DL) Assumption). *We say that the discrete logarithm assumption holds over $\mathcal{G}$ if for every* PPT *adversary $\mathcal{A}$*

$$\Pr[\mathsf{DL}_\mathcal{A}(\mathcal{G}) = 1] \leq \mathsf{negl}(\lambda)\,,$$

where the game $\mathsf{DL}_\mathcal{A}(\mathcal{G})$ is defined in Figure 13.

$\underline{\mathsf{DL}_\mathcal{A}(\mathcal{G})}$

$x \leftarrow\!\!\$\ \mathbb{Z}_q$

$x' \leftarrow \mathcal{A}(\mathbb{G}, q, G, G^x)$

$b := (G^x = G^{x'})$

return b

Figure 13: Security Game for Discrete Logarithm

Definition C.2 (Decisional Diffie-Hellman (DDH))**.** *We say that the DDH assumption according to holds over $\mathcal{G}$ if for all $\ell \in \mathsf{poly}(\lambda)$, every* PPT *adversary $\mathcal{A}$*

$$\left| \Pr\left[\mathsf{DDH}^0_{\mathcal{A}}(\mathcal{G}) = 1\right] - \Pr\left[\mathsf{DDH}^1_{\mathcal{A}}(\mathcal{G}) = 1\right] \right| \leq \mathsf{negl}(\lambda)\,,$$

where the game $\mathsf{DDH}_{\mathcal{A}}(\mathcal{G})$ is defined in Figure 14.

$\mathsf{DDH}^b_{\mathcal{A}}(\mathcal{G})$

$a, b, c \leftarrow_{\$} \mathbb{Z}_q^*$
$b' \leftarrow_{\$} \{0,1\}$
$Z_0 = G^{ab}$
$Z_1 = G^c$
$b'' \leftarrow \mathcal{A}(\mathbb{G}, q, G, G^a, G^b, Z_{b'})$
$b_0 := (b' = b'')$
return b_0

Figure 14: Security Game for DDH

D Security Analysis of VTLRS For Transaction Scheme of Monero

We now give the security analysis of Theorem 5.5.

Timed Privacy. We show that the protocol (Figure 3) is private against an adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ where $\mathcal{A}_1$ is PPT and $\mathcal{A}_2$ is of depth bounded by $\mathbf{T}^\epsilon$, for some non-negative $\epsilon < 1$. We now gradually change the simulation through a series of hybrids and then we argue about the proximity of neighbouring experiments. The following hybrids only deal with the if-else portion of the experiment where if $b = 0$ we have the honest prover generating C, π, while if $b = 1$ we have the simulator generating the same without access to the $\mathcal{I}$ or σ.

Hybrid $\mathcal{H}_0$: This is the original execution corresponding to $b = 0$.

Hybrid $\mathcal{H}_1$: This is identical to the previous hybrid except that the random oracle H' is simulated by lazy sampling. In addition a random set I^* (where $|I^*| = t - 1$) is sampled ahead of time, and the output of the random oracle on the cut-and-choose instance is programmed to I^*. Note that the changes of this hybrid are only syntactical and therefore the distribution is unchanged.

Hybrid $\mathcal{H}_2$: This is identical to the previous hybrid except that the random oracle $\mathsf{H_S}$ and $\mathsf{H_P}$ are simulated by lazy sampling. As before, the changes of the hybrids are only syntactical and therefore the distribution is unchanged.

Hybrid $\mathcal{H}_3$: In this hybrid we sample a simulated common reference string crs_{rng}. By the zero-knowledge property of $(\mathsf{Setup}_{\mathsf{NIZK},\mathcal{L}_{\mathsf{rng}}}, \mathcal{P}_{\mathsf{NIZK},\mathcal{L}_{\mathsf{rng}}}, \mathcal{V}_{\mathsf{NIZK},\mathcal{L}_{\mathsf{rng}}})$ this change is computationally indistinguishable.

Hybrid $\mathcal{H}_4 \ldots \mathcal{H}_{4+n}$: In the hybrid $\mathcal{H}_{4+i}$, for all $i \in [n]$, the proof $\pi_{\mathsf{rng},i}$ is computed via the simulator provided by the underlying NIZK proof. By the zero-knowledge property of $(\mathsf{Setup}_{\mathsf{NIZK},\mathcal{L}_{\mathsf{rng}}}, \mathcal{P}_{\mathsf{NIZK},\mathcal{L}_{\mathsf{rng}}}, \mathcal{V}_{\mathsf{NIZK},\mathcal{L}_{\mathsf{rng}}})$, the distance between neighboring hybrids is bounded by a negligible function in the security parameter.

Hybrid $\mathcal{H}_{4+n} \ldots \mathcal{H}_{4+2n-t+1}$: In the i-th hybrid $\mathcal{H}_{4+i}$, for all $i \in [n - (t-1)]$, the puzzle corresponding to the i-th element of the set $\bar{I}^*$ is computed as $\mathsf{LHTLP.PGen}(\mathsf{p}, 0^\lambda; r_i)$, where $\bar{I}^*$ is the complement of I^*. Since the distinguisher is depth-bounded, indistinguishability follows from an invocation of the security of LHTLP.

Hybrid $\mathcal{H}_{4+2n-t+2}$: In this hybrid the prover behaves as follows.

parse $tx := \left(\left\{pk_i^{\mathcal{R}}\right\}_{i=1}^{|\mathcal{R}|}, tag, \left\{pk_i^{\mathcal{O}}\right\}_{i=1}^{|\mathcal{O}|}, \mu\right)$

$(s_1, \ldots, s_{|\mathcal{R}|-1}) \leftarrow \mathbb{Z}_q^*$

$(h_0, h_1, \ldots, h_{|\mathcal{R}|-1}) \leftarrow \mathbb{Z}_q^*$

$r_L, r_R \leftarrow \mathbb{Z}_q^*$

set $L_{|\mathcal{R}|} := G^{r_L}, R_{|\mathcal{R}|} := G^{r_R}$

for $i \in [|\mathcal{R}| - 1]$ **do**

$L_i := G^{s_i} pk_i^{h_{i-1}}$

$R_i := \mathsf{H}_{\mathsf{P}}(pk_i)^{s_i} tag^{h_{i-1}}$

set $\mathsf{H}_{\mathsf{S}}(tx||L_i||R_i) := h_i$

endfor

set $\mathsf{H}_{\mathsf{S}}(tx||L_{|\mathcal{R}|}||R_{|\mathcal{R}|}) := h_0$

set $\tilde{G} := \dfrac{L_{|\mathcal{R}|}}{pk_{|\mathcal{R}|}^{h_{|\mathcal{R}|-1}}}$

set $\tilde{H} := \dfrac{R_{|\mathcal{R}|}}{tag^{h_{|\mathcal{R}|-1}}}$

The prover then samples $\forall i \in I^*$ $\alpha_i \leftarrow \mathbb{Z}_q^*, K_i := G^{\alpha_i}, Y_i := \mathsf{H}_{\mathsf{P}}(pk_{|\mathcal{R}|})^{\alpha_i}$ and computes the puzzles for these indices as described in the protocol. On the other hand, for all $i \notin I^*$ it computes

$$K_i = \left(\frac{\tilde{G}}{\prod_{j \in I^*} K_j^{\ell_j(0)}}\right)^{\ell_i(0)^{-1}}$$

$$Y_i = \left(\frac{\tilde{H}}{\prod_{j \in I^*} Y_j^{\ell_j(0)}}\right)^{\ell_i(0)^{-1}}.$$

The rest of the committing algorithm is unchanged. Specifically, the commitment C is set as $C := (\tilde{G}, \tilde{H}, \{s_i\}_{i \in [|\mathcal{R}|-1]}, h_0, \{Z_i\}_{i \in [n]}, \mathbf{T})$ and the proof $\pi := (\{K_i, Y_i, \pi_{\mathsf{rng},i}\}_{i \in [n]}, I^*, \{\alpha_i, r_i\}_{i \in I^*})$ Note that for all $i \notin I^*$, we have

$$\left(K_i^{\ell_j(0)} \cdot \prod_{j \in I^*} K_j^{\ell_j(0)} = \tilde{G}\right)$$

$$\left(Y_i^{\ell_j(0)} \cdot \prod_{j \in I^*} Y_j^{\ell_j(0)} = \tilde{H}\right).$$

Furthermore, observe that the distribution induced by the terms $(\tilde{G}, \tilde{H}\{s_i\}_{i \in [|\mathcal{R}|-1]}, h_0)$ is identical to that of the previous hybrid. It follows that the changes here are only syntactical and the distribution induced by this hybrid is identical to that of the previous one.

Simulator $\mathcal{S}$: The simulator is defined to be identical to the last hybrid. Note that no information about the signing key sk or the signature σ is used to compute the proof. This is identical to the real execution when $b = 1$. Therefore we have successfully switched hybrids in an indistinguishable manner to arrive at the execution where $b = 1$. This concludes our proof. □

We now show that our protocol (Figure 3) is sound and the proof of Theorem 5.6.

Soundness. We analyze the protocol in its interactive version and the soundness of non-interactive protocol follows from the Fiat-Shamir transformation [39] for constant-round protocols. Let $\mathcal{A}$ be an adversary that efficiently breaks the soundness of the protocol. In particular this means that the adversary produces

a commitment $(G^{s_0}, \mathsf{H_P}(pk_{|\mathcal{R}|})^{s_0}, \{s_i\}_{i\in[|\mathcal{R}|-]}, h_0, Z_1, \ldots, Z_n, \mathbf{T})$ such that for all $Z_i \notin I$ it holds that $\mathsf{LHTLP.PSolve}(\mathsf{p}, Z_i) = \tilde{\alpha}_i$ such that

$$\forall i \notin I \; K_i \neq G^{\alpha_i}$$

or if,

$$\forall i \notin I \; Y_i \neq \mathsf{H_P}(pk_|\mathcal{R}|)^{\alpha_i}$$

Assume the contrary, then we could recover a valid signature on tx by interpolating $\tilde{\alpha}_i$ with $\{\sigma_i\}_{i\in I}$, which satisfy the above relation by definition of the verification algorithm. Further observe that all puzzles $(Z_1, \ldots, Z_n)$ are well-formed, i.e., the solving algorithm always outputs some well-defined value, except with negligible probability, by the soundness of the range NIZK.

