A^{2}L: Anonymous Atomic Locks for Scalability and Interoperability in Payment Channel Hubs

Erkan Tairi
TU Wien
erkan.tairi@tuwien.ac.at

Pedro Moreno-Sanchez
TU Wien
pedro.sanchez@tuwien.ac.at

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

ABSTRACT

Payment channel hubs (PCHs) constitute a promising solution to the inherent scalability problems of blockchain technologies, allowing for off-chain payments between sender and receiver through an intermediary, called the tumbler. While state-of-the-art PCHs provide security and privacy guarantees against a malicious tumbler, they fall short of other fundamental properties, such as interoperability, fungibility, and efficiency.

In this work, we present Trilero, the first PCH protocol to achieve all aforementioned properties. Trilero builds upon A^{2}L, a novel cryptographic primitive that realizes a three-party protocol for conditional transactions, where the intermediary pays the receiver only if the latter solves a cryptographic challenge with the help of the sender, which implies the sender has paid the intermediary. We prove the security and privacy guarantees of Trilero and A^{2}L in the Universal Composability framework and present three provably secure instantiations based on Schnorr, ECDSA and DLSAG signatures. Trilero requires only digital signatures and timelock functionality from the underlying cryptocurrencies and thus is backwards compatible with virtually all cryptocurrencies available today, even those that do not support any form of scripting language such as Monero, Ripple or Stellar. Moreover, transactions required in Trilero are structurally equal to simple payments, helping thereby to maintain the fungibility of the cryptocurrencies.

We implemented Trilero and compared it to TumbleBit, the state-of-the-art Bitcoin-compatible PCH. Asymptotically, Trilero has a communication and computation complexity that is linear, as opposed to binomial, in the security parameter. In practice, our ECDSA-, DLSAG- and Schnorr-based constructions require ~19x, ~23x and ~30x less bandwidth, respectively, while being comparable to TumleBit with respect to computation time.

1 INTRODUCTION

The increasing adoption of cryptocurrencies has raised scalability issues [15] that go beyond the rapidly growing blockchain size. For instance, the permissionless nature of the consensus algorithm underlying widely deployed cryptocurrencies such as Bitcoin and Ethereum strictly limits their transaction throughput to tens of transactions per second at best [15], which contrasts with the throughput of centralized payment networks such as Visa that supports peaks of up to 47,000 transactions per second [51].

Among the several efforts to mitigate these scalability issues [32, 33, 45], payment channels have emerged as the most widely deployed solution in practice. The core idea of payment channels is to let users lock a certain amount of coins (called *collateral*) in a multisig address[1] (called *channel*) controlled by them, storing the corresponding transaction on-chain. From that moment on, these two users can pay each other by simply agreeing on a new distribution of the coins locked in the channel: the corresponding transactions are stored locally, that is, off-chain. When the two users disagree on the current redistribution or simply terminate their economical relation, they submit an on-chain transaction that sends back the coins to their owners according to the last agreed distribution of coins, thereby closing the channel. Thus, payment channels require only two on-chain transactions (i.e., open and close channel), yet supporting arbitrarily many off-chain payments, which significantly enhances the scalability of the underlying blockchain.

The problem with this simple construction is that in order to pay different people, a user should establish a channel with each of them, which is computationally and financially prohibitive, as this user would have to lock an amount of coins proportional to the number of users she wants to transact with. Payment channel networks (PCNs) offer a partial solution to this problem, enabling multi-hop payments along channel paths: if one sees a PCN as a graph where nodes are users and edges are channels, PCNs enable payments between any pair of nodes connected by a path in the graph. PCNs, however, rise the issue of finding paths in a network and maintaining the network topology.

1.1 Payment Channel Hubs (PCHs)

PCHs constitute a conceptually simpler solution to the aforementioned problem. Each party opens a channel with a central party, called the *tumbler*, which mediates payments between each pair of sender and receiver. In particular, if the sender wants to transfer x coins to the receiver, the sender pays $x + fee$ to the tumbler, which then forwards x coins to the receiver, where *fee* denotes a fee charged by the tumbler to conduct the transaction. Such a naïve construction, despite being still deployed in many gateways, suffers from obvious *security and privacy issues*: the tumbler could steal coins [5, 52] from honest parties (e.g., by simply not forwarding a payment) and identify who is paying to whom [5, 25].

Security can be seen in terms of transaction atomicity and should protect the two participants who are sending coins. Atomicity is thus two-fold: (i) the tumbler should receive the money from the sender only if the tumbler has forwarded the corresponding amount to the receiver; (ii) the receiver should receive money from the tumbler only if the sender has paid the corresponding amount to the tumbler. Privacy covers unlinkability, that is, the tumbler should not able to link the sender and receiver of a given payment. As these properties seem contradictory (i.e., how can the tumbler

[1] A multisig address requires all address owners to agree on the usage of the coins stored therein, which is achieved by signing the corresponding transaction.

Table 1: Comparison among state-of-the-art PCH.

	Atomicity	Unlinkability	Fungibility	Interoperability (Required functionality)
BOLT [26]	●	●	●	○ (Zcash)
Perun [20]	●	○	○	○ (Ethereum)
NOCUST [31]	●	○	○	○ (Ethereum)
Teechain [34]	●	●	●	◐[2] (Trusted Execution Environment)
TumbleBit [27]	●	●[1]	○	◐[3] (HTLC-based currencies)
A^{2}L	●	●[1]	●	● (Digital signature verification and time locks)

[1] Payments have fixed amounts; [2] Every user must run a TEE; [3] Not supported by scriptless cryptocurrencies (e.g., Monero).

ensure atomicity without knowing who pays to whom?), designing a secure and privacy-preserving PCH is a technical challenge.