It follows that, given $(Z_1, \ldots, Z_n)$ we can recover some set I' in polynomial time by solving the puzzles and checking which of the α's satisfy the above relation. In order for the verifier to accept, it must be the case that $I' = I$ which means that the prover correctly guesses a random n-bit string uniformly chosen from the set of strings with exactly $n/2$-many 0's. This happens with probability exactly $\frac{(n/2!)^2}{n!}$.

Observe that, in the non-interactive variant of the protocol, the above argument holds even in the presence of an arbitrary (polynomial) number of simulated proofs, as long as the range NIZK is simulation-sound. Therefore, if we instantiate the range NIZK with a simulation-sound scheme, then so is the resulting VTLRS-TS. □

Privacy, Non-Slanderability and Linkability. We require that VTLRS inherits privacy, non-slanderability, and linkability from its LRS transaction scheme. These notions are immediately satisfied by our modified scheme because the commit algorithm takes no secret information about the spender. Instead it takes the transaction and the signature and generates a commitment and a proof. Therefore, these operations can be performed by the LRS adversary itself and thus it gains no additional information.

E Range Proofs for HTLPs

In this Section we describe a protocol from [30] which allows a prover to convince a verifier in zero-knowledge that a list of linearly homomorphic time-lock puzzles are well formed. This allows us to homomorphically pack them into a single time-lock puzzle.

The construction requires a linearly homomorphic time-lock puzzle which is also homomorphic in the random coins. The construction of [40] satisfies this property.

We assume that plaintexts in a ring $\mathbb{Z}_q$ are represented via the central representation in $[-q/2, q/2]$.

The protocol ensures that every plaintext is in the interval $[-L, L]$, given that $2L$ is smaller than the modulus of the plaintext space. This protocol can be readily used to prove that plaintexts are in a non-centered interval $[a, b]$ via homomorphically shifting plaintexts by $-(a+b)/2$, mapping the interval $[a, b]$ to $[-(b-a)/2, (b-a)/2]$. Consequently, for the sake of simplicity we will only discuss the case of centered intervals. Formal analysis of soundness and zero-knowledge of the protocol can be found in [30].

F Joint Key Generation And Joint Spending Protocols

In Figure 16 we describe our protocol for a joint generation of a public key and a tag. And in Figure 17 we present our protocol for the joint signature of a transaction. Together they realise the ideal functionality $\mathcal{F}_{\mathsf{J-LRS}}$.

In the formal description of the interactive protocols, we define the hash function $\mathsf{H} : \{0,1\}^* \to \{0,1\}^\lambda$, that is used to commit different values at different stages. We also make use of a NIZK proof system $(\mathcal{P}_{\mathsf{NIZK}}, \mathcal{V}_{\mathsf{NIZK}})$ for the language $\mathcal{L}_{\mathsf{eqdl}}$ is defined as follows [56]:

$$\mathcal{L}_{\mathsf{eqdl}} := \left\{ \begin{matrix} stmt = (Y, X, K) : \exists wit = y \text{ s.t.} \\ (Y = G^y \;\; \wedge X = K^y) \end{matrix} \right\}$$

Proving the security of Figures 16 and 17 means that we have to show they securely realize the functionality $\mathcal{F}_{\mathsf{J-LRS}}$. The formal statement is given in Theorem F.1. In the security analysis of the theorem the hash functions $\mathsf{H}, \mathsf{H_P}, \mathsf{H_P}$ are modelled as random oracles, where $\mathsf{H_P}$ and $\mathsf{H_S}$ are from Section 4.2.

Setup: An RSA modulus N, public parameters pp for LHTLP, interval parameters L and B with $B < L$. In this protocol we use k as a statistical security parameter.
Common input: time-lock puzzles $Z_1, \ldots, Z_\ell$.
Prover: On input wit, where $wit := ((\alpha_1, r_1), \ldots, (\alpha_\ell, r_\ell))$ and $\alpha_i \in [-B, B]$ such that for all i it holds $Z_i \leftarrow \mathsf{LHTLP.PGen}(\mathsf{pp}, \alpha_i; r_i)$, the prover algorithm $\mathcal{P}$ does the following.

- Choose $y_1, \ldots, y_k \leftarrow [-L/4, L/4]$ and random coins $r'_1, \ldots, r'_k$ from their corresponding ring.
- For $i = 1, \ldots, k$ compute $D_i \leftarrow \mathsf{LHTLP.PGen}(\mathsf{pp}, y_i; r'_i)$
- Compute $(\mathbf{t}_1, \ldots, \mathbf{t}_k) \leftarrow H(Z_1, \ldots, Z_\ell, D_1, \ldots, D_k)$, where the $\mathbf{t}_i \in \{0,1\}^\ell$.
- For $i = 1, \ldots k$ compute $v_i \leftarrow y_i + \sum_{j=1}^{\ell} t_{i,j} \cdot \alpha_j$ and $w_i \leftarrow r'_i + \sum_{j=1}^{\ell} t_{i,j} \cdot r_j$
- Set $\pi \leftarrow (D_i, v_i, w_i)_{i \in [k]}$ and output π

Verifier: On input $\pi = (D_i, v_i, w_i)_{i \in [k]}$ then do the following.

- Compute $(\mathbf{t}_1, \ldots, \mathbf{t}_k) \leftarrow H(Z_1, \ldots, Z_\ell, D_1, \ldots, D_k)$
- For $i = 1, \ldots k$ check if $v_i \in [-L/2, L/2]$, compute $F_i \leftarrow D_i \cdot \prod_{j=1}^{\ell} Z_j^{t_{i,j}}$ and check if $F_i = \mathsf{LHTLP.PGen}(\mathsf{pp}, v_i; w_i)$.
- If all checks pass output 1, otherwise 0.

Figure 15: NIZK protocol for wellformedness of a vector of homomorphic time-lock puzzles

Theorem F.1. *Let $(\mathcal{P}_{\mathsf{NIZK},\mathcal{L}_{\mathsf{eqdl}}} \mathcal{V}_{\mathsf{NIZK},\mathcal{L}_{\mathsf{eqdl}}})$ be a simulation sound NIZK for the language $\mathcal{L}_{\mathsf{eqdl}}$. Then protocol described in Figure 16 and Figure 17 UC realize $\mathcal{F}_{\mathsf{J-LRS}}$ in the random oracle model.*

F.1 Security Analysis of $\mathcal{F}_{\mathsf{J-LRS}}$

Below we give the ideal functionalities for joint key generation and joint signing of transactions in Monero.

Proof. We consider the case where either parties are corrupted. We denote U_0 and U_1 by Alice and Bob, respectively.
Alice is corrupt: We describe a sequence of hybrid executions and argue that they are indistinguishable from each other.
Hybrid $\mathcal{H}_0$: is identical to the real protocol execution with an honest Bob.
Hybrid $\mathcal{H}_1$: is the same as the previous hybrid except now the simulator answers the adversary's random oracle H queries with random responses and stores the query response pair locally. This is otherwise referred to as lazy sampling. This change is purely syntactical and therefore this hybrid is identical to the previous one.
Hybrid $\mathcal{H}_2$: is identical to the previous hybrid except that all NIZK proofs produced by Bob are computed using the simulator and all proofs output by Alice are extracted using the extractor algorithm of the NIZK system. If the extraction fails, then Bob aborts. By the simulation extractability of the NIZK system, this event happens only with negligible probability and therefore the difference with respect to the previous hybrid is bounded by a negligible function.
Hybrid $\mathcal{H}_3$: is the same as the previous hybrid except that when the adversary sends h, the simulator checks if h was previously queried to some H oracle query. If not, the simulator aborts. If yes, it retrieves (G_1, h) where G_1 was the query. The simulator generates $G_2 := \frac{pk}{G_1}$, where pk is sampled by the ideal functionality. It responds to the adversary with G_2. If the adversary responds with G'_1 such that $G_1 \neq G'_1$ and $\mathsf{H}(G_1) = \mathsf{H}(G'_1)$, then the simulator aborts. The rest of the execution is the same as the previous hybrid. The only difference from the previous hybrid is when the simulator aborts when the adversary finds a collision on H. This happens with at most negligible probability and therefore the hybrids are indistinguishable.
Hybrid $\mathcal{H}_4$: is the same as the previous hybrid except now the simulator sends h' chosen randomly as the random oracle H's evaluation at some point to be determined later. It sets $\mathsf{H}_{\mathsf{P}}(pk) := G^r$ for some random r and hands it over to the adversary when it queries the random oracle on pk. The rest of the execution is unchanged. Note that the change here is only syntactical and therefore the view of the adversary is unchanged.