Besides security and privacy, another fundamental property is *interoperability*: the tumbler should be able to mediate payments in different cryptocurrencies (e.g., the sender transfering bitcoins and the receiver getting ethers), thereby enabling cross-chain applications like exchanges and cross-currency mixing.

Finally, a desirable property in any currency is *fungibility*, which means that all coins should be indistinguishable from each other: in the specific case of PCHs, payments performed through the tumbler should look the same as standard payments, as otherwise, e.g., coins produced by a tumbler might be considered tainted and not accepted by certain parties.

1.2 State-of-the-art in PCH

BOLT [26] is an off-chain protocol for PCHs that provides strong anonymity guarantees by leveraging the zero-knowledge proofs of the underlying Zcash cryptocurrency. BOLT also inherits the fungibility guarantees provided by Zcash.[2] BOLT, however, is only compatible with ZCash since it requires zero-knowledge proofs.

Perun [20] is an off-chain protocol that relies on Turing-complete smart contracts to support payment channels. Moreover, Perun builds the PCH upon virtual channels, a smart contract-based construction that intuitively allows to fold two channels (e.g., Alice → Tumbler → Bob) into a single channel (Alice → Bob). This technique, however, inherently leaks the sender-receiver relation between Alice and Bob to the tumbler. Additionally, Perun lacks fungibility, as transactions encode a logic that makes them distinguishable from transactions performed by other contracts, as well as interoperability, as it works only in Ethereum.

NOCUST [31] is an off-chain protocol that relies on an untrusted operator to manage the off-chain payments among parties. The operator periodically includes a summary on-chain that includes the balances and the transactions for public verifiability. As Perun, NOCUST does not provide unlinkability against a malicious intermediate (i.e., operator). Similar to Perun, NOCUST also lacks fungibility and interoperability.

TeeChain [34] is an off-chain protocol that leverages trusted execution environments (e.g., Intel SGX) to manage off-chain payments and the handling of disputes. Thus, parties are required to run a TEE, which hinders the widespread deployment of this approach.

TumbleBit [27] is a cryptographic protocol for PCHs that makes transactions unlinkable (i.e., the tumbler does not learn who is paying whom). TumbleBit requires to fix the same value for all transactions, achieving a value privacy property that is weaker than the one provided by Bolt, called privacy of the compatible interaction graph: the tumbler learns how many coins each party sends and receives in aggregated form, but not how much is sent in each transaction. However, due to the underlying cut-and-choose technique, TumbleBit requires computation and communication costs that grow binomially in the security parameter. For instance, enforcing only 80 bits of security requires messages of size between 250 and 400 KB for a single payment, which implies running times of up to 10 seconds in the worst case. Moreover, TumbleBit relies on the hash-time lock contract (HTLC), a Bitcoin script-based construction that allows for payments conditioned on obtaining the preimage of a hash function. This, however, limits the deployment of TumbleBit to those cryptocurrencies supporting HTLC, ruling out scriptless ones such as Ripple, Stellar, or Monero. Furthermore, it hinders fungibility as multisig HTLC-based payments are clearly distinguishable from standard payments.

We summarize the properties achieved by each PCH construction in Table 1. Most notably, all state-of-the-art PCHs fail to achieve at least one of the aforementioned properties and, in particular, all of them fall short of interoperability: while BOLT, Perun and NOCUST totally lack it, Teechain achieves it at the cost of adding a new trust assumption (TEE), whereas TumbleBit is restricted to blockchains supporting HTLC contracts, further suffering from a high communication and computation complexity.

This state of affairs leads to the following question:

Is it possible to have a PCH interoperable with virtually all cryptocurrencies that is practical and achieves the security and privacy notions of interest?

1.3 Our Contributions

This work answers the previous question in the affirmative and presents the first secure, privacy-preserving, interoperable, and fungibility-preserving PCH cryptographic construction, whose communication and computational complexity is just linear in the security parameter. Specifically,

- We introduce Trilero, a cryptographic PCH realization whose core technical ingredient is a novel cryptographic primitive called anonymous atomic locks (A^{2}L). Intuitively, A^{2}L realizes a three-party protocol for conditional transactions, where the intermediary pays the receiver only if the latter solves a cryptographic challenge with the help of the sender, which implies that the sender has paid the intermediary. We model Trilero as well as its security and privacy properties (namely, atomicity and unlinkability) in the Universal Composability (UC) framework [7], providing the first formalization of the PCH problem in UC.

[2]Here we consider only coins held at shielded addresses that have not been tainted by combining them with unshielded addresses [29].

- We give three concrete instantiations, based on Schnorr, ECDSA and DLSAG signature scheme, respectively. While Schnorr provides the most efficient protocol in terms of communication and computation overhead, ECDSA is arguably the most widely deployed signature scheme in practice, thereby achieving a high degree of interoperability (e.g., we can realize a tumbler receiving bitcoins and forwarding ethers). Notice also that it is possible to combine our constructions if they are instantiated over the same group [39]. By dispensing from HTLCs, our instantiations offer the highest degree of interoperability among the state-of-the-art PCHs: e.g., Ripple and Stellar support ECDSA and Schnorr but not HTLCs. We further show how to integrate A^{2}L in a scriptless cryptocurrency such as DLSAG-based Monero [41], a variant of Monero proposed to add support for payment channels that is being considered in the Monero community [43].
- Our A^{2}L instantiations incur communication and computation costs that are linear in the security parameter. Our evaluation shows that they require a running time of ~1 second for ECDSA, ~0.7 second for DLSAG and ~0.5 second for Schnorr. Furthermore, the communication cost is 17.3KB for ECDSA, 14.3KB for DLSAG and 10.8KB for Schnorr. When compared to TumbleBit, the most interoperable PCH prior to this work, which requires binomial communication and computation complexity, our experimental evaluation shows that ECDSA-, DLSAG- and Schnorr-based A^{2}L require ~19x, ~23x and ~30x less bandwidth, respectively, and similar computation costs, despite targeting a higher security level (128-bit vs 80-bit). These results demonstrate that A^{2}L is the most efficient Bitcoin-compatible PCH. Furthermore, A^{2}L transactions are indistinguishable from standard transactions in that they rely on neither multisigs nor HTLCs, thereby offering fungibility guarantees.