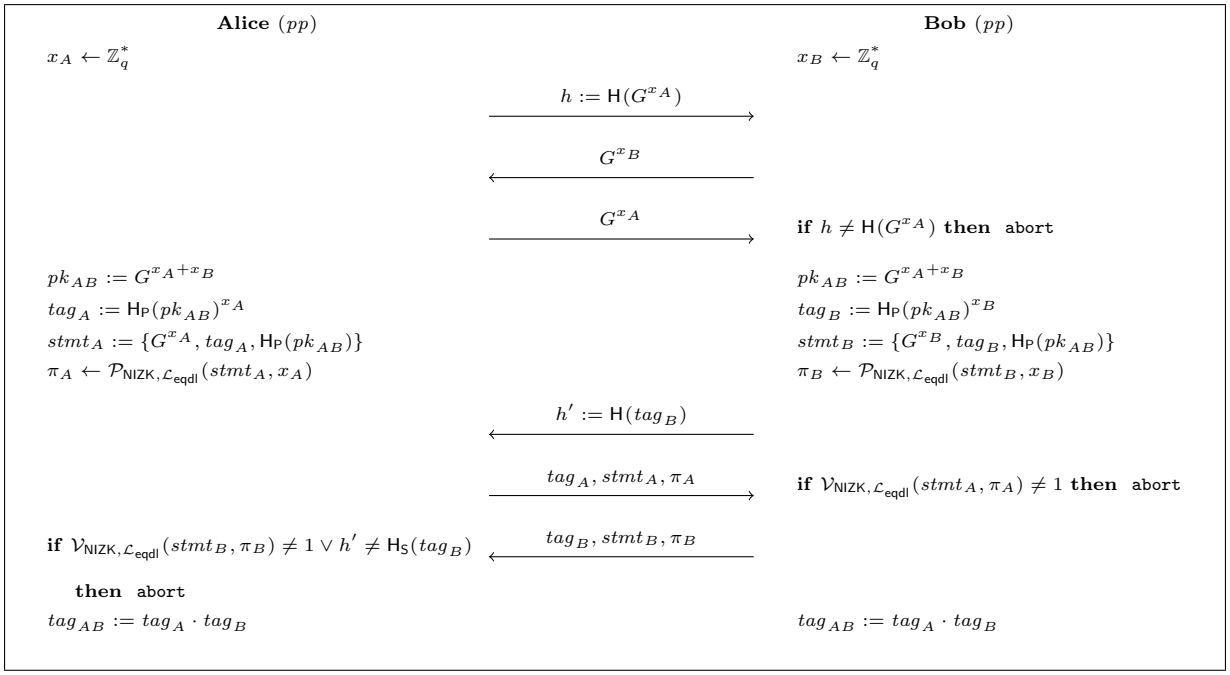

Figure 16: Joint Address and tag Generation

Hybrid $\mathcal{H}_5$: the simulator sets $tag_B := \frac{tag}{tag_A}$ and sets (tag_B, h') as the query response pair for H. Notice that when the adversary checks if $h' = \mathsf{H}(tag_B)$, the check is successful. Again this hybrid is identical to the previous one.

The simulator is defined as the previous hybrid. Notice that the simulator resembles (up to a negligible factor) the honest Bob for the adversary and succeeds in setting the joint key as pk and the joint tag as tag for the adversary.

Bob is corrupt: What is left is to analyse the case of corrupt Bob. We begin by describing a series of hybrid executions where we argue that each neighbouring hybrid executions are indistinguishable. The hybrids are given in the following.

Hybrid $\mathcal{H}_0$: the execution is the same as the real protocol.

Hybrid $\mathcal{H}_1$: is the same as for the case of a corrupted Alice.

Hybrid $\mathcal{H}_2$: is the same as for the case of a corrupted Alice.

Hybrid $\mathcal{H}_3$: is the same as the previous hybrid except that the simulator sends a random value h. When the adversary responds with G_1, the simulator samples sets $G_2 := \frac{pk}{G_1}$, where pk is sampled by the ideal functionality. It then sets (G_2, h) as the random oracle query response for H. It sends G_2 to the adversary. The rest of the execution is the same as the previous hybrid. The distribution is identical to that of the previous hybrid.

Hybrid $\mathcal{H}_4$: is the same as the previous hybrid, except now when the adversary sends h', the simulator checks for previous queries from the adversary of the form (T, h') to H. If no such query exists, the simulator aborts. Else, it sets $tag_A := \frac{tag}{T}$. The rest of the execution is unchanged. Note that this hybrid differs from the previous one only if the adversary finds a collision in the random oracle, which happens only with negligible probability.

Hybrid $\mathcal{H}_5$: the simulator as before checks if $\mathcal{V}_{\mathsf{NIZK},\mathcal{L}_{\mathsf{eqdl}}}(stmt_B, \pi_B) = 1$ where $stmt_B := (G_1, tag_B, G^r)$. If the check is successful, the simulator extracts a witness x_B such that it holds that $G_1 = G^{x_B}$ and $tag_B = G^{rx_B}$. Rest of the execution is as before. Notice that the only difference between the hybrids is that the simulator extracts a witness if the proof verifies successfully. This directly reduces to the simulation extractability of the NIZK proof and therefore the hybrids are indistinguishable.

Hybrid $\mathcal{H}_6$: the execution proceeds as above except now the simulators sends a random hash value h_A to the adversary. It additionally invokes $\mathsf{JSpend}(U_1, (s_1, \ldots, s_{|\mathcal{R}|-1}), pk_{U_0U_1}, tx)$ of $\mathcal{F}_{\mathsf{J-LRS}}$. The interface returns a signature $\sigma := (s_0, s_1, \ldots, h_0)$. The simulator completes the rest of the simulation by first

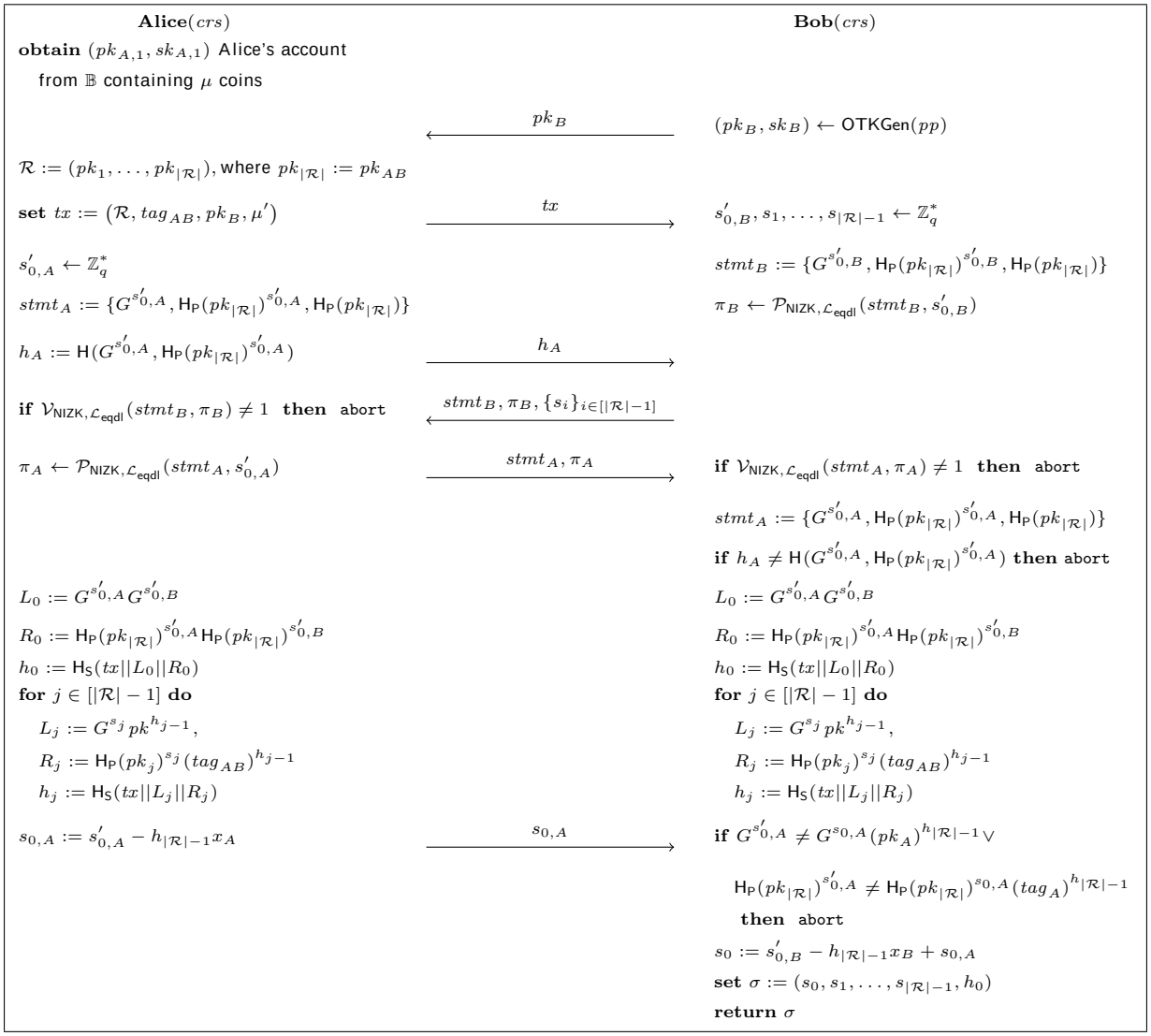

Figure 17: Joint Spending of Alice and Bob in Monero

computing

$$\begin{aligned}&\textbf{for } i \in [|\mathcal{R}| - 1] \textbf{ do}\\ &\quad L_i := G^{s_i} pk_i^{h_{i-1}}\\ &\quad R_i := \mathsf{H_P}(pk_i)^{s_i} tag^{h_{i-1}}\\ &\quad h_i := \mathsf{H_S}(tx||L_i||R_i)\end{aligned}$$

and $s_{0,B} := s'_{0,B} - h_{|\mathcal{R}|-1}x_B$ and $s_{0,A} := s_0 - s_{0,B}$, where $s'_{0,B}$ and x_B are extracted from the NIZKs produced by the adversary. It then computes $G^{s'_{0,A}} = G^{s_{0,A}} pk_A^{h_{|\mathcal{R}|-1}}$ and $\mathsf{H_P}(pk_{|\mathcal{R}|})^{s'_{0,A}} = \mathsf{H_P}(pk_{|\mathcal{R}|})^{s_{0,A}} tag_A^{h_{|\mathcal{R}|-1}}$ and sets the pre-image of h_A appropriately. Note that this change is only syntactical and does not change the view of the adversary.

The final simulator for the case of corrupted Bob is defined as execution in the previous hybrid. Notice that the simulator succeeds in setting the joint key as pk and the joint tag as tag for the adversary and the joint signature as σ that was returned by $\mathcal{F}_{\mathsf{J-LRS}}$. This concludes the proof. □

In the following we prove a useful fact about our ideal functionality, i.e., that the resulting signature is non-slanderable for all (possibly adversarial) choices of $(s_1, \ldots, s_{|\mathcal{R}|-1})$.

Lemma 1. *If DL problem is hard, then $\mathcal{F}_{\mathsf{J-LRS}}$ is non-slanderable.*

Proof. Consider an adversary $\mathcal{A}$ that violates the non-slanderability of $\mathcal{F}_{\mathsf{J-LRS}}$. This means that we can construct a reduction $\mathcal{S}$ that simulates the interfaces of $\mathcal{F}_{\mathsf{J-LRS}}$ for $\mathcal{A}$ and based on its output solve the DL problem. Specifically, let us denote the total interactions of $\mathcal{A}$ with **JointSpend** as q_S. The total number of queries made by $\mathcal{A}$ to all oracles are upper bounded by some polynomial $p(\lambda)$.

We now describe the reduction procedure. $\mathcal{S}$ first receives as input $(\mathbb{G}, q, G, G^a)$ as the DL challenge. $\mathcal{S}$ gives the adversary $(\mathbb{G}, q, G)$ as input. When the adversary calls the interface **KeyTagGen** as user U_0 along with another user id U_1, it samples $x_0, r \leftarrow \mathbb{Z}_q^*$. It sets $pk := G^a$ and $\mathsf{H_P}(pk) := G^r$. The reduction then returns $(x_0, pk, (G^a)^r, \mathsf{H_S}, \mathsf{H_P})$ to the adversary. $\mathcal{A}$ makes $q_{\mathsf{H_P}}$ and $q_{\mathsf{H_S}}$ oracle queries to $\mathsf{H_P}$ and $\mathsf{H_S}$ respectively. We assume w.l.o.g that the adversary $\mathcal{A}$ does not make redundant queries to any of the oracles. It runs the setup for LRS by running $\mathsf{Setup}(1^\lambda, 1^\alpha)$ and obtains pp. It then invokes $\mathcal{A}$ by giving it pp as input.

For oracle queries to $\mathsf{H_P}$ of the form pk', the reduction samples fresh random coins $r_j \leftarrow \mathbb{Z}_q^*$ for the j-th query and sets the response as G^{r_j}. It records the oracle response (pk', G^{r_j}, r_j) into the list $\mathcal{L}_{\mathsf{H_P}}$.

When the adversary queries the **JointSpend** interface with inputs $(U_0, (s_1, \ldots, s_{|\mathcal{R}|-1}), pk, tx)$ then $\mathcal{S}$ does the following:

- Parse $tx := \{\mathcal{R}, tag, pk^{\mathcal{O}}, \mu\}$
- Parse $\mathcal{R} := (pk_1, \ldots, pk_{|\mathcal{R}|})$, notice that $pk_{|\mathcal{R}|} = G^a$ and $tag = G^{ar}$
- Sample $h_{|\mathcal{R}|}, s_{|\mathcal{R}|} \leftarrow \mathbb{Z}_q^*$ and set $h_0 = h_{|\mathcal{R}|}$, for $j \in [|\mathcal{R}| - 1]$,
  - Check if $(pk_j, G^{e_j}) \in \mathcal{L}_{\mathsf{H_P}}$, and if not set $\mathsf{H_P}(pk_j) := G^{e_j}$ for random $e_j \leftarrow \mathbb{Z}_q^*$
  - Set $\mathsf{H_S}(tx^*||G^{s_j} pk_j^{h_{j-1}}||G^{e_j s_j}(G^{ch_{j-1}})) = h_j$
- Set $\mathsf{H_S}(tx||G^{s_{|\mathcal{R}|}}(G^a)^{h_{|\mathcal{R}|-1}}||(G^r)^{s_{|\mathcal{R}|}}(G^{ar})^{h_{|\mathcal{R}|-1}}) = h_{|\mathcal{R}|}$
- Set $\sigma := (s_{|\mathcal{R}|}, s_1, \ldots, s_{|\mathcal{R}|-1}, h_0)$ and return σ to $\mathcal{A}$.

Finally, $\mathcal{A}$ outputs its slander (tx^*, σ^*). The reduction aborts the execution if it is not a valid signature, or if the pair was previously obtained through the **JointSpend** interface. Let $tx^* := \{\mathcal{R}, tag, \mathcal{O}, \mu\}$ and the reduction also aborts if $tag \neq (G^a)^r$.

Operations of $\mathcal{S}$ Note that $\mathsf{H_S}$ and $\mathsf{H_P}$ queries are made by $\mathcal{S}$ during the verification process of the slander. To be more precise, a total of $|\mathcal{R}|$ number of '$\mathsf{H_S}$ operations' are needed to verify the signature (in this case the slander). We therefore have two events. Firstly, the event $\mathbf{E}$ where each of the $|\mathcal{R}|$ queries corresponding to the $|\mathcal{R}|$ verification queries (to verify the slander) have already been included in the $q_{\mathsf{H_S}}$ number of hash queries (made by $\mathcal{A}$) or in the q_S number of **JointSpend** interactions. Secondly, the event $\neg\mathbf{E}$ which denotes the complement of the above event, where $\mathcal{S}$ aborts the execution by outputting $\mathtt{abort}_1$.

If $\mathbf{E}$ happens, consider the set of $|\mathcal{R}|$ queries made by $\mathcal{A}$ to $\mathsf{H_S}$ that match the $|\mathcal{R}|$ queries made in the verification process ($|\mathcal{R}|$ queries per slander) by $\mathcal{S}$. We let $X_{i_1}, X_{i_2}, \ldots, X_{i_{|\mathcal{R}|}}$ denote the first appearance on the transcript T of each of the queries to $\mathsf{H_S}$ used by $\mathcal{S}$ for verification of the slander where $1 \leq i_1 \leq \ldots \leq i_{|\mathcal{R}|}$. (This is to consider the case of repetition of queries)

Let ℓ be such that

$$X_{i_{|\mathcal{R}|}} = \mathsf{H_S}(tx^* || G^{s_\ell} {pk_\ell}^{h_{\ell-1}} || (G^{r_\ell})^{s_\ell} (tag)^{h_{\ell-1}})$$

in the verification process by $\mathcal{S}$. We call this ℓ as the gap of σ^*.

We annotate a successful slander σ^* by $\mathcal{S}$ as a (w, ℓ)-slander if $i_1 = w$ i.e, the first appearance of all verification related queries is the w-th query to the $\mathsf{H_S}$ oracle and ℓ is the gap. $\mathcal{S}$ aborts by outputting $\mathtt{abort}_2$ if $tag \neq (G^a)^r$ and continues to perform a rewind simulation otherwise.

$\mathcal{S}$ has recorded the transcript T for the (w, ℓ) slander output by $\mathcal{A}$. Given this successful (w, ℓ) slander, $\mathcal{S}$ now rewinds the transcript T to the w-th query and gives it to $\mathcal{A}$ as a rewind simulation to obtain another transcript T' which is a successful (w, ℓ) slander again. If T' is not a successful (w, ℓ) slander again then $\mathcal{S}$ aborts by outputting $\mathtt{abort}_3$, and if not, it continues. New coin flips that are independent to T are made for all queries subsequent to the w-th query (where w is learnt from the (w, ℓ) slander from T). Note that consistency is maintained with the previous queries and also that both T and T' use the same program in $\mathcal{A}$.

Let the w-th query common to T and T' be denoted by $\mathsf{H_S}(tx^*, G^u, G^v)$.

Here, $\mathcal{S}$ knows G^u and G^v but not u and v at the time of rewind. After $\mathcal{A}$ returns the output from the rewind simulation, $\mathcal{S}$ proceeds to compute the discrete log a of G^a.

The transcript T and the rewind simulation transcript T' contain two (w, ℓ)-slander signatures such that the following pairs of equalities hold.

$$\begin{aligned} G^u &= G^{s_\ell} {pk_\ell}^{h_{\ell-1}} = G^{s_\ell + x_\ell h_{\ell-1}} \\ G^u &= G^{s'_\ell} {pk_\ell}^{h'_{\ell-1}} = G^{s'_\ell + x_\ell h'_{\ell-1}} \end{aligned} \tag{6}$$

$$\begin{aligned} G^v &= G^{r_\ell s_\ell} (G^{ar})^{h_\ell} = G^{r_\ell s_\pi + arh_{\ell-1}} \\ G^v &= G^{r_\ell s'_\ell} (G^{ar})^{h'_{\ell-1}} = G^{r_\ell s'_\ell + arh'_{\ell-1}} \end{aligned} \tag{7}$$

Notice that in equation (1) the public key pk_ℓ may be of adversary's choice. Therefore x_ℓ is not known to the reduction. Therefore the reduction uses the equation (2) to solve the DL problem and retrieve a. As the reduction knows $r_\ell, s_\ell, s'_\ell, r, h_{\ell-1}, h'_{\ell-1}$, it can do the following:

$$a = \frac{r_\ell s_\ell - r_\ell s'_\ell}{r h'_{\ell-1} - r h_{\ell-1}} \mod q$$

Analysis We first analyze the probability of $\mathcal{S}$ outputting $\mathtt{abort}_1$. We can see that in the case of event $\neg\mathbf{E}$ the conditional probability of h_0 in the forged signature σ^* satisfying the final equation in the verification process is at most $\frac{1}{t - q_H - |\mathcal{R}| q_S}$ (where t denotes all possible hash values for an input) which is negligible. For the given adversary $\mathcal{A}$ we have

$$\begin{aligned} \epsilon(\lambda) &< \Pr[\mathbf{E}] \Pr[\mathcal{A} \text{ slanders} | \mathbf{E}] + \Pr[\neg\mathbf{E}] \Pr[\mathcal{A} \text{ slanders} | \neg\mathbf{E}] \\ &\leq \Pr[\mathbf{E}] \Pr[\mathcal{A} \text{ slanders} | \mathbf{E}] + 1 \left(\frac{1}{t - q_H - |\mathcal{R}| q_S} \right) \end{aligned}$$

The probability of $\mathcal{A}$ returning a slander and having already queried for all $|\mathcal{R}|$ queries used in verification is greater than $\epsilon(\lambda)$ as $\left(\frac{1}{t - q_H - |\mathcal{R}| q_S} \right)$ is negligible.