2 PROBLEM DESCRIPTION

A payment-channel hub (PCH) can be represented as a graph, where each vertex represents a party P, and each weighted edge $(P_i, P_j)^{x_i, x_j}$ represents a payment channel between two parties P_i and P_j. The weight x_i, x_j on an edge (P_i, P_j) is a pair representing the balance of each channel end-point. A PCH allows for off-chain payments between two parties connected by an intermediary, called tumbler and denoted as P_t. A payment from P_s (sender) to P_r (receiver) for amt coins through channels $(P_s, P_t)^{x_s, x_t}$, $(P_t, P_r)^{x'_t, x'_r}$ requires that $x_s \geq \text{amt}$ and $x'_t \geq \text{amt}$. If these prerequisites are met, a payment updates both channels as follows: $x_s := x_s - \text{amt}$, $x_t := x_t + \text{amt}$, $x'_t := x'_t - \text{amt}$ and $x'_r := x'_r + \text{amt}$. In other words, amt is moved from the sender to the tumbler and from the tumbler to the receiver, updating the respective channel deposits. We formalize the notion of a PCH later in Definition 3.2.

Challenges in PCH. A straw man approach to realize a PCH is to let P_s pay P_t, and later on let P_t forward the payment to P_r. However, this falls short of two fundamental properties, namely, unlinkability and atomicity. For the former, it is easy to see that even in the presence of several simultaneous payments, P_t learns that the two channels that are immediately updated are part of the same payment, i.e., who pays to whom. Hence the transactions belonging to different payments must be intertwined in a non-predictable way. For the latter, if P_s pays P_t first, then a malicious

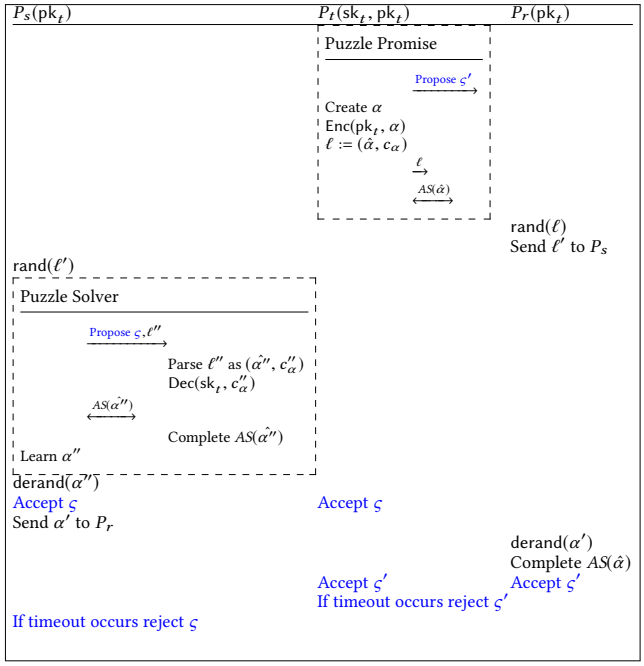

Figure 1: Overview of Trilero. Here ς and ς' denote the updates $(P_s, P_t)^{x_s - \text{amt}, x_t + \text{amt}}$ and $(P_t, P_r)^{x'_t - \text{amt}, x'_r + \text{amt}}$ respectively. Black pseudocode denotes operations in A^{2}L.

P_t could get the money from P_s without paying P_r; even if P_t pays first, there is still an attack, since a malicious P_s could abstain from paying P_t, which would thus incur in a loss. Hence, the update in the channel between P_t and P_r should be *conditioned* to the update in the channel between P_s and P_t. Achieving unlinkability and atomicity simultaneously is challenging, since one has to condition two updates without establishing any observable link between the two of them.

2.1 Our Approach

Our approach to achieve unlinkability and atomicity consists of two steps. First, we design anonymous atomic locks (A^2L), a three-party cryptographic primitive that allows for synchronizing the updates on two payment channels atomically while preserving unlinkability. Second, we introduce Trilero, a PCH protocol that performs the actual update on the channels by leveraging A^2L.

2.1.1 Anonymous Atomic Locks (A^2L). This primitive tackles the unlinkability and atomicity challenges by conditioning the updates on channels $(P_s, P_t)^{x_s, x_t}$ and $(P_t, P_r)^{x'_t, x'_r}$. For that, P_t creates a cryptographic puzzle ℓ so that an update on $(P_t, P_r)^{x'_t, x'_r}$ is successful only if P_r can provide a solution to ℓ. At the beginning, only P_t knows this solution. In order to obtain it, P_s initiates an update on channel $(P_s, P_t)^{x_s, x_t}$ that is successful only if P_t reveals a solution to a randomized version of ℓ, which is in turn a randomized solution of the original puzzle. This mechanism thereby provides atomicity. Moreover, if there are several payments in parallel, the randomization prevents P_t from linking the sender to the receiver.

In our instantiation of A^{2}L, we leverage *adaptor signatures* [39, 44], a two-party protocol to create a signature σ^* on a message m

that is "almost valid", meaning that the following two conditions hold: (i) a party P_2 can finish the signature on its own only if it gets to know a secret value α^*; (ii) if P_2 finishes the signature on its own, then the other party P_1 learns the secret value α^*. In the following, we denote this protocol as $\{\perp, \sigma^*\} \leftarrow \langle AS_{P_1}(m, \hat{\alpha^*}), AS_{P_2}(m, \hat{\alpha^*})\rangle$, where $\hat{\alpha^*}$ denotes a one-way encoding of α^*.

We have divided A^{2}L in two main protocols, namely *puzzle promise* and *puzzle solver*, as shown in Figure 1. The puzzle promise protocol takes as input an update on channel $(P_t, P_r)^{x'_t, x'_r}$. First, P_t generates a secret α and creates the cryptographic challenge $\ell := (\hat{\alpha}, c_\alpha)$, where $\hat{\alpha}$ is the output of a one-way function with α as input, and c_α is an encryption of α using a homomorphic encryption scheme under P_t's public key pk_t. Intuitively, $\hat{\alpha}$ is required to set the condition on the update of the channel $(P_t, P_r)^{x'_t, x'_r}$, whereas c_α is required in the *puzzle solver* protocol. The *puzzle promise* ends with P_t and P_r computing $\left\{\perp, \sigma^*_{P_t,P_r}\right\} \leftarrow \langle AS_{P_t}((P_t, P_r)^{x'_t, x'_r}, \hat{\alpha}), AS_{P_r}((P_t, P_r)^{x'_t, x'_r}, \hat{\alpha})\rangle$. It is important to note that at this point, P_r cannot yet convert $\sigma^*_{P_t,P_r}$ into a valid signature as P_r does not know α (see condition (i) of AS). Moreover, it cannot forge the signature, revert the one-way function or decrypt c_α. Instead, P_r randomizes both elements of ℓ into ℓ', and sends it to P_s, thereby triggering the start of the puzzle solver protocol.

The puzzle solver protocol takes as input an update on channel $(P_s, P_t)^{x_s, x_t}$ and the aforementioned randomized puzzle ℓ'. First, P_s further randomizes ℓ' into ℓ'' to preserve its own anonymity and sends ℓ'' to P_t. If P_s does not randomize the ciphertext, a malicious P_t colluding with P_r can learn the identity of P_s (e.g., P_r reveals the pair (c_α, c'_α) to P_t). We note that the mentioned attack only makes sense in a scenario where P_s wants to pay without revealing her true identity (e.g., if P_s is a Tor user and wants to perform an anonymous donation). For more information about this attack we refer the reader to Appendix E. Next, P_s and P_t compute the conditional update of their channel as $\left\{\perp, \sigma^*_{P_s,P_t}\right\} \leftarrow \langle AS_{P_s}((P_s, P_t)^{x_s, x_t}, \hat{\alpha''}), AS_{P_t}((P_s, P_t)^{x_s, x_t}, \hat{\alpha''})\rangle$. This is where the element c''_α of the puzzle is crucial as it allows P_t to decrypt it and obtain α'', the doubly randomized version of the value α (i.e., the secret required by P_r to complete the update on channel $(P_t, P_r)^{x'_t, x'_r}$). As α'' is randomized, P_t cannot link it to P_r and yet can use it to convert $\sigma^*_{P_s,P_t}$ into a valid signature for the update of $(P_s, P_t)^{x_s, x_t}$. Interestingly, the condition (ii) of AS allows P_s to learn the randomized secret α''. Then, P_s can remove its part of the randomness in α'' and send the result to P_r, who can also remove its part of the randomness, and thereby getting the original value α, which it uses to convert the "almost valid" signature $\sigma^*_{P_t,P_r}$ into a fully valid one.

2.1.2 Trilero. The A^{2}L primitive is agnostic of the actual content of the channel updates and it does not provide any timelock mechanism. Trilero realizes a PCH by augmenting A^{2}L to provide the two aforementioned functionalities, which are crucial for a PCH. First, Trilero ensures that channel updates reflect a payment from P_s to P_r through P_t for a fixed amount amt of coins, thereby ensuring that parties have an economical incentive to execute A^{2}L. Second, Trilero implements a timelock mechanism to return the coins to the initial owners if a payment operation fails (e.g., a party does not answer), which ensures that the honest users do not lose coins or that coins do not get infinitely locked.

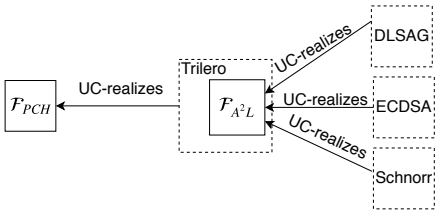

Figure 2: Our modeling for PCH. Dotted boxes represent protocols and solid boxes represent ideal functionalities.

3 SECURITY AND PRIVACY MODEL

3.1 Preliminaries

We define our security and privacy model modularly, as shown in Figure 2, by leveraging the Universal Composability (UC) framework from Canetti [7]. We first describe the payment-channel hub (PCH) ideal functionality $\mathcal{F}_{PCH}$ covering the security and privacy notions for a PCH. Later, in Section 4, we present Trilero, our PCH implementation that UC-realizes $\mathcal{F}_{PCH}$. Trilero relies on another ideal functionality $\mathcal{F}_{\mathsf{A^2L}}$, which captures the expected behavior as well as the security and privacy properties of the interaction among sender, receiver, and tumbler. In Section 4 we show two instantiations of A^{2}L based on Schnorr and ECDSA. We defer our DLSAG-based instantiation to Appendix F.