For analyzing the probabilities $\Pr[\neg\mathtt{abort}_2]$ and $\Pr[\neg\mathtt{abort}_3]$, we do the following. We refer to the run of $\mathcal{S}$ resulting in transcript T as the first and T' as the second. The probability of a slander in the first run is $(\epsilon(\lambda))$. We can compute $\Pr[\neg\mathtt{abort}_3]$ which is the probability of the second run of $\mathcal{S}$ also resulting in a (w,ℓ) slander as $\left(\frac{\epsilon(\lambda)}{(q_H+q_S)}\right)$.

To bound the success of reduction $\mathcal{S}$ we refer to the forking lemma proposed by Bellare et al. [57]. By the forking lemma we have that $\mathcal{S}$ solves the DL problem with probability

$$\Pr\left[\mathcal{S}\text{ succeeds}\right] \geq \epsilon(\lambda)\cdot\left(\frac{\epsilon(\lambda)}{(q_H+q_S)}-\frac{1}{h}\right).$$

Here $\frac{1}{h}$ refers to the probability with which the randomness used in the second run is the same as that in the first run and this is negligible.

We can see that the complexity of $\mathcal{S}$ is no more than (q_H+q_S) times that of $\mathcal{A}$ and the probability of success of $\mathcal{S}$ against the DL problem is at least $\frac{1}{q_H+q_S}\cdot(\epsilon(\lambda))^2$ which is non-negligible. Therefore we arrive at the contradiction, thereby proving non-slanderability.

□

G Security Analysis of Payment Channels in Monero Using VTLRS

Following is the proof for Theorem 6.1.

Intuition. A in the security analysis is that the privacy property of our VTLRS is against PRAM adversaries, meaning that such an adversary cannot violate the privacy of the VTLRS commitments before the hiding time $\mathbf{T}$. To prove security, we want to switch from a hybrid execution where the VTLRS commitments are commitments to valid signatures to a hybrid where the commitments are to a zero string. To model this in the analysis (UC setting), we cannot simply switch between the hybrids as we do for standard commitment or encryption schemes. This is because the environment $\mathcal{E}$ is a polynomial time machine that will force open the VTLRS commitment eventually and thereby can distinguish the two hybrids. We deal with this by still having the second hybrid having a commitment to the valid signature but letting the simulator abort the simulation with $\mathtt{abort}_{\mathsf{priv}}$ if the adversary sends the valid signature committed to in the VTLRS commitment before hiding time $\mathbf{T}$. We can then argue that two hybrid executions are indistinguishable provided the probability of $\mathtt{abort}_{\mathsf{priv}}$ is negligible, which is guaranteed by the privacy of VTLRS. Below is the detailed proof.

Proof. In order to prove that payment channel protocol is secure, we need to describe a simulator $\mathcal{S}$ that simulates the PC operations to an adversary. Below we describe the simulation for the channel opening, channel closing and payment operation of $\mathcal{F}_{\mathsf{PC}}$. Let VTLRS-TS ($\mathsf{Setup}, \mathsf{Commit}, \mathsf{Verify}, \mathsf{Open}, \mathsf{ForceOp}$) constitute the LRS transaction scheme of monero ($\mathsf{LRS.Setup}, \mathsf{LRS.OTKGen}, \mathsf{LRS.TagGen}, \mathsf{LRS.Spend}, \mathsf{LRS.Vf}$). Let $\mathcal{S}_{\mathsf{VTLRS}}$-$\mathsf{TS}$ be the simulator guaranteed for the privacy property that simulates a proof without the knowledge of the signature.

$\mathsf{OpChannel}(c_{\langle u_1,u_2\rangle},\mu,u_1,t)$: Let u_1 be Alice and u_2 be Bob. Let Alice be the user who wants to open a channel. We have two cases to analyze: (1) Alice is corrupt by adversary $\mathcal{A}$, and (2) Bob is corrupt by adversary $\mathcal{A}$.

Alice is corrupt: We begin by describing a series of hybrid executions where we argue that each neighbouring hybrid executions are indistinguishable. If at some point, the adversary responds with a $\mathtt{abort}$, the simulator reports a failure message to the ideal functionality $\mathcal{F}_{\mathsf{PC}}$ thereby not allowing the channel creation.

Hybrid $\mathcal{H}_0$: is the the same as the real protocol execution.

Hybrid $\mathcal{H}_1$: The simulator receives the adversary's messages that are calls to $\mathcal{F}_{\mathsf{J-LRS}}$ and simulates the execution by running the simulator for $\mathcal{F}_{\mathsf{J-LRS}}$ for a corrupt user U_0. The hybrids are indistinguishable given the security of $\mathcal{F}_{\mathsf{J-LRS}}$.

We define our final simulator as following the same execution as the final hybrid. Additionally, when the adversary sends (tx',σ'), the simulator checks if it is well formed and the signature is valid. If so, the simulator forwards the message $(\mathtt{open}, c_{\langle u,u'\rangle},\mu,u',\mathbf{T})$ to the ideal functionality $\mathcal{F}_{\mathsf{PC}}$, where μ is the initial

channel capacity. When the functionality responds with $(c_{\langle u,u'\rangle}, \mu, \mathbf{T})$, the simulator authorises the operation. If everything is successful, then the simulator receives a h from $\mathcal{F}_{\mathsf{PC}}$, the simulator stores it locally.

Bob is corrupt: The simulator initiates the channel opening with $\mathcal{F}_{\mathsf{PC}}$ by sending $(\mathtt{open}, c_{\langle u,u'\rangle}, \mu, u', \mathbf{T})$ on behalf of a honest Alice. When the ideal functionality responds with $(c_{\langle u,u'\rangle}, \mu, \mathbf{T})$ to Bob, the simulator intercepts the message performs an execution as detailed below.

We begin by describing a series of hybrid executions where we argue that each neighbouring hybrid executions are indistinguishable. Finally we arrive at the simulator's execution that simulates the ideal world.

Hybrid $\mathcal{H}_0$: the execution is the same as the real protocol where a honest Alice's operations are performed. If at point, the adversary responds with `abort`, the simulator reports a failure message to the ideal functionality $\mathcal{F}_{\mathsf{PC}}$ thereby not allowing the channel creation.

Hybrid $\mathcal{H}_1$: is the same as the previous hybrid except that the simulator receives the adversary's messages that are calls to $\mathcal{F}_{\mathsf{J-LRS}}$ and simulates the execution by running the simulator for $\mathcal{F}_{\mathsf{J-LRS}}$ for a corrupt user U_1. The hybrids are indistinguishable given the security of $\mathcal{F}_{\mathsf{J-LRS}}$.

Hybrid $\mathcal{H}_2$: is the same as the previous execution except, the simulator receives (C, π) from the adversary, it obtains $\sigma \leftarrow \mathsf{ForceOp}(C)$. It then checks if $\mathsf{Verify}(tx_{\mathsf{rdm}}, C, \pi) = 1$ and $\mathsf{LRS.Vf}(tx_{\mathsf{rdm}}, \sigma) = 0$. If both checks are successful, the simulator aborts by outputting $\mathtt{abort}_{\mathsf{sound}}$. Notice that the only difference between the hybrids is the case where the simulator aborts by outputting $\mathtt{abort}_{\mathsf{sound}}$. We argue that the probability of the simulator aborting is negligible which is implied by the simulation soundness of VTLRS.

We define the final simulator as the last hybrid execution. When the adversary responds with (C, π), the simulator checks if $\mathsf{Verify}(tx_{\mathsf{rdm}}, C, \pi) = 1$ and other checks as in Alice's execution. If the checks are successful, it completes the execution on behalf Alice (posting to blockchain, etc.) and sends a message authorising the channel opening to $\mathcal{F}_{\mathsf{PC}}$ on behalf of the corrupt Bob. If the check above fails, then the simulator sends a failure message to the ideal functionality $\mathcal{F}_{\mathsf{PC}}$ thereby not authorising the channel creation. If everything is successful, then the simulator receives a h from $\mathcal{F}_{\mathsf{PC}}$, the simulator stores it locally.

$\mathsf{ClChannel}(c_{\langle u_1,u_2\rangle}, h)$: We have two cases where either Alice (u_1) or Bob (u_2) requests for channel.

Alice is corrupt and requests closing: we have the following hybrids,

Hybrid $\mathcal{H}_0$: $\mathcal{S}$ receives a closing request from the adversary on behalf of Alice, in the form of $(tx_{\mathsf{rdm}}, \sigma)$. The simulator sends $(\mathtt{close}, c_{\langle u_1,u_2\rangle}, h)$ where the simulator obtains h from when it stored during the opening of the channel. If the functionality aborts, then the simulator ignores the request from Alice and does not forward the transaction and signature to the blockchain. Otherwise, the functionality responds with $c_{\langle u_1,u_2\rangle}, \perp, h$, the simulator then forwards the transaction and the signature to the blockchain.

Hybrid $\mathcal{H}_1$: the execution is the same as before except now, the simulator aborts by outputting $\mathtt{abort}_{\mathsf{priv}}$ if the adversary outputs $(tx_{\mathsf{rdm}}, \sigma)$ such that $\mathsf{LRS.Vf}(tx_{\mathsf{rdm}}, \sigma) = 1$ before time $\mathbf{T}$. We argue that the probability with which the simulator aborts by outputting $\mathtt{abort}_{\mathsf{priv}}$ is negligible. To see this, consider an adversary who succeeds in outputting a valid transaction, signature pair given only the VTLRS on tx_{rdm} with non-negligible probability (the simulator aborts with $\mathtt{abort}_{\mathsf{priv}}$ with non-negligible probability). We can then construct a reduction that uses the adversary to break the timed privacy of the VTLRS. This is a contradiction and therefore the hybrids are indistinguishable.

The simulator is defined as the execution of the final hybrid.