Attacker Model. We model the parties as interactive Turing machines, which communicate with a trusted functionality $\mathcal{F}$ via secure and authenticated communication channels. We model the adversary $\mathcal{A}$ as a PPT machine. The adversary can corrupt a party P through an interface corrupt$(\cdot)$ that takes as input a party identifier P and provides the attacker with its internal state. Furthermore, all subsequent incoming and outgoing communication of P is routed through $\mathcal{A}$. As commonly done in the literature [20, 27, 38, 39], we consider the static corruption model, that is, the adversary commits to the identifiers of the parties it wishes to corrupt ahead of time.

Communication Model. Communication happens through the secure transmission functionality $\mathcal{F}_{\mathsf{smt}}$ [7] which informs the adversary whenever a communication between any two parties happens, and allows the adversary to delay the delivery of the messages arbitrarily. However, the adversary cannot read nor change the content of the messages. Additionally, we assume the existence of an anonymous communication channel as defined in [6], which we denote here as $\mathcal{F}_{\mathsf{anon}}$. For readability, we use the interfaces $\mathsf{send}_{\mathsf{smt}}$ and $\mathsf{receive}_{\mathsf{smt}}$ to exchange messages through the $\mathcal{F}_{\mathsf{smt}}$ functionality, and interfaces $\mathsf{send}_{\mathsf{anon}}$ and $\mathsf{receive}_{\mathsf{anon}}$ to exchange messages via $\mathcal{F}_{\mathsf{anon}}$. When sending of a message is not annotated with smt or anon, we assume by default that it is of the form $\mathsf{send}_{\mathsf{smt}}$.

We consider a synchronous communication network, where communication proceeds in discrete rounds, as defined in [30] and denoted here as $\mathcal{F}_{\mathsf{syn}}$. The parties are always aware of the current round, and if a party P sends a message in round r, the recipient party receives the message in the beginning of round $r + 1$. The adversary can change the order of messages, but we assume that the order of messages between honest parties cannot be changed (which

For brevity we write $\mathcal{F}$ for $\mathcal{F}_{\mathsf{PCH}}$, and denote Tumbler as P_T. We assume that the channel and promise identifiers are unique and generated at random. Additionally, there exists a lock randomizer function rand, and all the promises use a constant amount (amt).

Open Channel On input $(\text{pc-open}, \text{sid}, (P, P_t)^{x_P, x_{P_t}})$ from a party P, $\mathcal{F}$ checks whether $(P, P_t)^{x_P, x_{P_t}}$ is present in $\mathcal{C}$. If it is present, then $\mathcal{F}$ sends $(\text{pc-exists}, \text{sid}, \bot)$ to P, otherwise it sends $(\text{pc-request}, \text{sid}, (P, P_t)^{x_P, x_{P_t}})$ to P_t, who can either abort or authorize the operation. In the latter case, $\mathcal{F}$ receives $(\text{pc-open}, \text{sid}, (P, P_t)^{x_P, x_{P_t}})$ from P_t and checks whether $\mathcal{B}[P] \geq x_P$ and $\mathcal{B}[P_t] \geq x_{P_t}$. If the checks pass, $\mathcal{F}$ sends (remove, P, x_P) and $(\text{remove}, P_t, x_{P_t})$ to $\mathcal{F}_{\mathsf{B}}$. Lastly, $\mathcal{F}$ sends $(\text{pc-opened}, \text{sid}, (P, P_t)^{x_P, x_{P_t}})$ to P and P_t. Otherwise, channel opening fails and $\mathcal{F}$ sends $(\text{pc-failed}, \text{sid}, \bot)$ to P and P_t.

Pay On input $(\text{promise-request}, \text{sid}, (P_t, P_r)^{x_{P_t}, x_{P_r}})$ from a party P_r, $\mathcal{F}$ sends $(\text{create-promise}, \text{sid}, \varsigma)$ to P_t, who can either abort or authorize the operation. In the former case, $\mathcal{F}$ receives $(\text{promise}, \text{sid}, \bot)$ from P_t, and sends $(\text{promise-failed}, \text{sid}, \bot)$ to P_r. In the latter case, $\mathcal{F}$ receives $(\text{promise}, \text{sid}, \top)$ from P_t, and checks whether $x_{P_t} \geq \text{amt}$. If the condition is not satisfied it sends $(\text{promise-failed}, \text{sid}, \bot)$ to P_r and P_t. Otherwise, it generates and stores $\Pi := (\text{pid}, \text{lid}, \text{cid}, \nu, P_r)$ in $\mathcal{P}$, for a random but unique $\Pi.\text{pid}$ and $\Pi.\text{lid}$, a channel identifier $\Pi.\text{cid} = \text{id}((P_t, P_r)^{x_{P_t}, x_{P_r}})$, a validity period $\Pi.\nu$, and sends $(\text{promise-created}, \text{sid}, \Pi)$ to P_r.

On input $(\text{solve-request}, \text{sid}', \text{lid}, (P_s, P_t)^{x_{P_s}, x_{P_t}})$ from P_s, $\mathcal{F}$ sends $(\text{solve-promise}, \text{sid}', P_s, \text{rand}(\text{lid}), (P_s, P_t)^{x_{P_s}, x_{P_t}})$ to P_t, who can either abort or authorize the operation. In the former case, $\mathcal{F}$ receives $(\text{solve}, \text{sid}', \bot)$ from T and sends $(\text{solve-failed}, \text{sid}', \bot)$ to P_s. In the latter case, $\mathcal{F}$ receives $(\text{solve}, \text{sid}', \varrho)$ from P_t. At this point, $\mathcal{F}$ checks the following conditions: 1) there is an entry $\Pi \in \mathcal{P}$, such that $\Pi.\text{lid} = \text{lid}$ and $\Pi.\nu \geq \Delta$ (i.e., promise has not expired), 2) ϱ is a valid solution to the puzzle $\Pi.\text{lid}$, and 3) $x_{P_s} \geq \text{amt}$. If the conditions are satisfied, then $\mathcal{F}$ updates $(P_s, P_t)^{x_{P_s}, x_{P_t}}$ as $(P_s, P_t)^{x_{P_s} - \text{amt}, x_{P_t} + \text{amt}}$. Also, updates $(P_t, P_r)^{x_{P_t}, x_{P_r}}$ as $(P_t, P_r)^{x_{P_t} - \text{amt}, x_{P_r} + \text{amt}}$, where $\text{id}((P_t, P_r)^{x_{P_t}, x_{P_r}}) = \Pi.\text{cid}$. Lastly, $\mathcal{F}$ removes the entry Π from $\mathcal{P}$, and sends $(\text{solved}, \text{sid}', \top)$ to P_s. Otherwise, if any of the conditions fails, then $\mathcal{F}$ sends $(\text{solve-failed}, \text{sid}', \bot)$ to P_s and P_t.

Close Channel On input $(\text{pc-close}, \text{sid}, \text{cid}')$ from a party P, $\mathcal{F}$ checks whether there exists a payment channel $(P, P_t)^{x_P, x_{P_t}} \in \mathcal{C}$, such that $\text{id}((P, P_t)^{x_P, x_{P_t}}) = \text{cid}'$. If no such channel exists, $\mathcal{F}$ ignores the message. Otherwise, $\mathcal{F}$ checks whether there exists a $\Pi \in \mathcal{P}$, such that $\Pi.\text{cid} = \text{id}((P, P_t)^{x_P, x_{P_t}})$ and $\Pi.\nu \geq \Delta$ (i.e., a promise has not expired). If such a Π exists, then $\mathcal{F}$ removes Π from $\mathcal{P}$. Then, $\mathcal{F}$ sends (add, P, x_P) and $(\text{add}, P_t, x_{P_t})$ to $\mathcal{F}_{\mathsf{B}}$. Lastly, $\mathcal{F}$ removes ς from $\mathcal{C}$, and sends $(\text{pc-closed}, \top)$ to P and P_t.

Figure 3: Ideal functionality $\mathcal{F}_{\mathsf{PCH}}$ in the $(\mathcal{F}_{\mathsf{B}}, \mathcal{F}_{\mathsf{smt}}, \mathcal{F}_{\mathsf{syn}})$-hybrid model.

can easily be realized using message counters). For simplicity, we assume that computation is instantaneous.

Furthermore, as inputs of parties and the messages they send to our ideal functionalities do not contain any private information, we implicitly assume that the ideal functionalities forward all messages they receive to the simulator $\mathcal{S}$. We note that this is merely for ease of exposition, and it is commonly done in the literature [19, 21].

Ledger Functionality (Coins). We assume the existence of a blockchain $\mathcal{B}$ modeled as a trusted append-only bulletin board. The corresponding ideal functionality $\mathcal{F}_{\mathsf{B}}$, as defined in [20], is used to store and update the balance of every party. It is defined in the global UC (GUC) model [8], since it provides values that should be globally accessible, and it can be updated by multiple instances of our ideal functionality or by other protocols simultaneously. In order to update the balance of a party P, $\mathcal{F}_{\mathsf{B}}$ processes the messages (add, P, x) and (remove, P, x), which allow for adding/removing x coins to/from a party P's account, respectively. For readability we write the balance of a party P in $\mathcal{B}$ as $\mathcal{B}[P]$, we denote the number of entries in $\mathcal{B}$ as $|\mathcal{B}|$, and we model time as the number of entries of the blockchain $\mathcal{B}$ (i.e., time $\Delta = |\mathcal{B}|$).

Universal Composability. We now review the notion of secure realization in the UC framework. Intuitively, a protocol realizes an ideal functionality if the adversary has no way to distinguish between the ideal functionality and the real-world protocol, where a simulator translates the messages produced by the ideal functionality for the computational adversary. Here $\mathsf{EXEC}_{\pi, \mathcal{A}, \mathcal{E}}$ denotes the ensemble of the outputs of the environment $\mathcal{E}$ when interacting with the adversary $\mathcal{A}$ and users running protocol π.

Definition 3.1 (UC-realization). A protocol π 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 $\mathsf{EXEC}_{\pi, \mathcal{A}, \mathcal{E}}$ and $\mathsf{EXEC}_{\mathcal{F}, \mathcal{S}, \mathcal{E}}$ are computationally indistinguishable.

3.2 Payment Channel Hub (PCH)

Definition. Here, we formalize the notion of payment channel hub (PCH) and we defer the definition of correctness to Appendix A.

Definition 3.2 (Payment Channel Hub (PCH)). A PCH is defined as a graph $\mathbb{G} := (\mathbb{V}, \mathbb{E})$ equipped with the following operations:

- $\{(P_1, P_2)^{x_{P_1}, x_{P_2}}, \bot\} \leftarrow \text{OpenChannel}(P_1, P_2, x_{P_1}, x_{P_2})$: On input two parties $P_1, P_2 \in \mathbb{V}$, and two amounts x_{P_1}, x_{P_2}, if the operation is authorized by both parties, $\mathcal{B}[P_1] \geq x_{P_1}$ and $\mathcal{B}[P_2] \geq x_{P_2}$, OpenChannel creates a new payment channel $(P_1, P_2)^{x_{P_1}, x_{P_2}}$ and adds it to $\mathbb{E}$. Additionally, OpenChannel updates the blockchain as $\mathcal{B}[P_1] = \mathcal{B}[P_1] - x_{P_1}$ and $\mathcal{B}[P_2] = \mathcal{B}[P_2] - x_{P_2}$ and returns $\text{id}((P_1, P_2)^{x_{P_1}, x_{P_2}})$. Otherwise, OpenChannel returns $\bot$.
- $\{1, 0\} \leftarrow \text{CloseChannel}((P_1, P_2)^{x_{P_1}, x_{P_2}})$: On input a channel $(P_1, P_2)^{x_{P_1}, x_{P_2}}$, if it does not exist in $\mathbb{E}$, CloseChannel returns 0. Otherwise, CloseChannel updates $\mathcal{B}$ as $\mathcal{B}[P_1] = \mathcal{B}[P_1] + x_{P_1}$