Bob is corrupt and request closing: we have the following hybrids,

Hybrid $\mathcal{H}_0$: $\mathcal{S}$ receives a closing request from the adversary on behalf of Bob, in the form of a valid pair $(tx_{\mathsf{rdm},k}, \sigma_k)$. The simulator sends $(\mathtt{close}, c_{\langle u_1,u_2\rangle}, h)$ where it obtains h corresponding to $tx_{\mathsf{rdm},k}$ from the local list. If the functionality aborts, then the simulator ignores the request from Alice and does not forward the transaction and signature to the blockchain. Otherwise, the functionality responds with $c_{\langle u_1,u_2\rangle}, \perp, h$, the simulator then forwards the transaction and the signature to the blockchain.

Hybrid $\mathcal{H}_1$: the execution is the same as before except now the simulator aborts in case Bob returns a signature σ'_k that was not previously generated by Alice and Bob for payment $tx_{\mathsf{rdm},k}$. That is, if $L_{\mathsf{pay}} \neq (tx_{\mathsf{rdm},k}, \sigma'_k)$, the simulator aborts by outputting $\mathtt{abort}_{\mathsf{slander}}$. From Lemma 1 we see that this event occurs only with negligible probability and therefore we have that the hybrids are indistinguishable.

The simulator is defined as the execution of the final hybrid.

$\mathsf{Pay}(c_{\langle u_0,u_1\rangle}, v)$: is invoked by Alice ($u_0$) who wishes to make a payment of v coins through the channel to Bob (u_1). We have two possible cases of either Alice being corrupt or bob being corrupt.
Alice is Corrupt: We begin by describing a series of hybrid executions where we argue that each neighbouring hybrid executions are indistinguishable. If at point, the adversary responds with `abort`, the simulator reports a failure message to the ideal functionality $\mathcal{F}_{\mathsf{PC}}$ thereby not allowing the channel creation.
Hybrid $\mathcal{H}_0$: is the the same as the real protocol execution with a honest Bob.
Hybrid $\mathcal{H}_1$: The simulator receives the adversary's messages that are calls to $\mathcal{F}_{\mathsf{J-LRS}}$ and simulates the execution by running the simulator for $\mathcal{F}_{\mathsf{J-LRS}}$ for a corrupt user U_0. The hybrids are indistinguishable given the security of $\mathcal{F}_{\mathsf{J-LRS}}$.

The final simulator is the same execution as the previous hybrid. If the adversary did not abort at any stage, the simulator forwards the message $(\mathsf{pay}, v, c_{\langle u_0,u_1\rangle}, \mathbf{T})$ to the ideal functionality $\mathcal{F}_{\mathsf{PC}}$. The ideal functionality responds with $(h, c_{\langle u_0,u_1\rangle}, v)$ which the simulator locally stores as $(h, tx_{\mathsf{rdm},i}, \sigma_{\mathsf{rdm},i})$ denoting the i-th successful payment. If the adversary aborts at any stage, the simulator simply ignores the pay request from the adversary. If the ideal functionality aborts, the simulator does nothing.
Bob is corrupt: We begin by describing a series of hybrid executions where we argue that each neighbouring hybrid executions are indistinguishable. If at point, the adversary responds with a return 0, the simulator reports a failure message to the ideal functionality $\mathcal{F}_{\mathsf{PC}}$ thereby not allowing the channel creation.
Hybrid $\mathcal{H}_0$: is the same as the real world execution with a honest Alice.
Hybrid $\mathcal{H}_1$: is the same as the previous hybrid except that the simulator receives the adversary's messages that are calls to $\mathcal{F}_{\mathsf{J-LRS}}$ and simulates the execution by running the simulator for $\mathcal{F}_{\mathsf{J-LRS}}$ for a corrupt user U_1. The hybrids are indistinguishable given the security of $\mathcal{F}_{\mathsf{J-LRS}}$.

The simulator is the execution according to the last hybrid. If the adversary at no stage aborted, the simulator sends $(\mathsf{pay}, v, c_{\langle u_0,u_1\rangle}, \mathbf{T})$ to $\mathcal{F}_{\mathsf{PC}}$. The ideal functionality responds with $(h, c_{\langle u_0,u_1\rangle}, v)$ which the simulator locally stores as $(h, tx_{\mathsf{rdm},i}, \sigma_{\mathsf{rdm},i})$ denoting the i-th successful payment. If the ideal functionality aborts, the simulator does nothing.

Thus we have described simulators for corruptions of Alice and Bob for various operations pertaining to $\mathcal{F}_{\mathsf{PC}}$. This concludes the proof.

□

H Security Analysis of Atomic Multi-Hop Locks In Monero

Below we analyse Theorem 7.1.

Proof. We define the following sequence of hybrids, where we gradually modify the initial experiment. Unlike before, here we first describe the hybrids and later argue their indistinguishability.
Hybrid $\mathcal{H}_0$: is identical to the protocol described in Figures 7 and 8.
Hybrid $\mathcal{H}_1$: all calls to the hash function H are simulated via lazy sampling. The simulator picks a random λ bit string and responds to any query and ensures that repeated queries have consistent responses.
Hybrid $\mathcal{H}_2$: is the same as the previous hybrid except that all calls to the NIZK scheme are simulated using the simulator for the NIZK proof. And for any NIZK proof output by the adversary, the execution extracts the corresponding witness from the proof.
Hybrid $\mathcal{H}_3$: Consider the following ensemble of variables in the interaction with $\mathcal{A}$: A honest user U_j, a key pair (sk_j, pk), a state s^I, a tuple $(\ell_j, \ell_{j+1}, s^L, s^R)$ such that

$$\{\cdot, (\ell_j, s^L)\} \leftarrow \langle \cdot, \mathsf{Lock}^{\mathsf{amhl}}_{U_j}(s^I_j, sk_j, pk)\rangle$$

$$\{(\ell_{j+1}, s^R), \cdot\} \leftarrow \langle \mathsf{Lock}^{\mathsf{amhl}}_{U_j}(s^I_j, sk_j, pk), \cdot\rangle$$

If for any set of these variables, the adversary returns some k such that $\mathsf{Vf}^{\mathsf{amhl}}(\ell_{j+1}, k) = 1$ and $\mathsf{Vf}^{\mathsf{amhl}}(\ell_j, \mathsf{Rel}^{\mathsf{amhl}}(k, (s^I, s^L, s^R))) \neq 1$, then the experiment aborts.
Hybrid $\mathcal{H}_4$: Consider the following ensemble of variables in the interaction with $\mathcal{A}$: a pair of honest users (U_0, U_1) a set of (possibly corrupted) users $(U_1, \ldots, U_n)$, a key pair (sk_j, pk), a set of initial states

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

upon invocation by U_i
sends (sid, U_i, $\{L, R\}$) to U_j
if $b = \perp$ send $\perp$ to U_i and *abort*
if L insert (U_i, U_j) into $\mathcal{U}$ and sends (sid, U_i, U_j) to U_i
if R insert (U_j, U_i) into $\mathcal{U}$ and sends (sid, U_j, U_i) to U_i

Lock(sid, lid)

upon invocation by U_i
if getStatus(lid) $\neq$ `init` $\vee$ getLeft(lid) $\neq U_i$ then abort
send$_s$ (sid, lid, Lock) to getRight(lid)
receive$_s$ (sid, b) from getRight(lid)
if $b = \perp$ send $\perp$to U_i and abort
updateStatus(lid, Lock)
send$_s$ (sid, lid, Lock) to U_i

GetStatus(sid, lid)

upon invocation by U_i
return (sid, lid, getStatus(lid)) to U_i

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

upon invocation by U_i
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 $(lid_0, U_0, U_1, \texttt{init}, lid_1), (lid_{n-1}, U_{n-1}, U_n, \texttt{init}, \perp)$ into $\mathcal{L}$
send$_{an}$ (sid, $\perp$, lid_0, $\perp$, U_1, `init`) to U_0
send$_{an}$ (sid, lid_{n-1}, $\perp$, U_{n-1}, $\perp$, `init`) to U_n
$\forall i \in [1, n-1]$: insert $(lid_i, U_i, U_{i+1}, \texttt{init}, lid_{i+1})$ into $\mathcal{L}$
send$_{an}$ (sid, lid_{i-1}, lid_i, U_{i-1}, U_{i+1}, `init`) to U_i

Release(sid, 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$_s$(sid, lid, Rel) to getLeft(lid)

Figure 18: Ideal functionality $\mathcal{F}_{\mathsf{AMHL}}$ for cryptographic locks (AMHL) [5]

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

and a pair of locks (ℓ_{j-1}, ℓ_j) such that

$$\{\cdot, (\ell_{j-1}, \cdot)\} \leftarrow \langle \cdot, \mathsf{Lock}_{U_j}^{\mathsf{amhl}}(s_j^I, sk_j, pk) \rangle$$

$$\{(\ell_j, \cdot), \cdot\} \leftarrow \langle \mathsf{Lock}_{U_j}^{\mathsf{amhl}}(s_j^I, sk_j, pk), \cdot \rangle$$

If for any set of these variables, the adversary returns some k_{j-1} such that $\mathsf{Vf}^{\mathsf{amhl}}(\ell_{j-1}, k_{j-1}) = 1$ before the user U_j outputs a key k_j, such that $\mathsf{Vf}^{\mathsf{amhl}}(\ell_j, k_j) = 1$, then the experiment aborts.
Hybrid $\mathcal{H}_5$: Let $S = (U_0, \ldots, U_n)$ be an ordered set of (possibly corrupted) users. We say 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 set up a lock for a certain set $S = (A_1 || \ldots || A_{m'})$, it initialises 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.