KGen(sid)

Upon invocation by P_i, where $P_i \in \{P_s, P_r\}$:

send$_{\text{smt}}$ (sid, P_i) to P_t

receive$_{\text{smt}}$ (sid, b) from P_t

if $b = \perp$ then send (sid, $\perp$) to P_i and abort

else send$_{\text{smt}}$ (sid, P_i, P_t) to P_i

Open(sid, Π, ϱ')

Upon invocation by P_r:

if $\Pi = \perp$ or $\varrho' = \perp$ then abort

set $\varrho \leftarrow$ derand(ϱ')

if $\exists(\Pi^*, -, \Theta^*, \varrho^*, P_i^*) \in \mathcal{L}$ such that $\Pi^* = \Pi$ and $\varrho^* = \varrho$ and $P_i^* = P_r$, then send (sid, Θ^*) to P_r

else send (sid, $\perp$) to P_r and abort

PuzzleSolver(sid, ℓ, P_r)

Upon invocation by P_s:

if $\ell = \perp$ then abort

set $\ell' \leftarrow$ rand(ℓ)

send$_{\text{smt}}$ (sid, ℓ') to P_t

receive$_{\text{smt}}$ (sid, ϱ') from P_t

if $\varrho' = \perp$ or $\nexists(-, \ell^*, -, \varrho^*, P^*) \in \mathcal{L}$ such that $\ell^* = \ell$ and $\varrho^* =$ derand(ϱ') and $P^* = P_r$ then send$_{\text{smt}}$ (sid, $\perp$) to P_s and abort

else send$_{\text{smt}}$ (sid, ϱ') to P_s

and send$_{\text{anon}}$ (sid, ϱ') to P_r

PuzzlePromise(sid, P_s)

Upon invocation by P_r:

send$_{\text{smt}}$ (sid, P_r) to P_t

receive$_{\text{smt}}$ (sid, Π, ℓ, ϱ, Θ) from P_t

if $\Pi = \perp$ or $\ell = \perp$ or $\varrho = \perp$ or $\Theta = \perp$ then abort

insert (Π, ℓ, Θ, ϱ, P_r) into $\mathcal{L}$

send$_{\text{smt}}$ (sid, Π, ℓ) to P_r

send$_{\text{anon}}$ (sid, ℓ, P_r) to P_s

Verify(sid, Π, Θ)

Upon invocation by P_r:

if $\Pi = \perp$ or $\Theta = \perp$ then abort

if $\exists(\Pi^*, -, \Theta^*, \varrho^*, P_i^*) \in \mathcal{L}$ such that $\Pi^* = \Pi$ and $\Theta^* = \Theta$ and $P_i^* = P_r$, then send (sid, $\top$) to P_r

else send (sid, $\perp$) to P_r

Figure 4: Ideal functionality for $\mathcal{F}_{\text{A}^2\text{L}}$ construction.

and $\mathcal{B}[P_2] = \mathcal{B}[P_2] + x_{P_2}$, removes $(P_1, P_2)^{x_{P_1}, x_{P_2}}$ from $\mathbb{E}$ and returns 1.

- $\{1, 0\} \leftarrow$ Pay(P_s, P_r, amt) : On input two parties P_s and P_r, let $(P_s, P_t)^{x_{P_s}, x_{P_t}}$ and $(P_t, P_r)^{x'_{P_t}, x'_{P_r}}$ be the corresponding channels in $\mathbb{E}$. Then if $x_{P_s} \geq \text{amt}$ and $x'_{P_t} \geq \text{amt}$, Pay updates the channels as $(P_s, P_t)^{x_{P_s} - \text{amt}, x_{P_t} + \text{amt}}$ and $(P_t, P_r)^{x'_{P_t} - \text{amt}, x'_{P_r} + \text{amt}}$ and returns 1. Otherwise, no channel is updated and returns 0.

Data Structures. In order to simplify the exposition of our ideal functionality, we define the following data structures:

- List of promises $\mathcal{P}$, which keeps track of the currently existing promises. Each entry has the format (pid, lid, cid, ν, P_i), where pid is a promise identifier, lid is a lock identifier, cid is the channel identifier, ν is a validity period (expiration time) of the promise, and P_i is the party to whom the promise and lock are given. We note that pid and lid are unique identifiers, and each promise has a validity period defined as $\nu = \Delta + \upsilon$ for a constant value υ.
- List of open channels $\mathcal{C}$, which keeps track of the currently open channels. Each channel is defined as $(P_1, P_2)^{x_1, x_2}$ and identified by cid as the channel identifier. For simplicity, we assume a function id that returns the cid corresponding to $(P_1, P_2)^{x_1, x_2}$. We assume that one of the parties in each channel is always the Tumbler (P_t) as the intermediary required in any PCH.

Ideal Functionality. We use $\mathcal{F}_{\text{B}}$, $\mathcal{F}_{\text{smt}}$, and $\mathcal{F}_{\text{syn}}$, thus, our ideal functionality $\mathcal{F}_{\text{PCH}}$ is defined in the ($\mathcal{F}_{\text{B}}$, $\mathcal{F}_{\text{smt}}$, $\mathcal{F}_{\text{syn}}$)-hybrid model. We describe $\mathcal{F}_{\text{PCH}}$ in Figure 3. Additionally, $\mathcal{F}_{\text{PCH}}$ manages a list $\mathcal{C}$ (initially set to $\mathcal{C} := \emptyset$), to keep track of open payment channels.