$\mathcal{H}_0 \approx \mathcal{H}_1$: The indistinguishability of $\mathcal{H}_0$ and $\mathcal{H}_1$ follows because the distributions are identical given that H is modelled as a random oracle.
$\mathcal{H}_1 \approx \mathcal{H}_2$: The indistinguishability of $\mathcal{H}_1$ and $\mathcal{H}_2$ follows directly from the security of the NIZK scheme.
$\mathcal{H}_2 \approx \mathcal{H}_3$: To see the indistinguishability of $\mathcal{H}_2$ and $\mathcal{H}_3$ we introduce an intermediate hybrid $\mathcal{H}_2'$.
Hybrid $\mathcal{H}_2'$: The locking algorithms are substituted with the following ideal functionality. Such an interface is called by both users on input tx and $y = \sum_{k=0}^{j} y_k$, where j is the position of the lock in the chains and the y_k are defined as in the original protocol. Note that the key $pk_{U_0,U_1}, sk_{U_0,U_1}$ and $tag_{U_0U_1}$

Spend$(U_b, (s_1, \ldots, s_{|\mathcal{R}|-1}), pk_{U_0U_1}, tx, y)$

Upon invocation by both users U_0 and U_1:
where U_b ($b \in \{0,1\}$) gives inputs $(s_1, \ldots, s_{|\mathcal{R}|-1})$
parse $tx := (\mathcal{R}, tag_{U_0,U_1}, pk^{\mathcal{O}}, \mu)$
parse $\mathcal{R} := (pk_1, \ldots, pk_{|\mathcal{R}|}),\ s.t.\ pk_{|\mathcal{R}|} := pk_{U_0U_1}$
choose $s'_0 \leftarrow \mathbb{Z}_q^*$
Compute
$L_0 := G^{s'_0}, R_0 := \mathsf{H_P}(pk_{|\mathcal{R}|})^{s'_0}$
$h_0 := \mathsf{H_S}(tx||L_0||R_0)$
for $i \in [|\mathcal{R}| - 1]$ **do**
$L_i := G^{s_i} pk_i^{h_{i-1}}$,
$R_i := \mathsf{H_P}(pk_i)^{s_i} tag^{h_{i-1}}$
$h_i := \mathsf{H_S}(tx||L_i||R_i)$
endfor
set $s_0 := s'_0 - h_{|\mathcal{R}|-1} sk_{U_0U_1}$
$\sigma := (s_0, s_1, \ldots, s_{|\mathcal{R}|-1}, h_0)$
$\sigma' := (s_0 - y, s_1, \ldots, s_{|\mathcal{R}|-1}, h_0)$
return σ' to U_b

refers to the previously established keys and tag in the call to $\mathcal{F}_{\mathsf{J-LRS}}$.

We defer the proof of indistinguishability of $\mathcal{H}_2$ and $\mathcal{H}'_2$ to Lemma 2.

We now argue the indistinguishability of $\mathcal{H}'_2$ and $\mathcal{H}_3$.

Let cheat be the event that triggers the abort of the execution of $\mathcal{H}_3$. This happens when the adversary returns k such that $\mathsf{Vf}^{\mathsf{amhl}}(\ell_{j+1}, k) = 1$ and that $\mathsf{Vf}^{\mathsf{amhl}}(\ell_j, \mathsf{Rel}^{\mathsf{amhl}}(k, (s^I, s^L, s^R))) \neq 1$. Assume towards contradiction that $\Pr[\mathsf{cheat}|\mathcal{H}_3] \geq \frac{1}{p(\lambda)}$ for some polynomial $p()$. Then we can construct the following reduction against the non-slanderability of $\mathcal{F}_{\mathsf{J-LRS}}$ (Lemma 1). The reduction gets $(x, pk, tag, \mathsf{H_S}, \mathsf{H_P})$ as input. All calls to the spending algorithm are redirected to **JointSpend** interface of $\mathcal{F}_{\mathsf{J-LRS}}$. If the event cheat happens, the reduction obtains $(\ell^*, k^*) = (tx^*, pk, \sigma^*)$ where $\sigma^* := (s_0^*, \ldots, s_{|\mathcal{R}|-1}^*, h_0^*)$. The reduction returns tx^*, σ^* as its slander for the tag tag corresponding to pk that is included in the transaction tx^*.

The reduction is clearly efficient. Assume for the moment that $t \in [q]$ where $q \in p(\lambda)$ is the interaction where cheat happens. Let $j+1$ be the index that identifies the lock ℓ^* in the corresponding payment path. Since cheat happens we have that $\mathsf{LRS.Vf}(tx^*, k^*) = 1$ and the release fails, i.e., $\mathsf{Vf}^{\mathsf{amhl}}(\ell_j, \mathsf{Rel}^{\mathsf{amhl}}(k, (s_j^I, s_i^L, s_j^R))) \neq 1$ (where ℓ_j is the lock in the previous position of ℓ^*). Recall that the release algorithm parses $s_j^L := (s', s'_1, \ldots, s'_{|\mathcal{R}'|-1}, h'_0)$ and $\sigma^* := (s_0^*, \ldots, s_{|\mathcal{R}|-1}^*, h_0^*)$ and returns $((s' + s_0^* - s_j^R - y), s'_1, \ldots, s'_{|\mathcal{R}'|-1}, h'_0)$. We see that

$$
\begin{aligned}
&(s' + s_0^* - s_j^R - y) \\
&= \left(\left(s_{j,0} - \sum_{i=0}^{j-1} y_i \right) + s_0^* - \left(s_{t,0} - \sum_{i=0}^{j} y_i \right) - y \right) \\
&= (s_{j,0} + s_0^* - s_{t,0})
\end{aligned}
$$

where $s_{t,0}$ is part of the answer of the **JointSpend** interface in the t-th session. This implies that $s^* \neq s_{t,0}$, otherwise we have that $(s_{j,0}, s'_1, \ldots, s'_{|\mathcal{R}'|-1}, h'_0)$ as a valid signature since it is an output of the **JointSpend** interface. Since no transaction is queried twice in the same session to the interface (**Spend**), we have that (tx^*, σ^*) as a valid slander. By assumption this happened with probability

at least $\frac{1}{q \cdot p(\lambda)}$ which is a contradiction to the non-slanderability property of $\mathcal{F}_{\mathsf{J-LRS}}$. Therefore we have $\Pr[\mathsf{cheat}|\mathcal{H}_2'] \leq \mathsf{negl}(\lambda)$. Since $\mathcal{H}_2$ and $\mathcal{H}_3$ differ only when cheat happens and $\mathcal{H}_3$ aborts, we from Lemma 2 that $\mathcal{H}_2$ and $\mathcal{H}_3$ are indistinguishable.
$\mathcal{H}_3 \approx \mathcal{H}_4$: Let $q \in p(\lambda)$ where $p()$ is some polynomial be a bound on the number of interactions. Let cheat be the event that triggers an abort in $\mathcal{H}_4$ but not in $\mathcal{H}_3$. We show that $\Pr[\mathsf{cheat}|\mathcal{H}_4] \leq \mathsf{negl}(\lambda)$ which shows the indistinguishability between the hybrids. Assume towards contradiction, then we construct the following reduction against the discrete logarithm problem. The reduction takes as input some $Y^* \in \mathbb{G}$ and it guesses a session $j \in [q]$ and some index $i \in [n]$. The setup algorithm of the j-th session is modified as follows: it receives $\mathsf{H_P}(pk_{i-1,i}), \mathsf{H_P}(pk_{i,i+1})$. It sets $\mathsf{H_P}(pk_{i,i+1}) := G^{\alpha_i}$ where $\alpha_i \leftarrow \mathbb{Z}_q^*$. It then sets $Y_i := Y^*$ and $X_i := (Y^*)^{\alpha_i}$. Set $Y_{i-1} := \frac{Y_i}{G^{y_i}}$ for some randomly chosen $y_i \leftarrow \mathbb{Z}_q^*$.

Then for all $u \in \{i-1, \ldots, 0\}$, the setup samples some $y_u, \alpha_u, \alpha_{u-1} \in \mathbb{Z}_q^*$, sets $\mathsf{H_P}(pk_{u,u+1}) := G^{\alpha_u}, \mathsf{H_P}(pk_{u-1,u}) := G^{\alpha_{u-1}}$ and returns $(Y_{u-1}, Y_u, X_{u-1}, X_u, y_u, \pi_u)$ where $Y_{u-1} := \frac{Y_u}{G^{y_u}}$, $X_u := Y_u^{\alpha_u}$, $X_{u-1} := (Y_{u-1})^{\alpha_{u-1}}$ and $\pi_u := (\pi_{0,u}, \pi_{1,u})$ are both simulated proofs for statements $stmt_{0,u} := \{Y_{u-1}, X_{u-1}, G^{\alpha_{u-1}}\}$ and $stmt_{1,u} := \{Y_u, X_u, G^{\alpha_u}\}$ respectively.

Then for all $u \in [i+1, n-1]$, the setup gets $\mathsf{H_P}(pk_{u-1,u}), \mathsf{H_P}(pk_{u,u+1})$. It samples $y_u, \alpha_u \in Z_q^*$ and returns $(Y_{u-1}, Y_u, X_{u-1}, X_u, y_u, \pi_u)$ where $Y_u := Y_{u-1} \cdot G^{y_u}$, $X_u := Y_u^{\alpha_u}$ and $\pi_u := (\pi_{0,u}, \pi_{1,u})$ are both simulated proofs for statements $stmt_{0,u} := \{Y_{u-1}, X_{u-1}, G^{\alpha_{u-1}}\}$ and $stmt_{1,u} := \{Y_u, X_u, G^{\alpha_u}\}$ respectively.

The outputs are given to the parties where specifically to U_n the setup gives $(Y_{n-1}, X_{n-1}, x_n := \Sigma_{j=i}^{n-1} y_j)$, where $X_{n-1} := \mathsf{H_P}(pk_{n-1,n})^{x_n}$. If the node U_i is requested to release the lock, the reduction aborts. At some point the adversary $\mathcal{A}$ outputs $k^* = (s_0^*, s_1^*, \ldots, s_{|\mathcal{R}|-1}^*, h_0^*)$. The reduction parses s^R of user U_{i-1} and returns $(s_0^* + y_i - s^R)$.

The reduction does not abort whenever j and i are guessed correctly. Since U_i is honest, the distribution induced by the modified setup algorithm is identical to the original to the eyes of the adversary. The even cheat happens only when k^* is a valid opening for ℓ_{i-1} and the release algorithm is successful. Therefore, we have s^R which is of the form $s'_{0,0} + s'_{0,1} - h_{|\mathcal{R}|-1} \cdot (sk_{i-1} + sk_i) = s_0^* - y$ for some $y \in \mathbb{Z}_q^*$. We have

$$\begin{aligned} G^{(s_0^* + y_i - s^R)} &= G^{(s^R + y + y_i - s^R)} \\ &= G^{(y + y_i)} \\ &= G^{y_i} G^{y} \\ &= G^{y_i} Y_{i-1} \\ &= G^{y_i} \frac{Y^*}{G^{y_i}} \\ &= Y^* \end{aligned}$$

Notice that if we have $s^R = s' - y$ where $s' \neq s_0^*$, then this is a valid slander as we saw previously in the case of $\mathcal{H}_2' \approx \mathcal{H}_3$. Therefore the reduction is able to solve the DL problem with probability at least $\frac{1}{qnp(\lambda)}$. This is a contradiction and therefore we have that hybrids are indistinguishable.
$\mathcal{H}_4 \approx \mathcal{H}_5$: Adversarial sets are always interleaved by a hones node. Therefore in $\mathcal{H}_4$ 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 chosen randomly we have that $Y' = Y_{i-1} G^y$ for some uniformly sampled Y' from $\mathbb{G}$ which corresponds to the view of $\mathcal{A}$ in $\mathcal{H}_5$. Notice that for corresponding X_i's and π_i's we have simulated proofs and therefore the simulator in the execution need know the secret exponent of Y'.

The simulator therefore is defined as the last hybrid except that the actions of $\mathcal{S}$ are determined by the interaction with the ideal functionality $\mathcal{F}_{\mathsf{AMHL}}$. The simulator intercepts the communications of the adversary and receives queries from $\mathcal{F}_{\mathsf{AMHL}}$:

- $(\cdot, \cdot, \cdot, \cdot, \mathsf{Init})$: the simulator reconstructs the adversarial set from the id's and sets up a fresh lock chain.
- $(\cdot, \mathsf{lock})$: the simulator initiates the locking procedure with the adversary and replies with $\perp$ if the execution is not successful.
- $(\cdot, \mathsf{Rel})$: the simulator releases the lock and publishes the key.

If $\mathcal{A}$ interacts with a honest user the simulator queries the corresponding interface of $\mathcal{F}_{\mathsf{AMHL}}$.

Note that the simulator is efficient and interacts as the adversary with the ideal world. Furthermore, the simulator is always consistent with the ideal world regarding invalid messages and aborts. □

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

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

Proof. The proof consists of the description of the simulator for the interactive lock algorithm. We describe two simulators depending on whether the adversary is playing the role of the left or right user. For each proof, both the simulators implicitly check that the given witness is valid and abort if not.

Left user U_0 is corrupt: Prior to the interaction the simulator is sent (Y', Y, X', X, y, π) where $\pi := (\pi_0, \pi_1)$, such that $\pi_0 := (\mathsf{prove}, stmt_0, y_0^*)$, $\pi_1 := (\mathsf{prove}, stmt_1, y_1^*)$, $stmt_0 := \{Y', X', h'\}$ and $stmt_1 := \{Y, X, h\}$ for language $\mathcal{L}_{\mathsf{eqdl}}$. All these constitute the state corresponding to the execution of the lock. After agreeing on tx (which contains ring $\mathcal{R}$ and tag tag), the simulator chooses random $(s_1, \ldots, s_{|\mathcal{R}|-1}, h^1)$ and sends h^1 to the adversary. The simulator queries the interface **Spend** on input $(U_0, (s_1, \ldots, s_{|\mathcal{R}|-1}), pk_{U_0U_1}, tx, y_0^*)$ and receives a signature $\sigma := (s_0, s_1, \ldots, s_{|\mathcal{R}|-1}, h_0)$. At some point the adversary sends $(L'_{0,0}, R'_{0,0}, \pi)$ where $\pi := (\mathsf{prove}, stmt, s'_{0,0})$, $stmt := (L'_{0,0}, R'_{0,0}, h)$. The simulator does the following:

$$\begin{aligned}
&\textbf{for } i \in [|\mathcal{R}| - 1] \textbf{ do}\\
&\quad L_i := G^{s_i} pk_i^{h_{i-1}}\\
&\quad R_i := \mathsf{H}_{\mathsf{P}}(pk_i)^{s_i} tag^{h_{i-1}}\\
&\quad h_i := \mathsf{H}_{\mathsf{S}}(tx||L_i||R_i)
\end{aligned}$$

$$\begin{aligned}
L_{|\mathcal{R}|} &:= G^{s_0} pk_{|\mathcal{R}|}^{h_{|\mathcal{R}|-1}}\\
R_{|\mathcal{R}|} &:= \mathsf{H}_{\mathsf{P}}(pk_{|\mathcal{R}|})^{s_0} tag^{h_{|\mathcal{R}|-1}}
\end{aligned}$$

The simulator then sets

$$L'_{0,1} := \frac{L_{|\mathcal{R}|}}{L'_{0,0} \cdot Y}$$

and

$$R'_{0,1} := \frac{R_{|\mathcal{R}|}}{R'_{0,0} \cdot X}$$

$$\pi' := (\mathsf{proof}, sid, stmt')$$

where $stmt' := (L'_{0,1}, R'_{0,1}, h)$. It sets

$$\sigma' := ((s_0 - s'_{0,0} - h_{|\mathcal{R}|-1} \cdot x_0), s_1, \ldots, s_{|\mathcal{R}|-1})$$

Here x_0 is the extracted by the simulator during the joint key and tag generation functionality from $\mathcal{A}$. The simulator sends $(L'_{0,1}, R'_{0,1}, \pi', s_1, \ldots, s_{|\mathcal{R}|-1}, (s_0 - s'_{0,0} - h_{|\mathcal{R}|-1} \cdot x_0))$.

Right user U_1 is corrupt: Prior to the interaction the simulator is sent (Y', Y, X', X, y, π) where $\pi := (\pi_0, \pi_1)$, such that $\pi_0 := (\mathsf{prove}, stmt_0, y_0^*)$, $\pi_1 := (\mathsf{prove}, stmt_1, y_1^*)$, $stmt_0 := \{Y', X', h'\}$ and $stmt_1 := \{Y, X, h\}$ for language $\mathcal{L}_{\mathsf{eqdl}}$. All these constitute the state corresponding to the execution of the lock. After agreeing on tx (which contains ring $\mathcal{R}$ and tag tag) the simulator receives $((L'_{0,1}, R'_{0,1}, \pi', s_1, \ldots, s_{|\mathcal{R}|-1}))$ as a hash oracle query where $\pi' := (\mathsf{prove}, stmt', s'_{0,1})$ such that $stmt' := (L'_{0,1}, R'_{0,1}, h')$. Here $h' := \mathsf{H}_{\mathsf{P}}(pk_{|\mathcal{R}|})$ where $pk_{|\mathcal{R}|}$ is the shared key. The simulator then queries

the interface **Spend** with $(U_1, (s_1, \ldots, s_{|\mathcal{R}|-1}), pk_{U_0U_1}, tx, y_1^*)$ as input and receives a signature $\sigma := (s_0, s_1, \ldots, s_{|\mathcal{R}|-1}, h_0)$. The simulator does the following:

$$\begin{aligned}
&\textbf{for } i \in [|\mathcal{R}| - 1] \textbf{ do}\\
&\quad L_i := G^{s_i} pk_i^{h_{i-1}}\\
&\quad R_i := \mathsf{H_P}(pk_i)^{s_i} tag^{h_{i-1}}\\
&\quad h_i := \mathsf{H_S}(tx||L_i||R_i)
\end{aligned}$$

$$\begin{aligned}
L_{|\mathcal{R}|} &:= G^{s_0} pk_{|\mathcal{R}|}^{h_{|\mathcal{R}|-1}}\\
R_{|\mathcal{R}|} &:= \mathsf{H_P}(pk_{|\mathcal{R}|})^{s_0} tag^{h_{|\mathcal{R}|-1}}
\end{aligned}$$

In the above computation, the simulator sets random oracle values $\mathsf{H_S}$ and $\mathsf{H_P}$ on the fly (if not set already). The simulator then sets

$$L'_{0,0} := \frac{L_{|\mathcal{R}|}}{L'_{0,1} \cdot Y'}$$

and

$$R'_{0,1} := \frac{R_{|\mathcal{R}|}}{R'_{0,0} \cdot X'}$$

$$\pi' := (\mathsf{proof}, sid, stmt')$$

where $stmt' := (L'_{0,0}, R'_{0,0}, h)$. The simulator specifically sets $\mathsf{H_S}(tx||L_{|\mathcal{R}|}||R_{|\mathcal{R}|}) := h_0$ that it obtained from the interface **Spend**. It receives $(L_{0,1'}, R'_{0,1}, \pi', s'_1, \ldots, s'_{|\mathcal{R}|-1}, s_{0,1})$ from the adversary and checks as in the protocol. The simulator computes the following:

$$\begin{aligned}
&\textbf{for } i \in [|\mathcal{R}| - 1] \textbf{ do}\\
&\quad L'_i := G^{s'_i} pk_i^{h'_{i-1}}\\
&\quad R'_i := \mathsf{H_P}(pk_i)^{s'_i} tag^{h'_{i-1}}\\
&\quad h'_i := \mathsf{H_S}(tx||L'_i||R'_i)
\end{aligned}$$

The simulator then checks if $s_{0,1} = s'_{0,1} - h'_{|\mathcal{R}|-1} \cdot x_1$. Here x_1 denotes the right user's secret that was extracted by the simulator of the joint key and tag generation functionality for $\mathcal{A}$. If the above checks are successful, the simulator responds with s_0 (obtained from the interface **Spend**) to the adversary.

Both simulators are efficient and the distributions induced by the simulated views are negligibly close to the real execution. This concludes the proof of Lemma 2. □