In our model, every payment transfers a fixed amount amt of coins, which we assume is globally available to all parties. If $\mathcal{F}_{\text{PCH}}$ observes a payment amount other than amt, it aborts. In order to simplify the model we do not include any transaction fees, but we note that our protocol retains its security and privacy properties even in the presence of constant transaction fees[3].

[3]In case of transaction fees, we can define our constant amt as amt = transaction + fee, for some constants transaction and fee. If one does not use constant amounts, then it becomes trivial to link the transacting parties.

OpenChannel and CloseChannel operations are the standard channel operations [1, 38]. Pay is divided into two suboperations: (i) a promise operation where a unique lock lid is generated modeling the puzzle that P_r must solve to get amt coins from P_t; (ii) a solve operation where P_s uses a randomized version of lid to trigger an atomic transfer of amt coins from P_s to P_r through P_t.

Discussion. We discuss here how the ideal functionality captures the security and privacy notions of interest for payment hubs.

Balance Security: The system should not be exploited to print new money or steal existing money, even when parties collude. $\mathcal{F}_{\text{PCH}}$ provides balance security as the only place where the balances are updated is inside the payment operation, and it makes sure that either all the balances are updated or none. Additionally, it assures that the balances are updated only if the correct solution for a lock is provided by the Tumbler. The atomicity and correctness properties are enough to ensure balance security.

Unlinkability: The intermediary should not learn information that allows it to associate the sender and the receiver of a payment. We argue unlinkability in terms of *interaction multi-graph* as defined in [27]. An interaction multi-graph is a mapping of payments from senders to receivers. For each successful payment completed upon a query from sender S_j at time t, this graph has an edge, labeled with time t from the sender S_j to the tumbler and an additional edge also labeled with time t from tumbler to the receiver R_i. An interaction graph is *compatible* if it explains the view of the tumbler, that is, the number of edges labeled with t incident on R_i equals the number of edges labeled with t from S_j. Then, unlinkability requires all compatible interaction graphs to be equally likely. The anonymity set depends thus on the number of compatible interaction graphs. We defer a further discussion to Appendix E. $\mathcal{F}_{\text{PCH}}$ achieves unlinkability while it uses constant amounts and random but unique identifiers for locks, which gets rerandomized before reaching the Tumbler.

3.3 Anonymous Atomic Lock (A^{2}L)

Definition. Here, we formalize the notion of anonymous atomic locks (A^{2}L) and we defer the correctness definition to Appendix A.

Definition 3.3 (Anonymous Atomic Lock (A^{2}L)). An A^{2}L consists of the following protocols:

- $\{(\mathsf{sk}_t, \mathsf{pk}_{i,t}), (\mathsf{sk}_i, \mathsf{pk}_{i,t})\} \leftarrow \langle \mathsf{KGen}_{P_t}(1^\lambda), \mathsf{KGen}_{P_i}(1^\lambda)\rangle$: On input the security parameter 1^λ, KGen returns a shared public key $\mathsf{pk}_{i,t}$ and a secret key sk_t (resp. sk_i) to P_t (resp. P_i).
- $\{\cdot, (\Pi, \ell)\} \leftarrow \langle \mathsf{PuzzlePromise}_{P_t}(\mathsf{sk}_t, \mathsf{pk}_{r,t}), \mathsf{PuzzlePromise}_{P_r}(\mathsf{sk}_r, \mathsf{pk}_{r,t})\rangle$: On input two secret keys $\mathsf{sk}_t, \mathsf{sk}_r$, and a public key $\mathsf{pk}_{r,t}$, the PuzzlePromise protocol is executed between two parties (namely, P_t and P_r), and it returns a promise Π and a lock ℓ to P_r.
- $\{\varrho, \cdot\} \leftarrow \langle \mathsf{PuzzleSolver}_{P_s}(\mathsf{sk}_s, \mathsf{pk}_{s,t}, \ell), \mathsf{PuzzleSolver}_{P_t}(\mathsf{sk}_t, \mathsf{pk}_{s,t})\rangle$: On input two secret keys sk_s and sk_t, a public key $\mathsf{pk}_{s,t}$, and a lock ℓ, the PuzzleSolver protocol is executed between two parties (namely, P_s and P_t) and it returns the solution ϱ of the lock ℓ to P_s.
- $\Theta \leftarrow \mathsf{Open}(\Pi, \varrho)$: On input a promise Π and a solution ϱ, the opening algorithm returns the promise fullfillment Θ.
- $\{0, 1\} \leftarrow \mathsf{Verify}(\Pi, \Theta)$: On input a promise Π and a promise fullfillment Θ, the verification algorithm returns a bit $b \in \{0, 1\}$.

Ideal Functionality. We illustrate the ideal functionality for A^{2}L in Figure 4. We use $\mathcal{F}_{\mathsf{anon}}$, $\mathcal{F}_{\mathsf{smt}}$ and $\mathcal{F}_{\mathsf{syn}}$, thus, our functionality is defined in the $(\mathcal{F}_{\mathsf{anon}}, \mathcal{F}_{\mathsf{smt}}, \mathcal{F}_{\mathsf{syn}})$-hybrid model. Additionally, $\mathcal{F}_{\mathsf{A^2L}}$ manages a list $\mathcal{L}$ (initially set to $\mathcal{L} := \emptyset$), to keep track of each cryptographic puzzle and its corresponding solution. The entries in the list have the format $(\Pi, \ell, \Theta, \varrho, P_i)$, where Π is a promise, ℓ is a lock, Θ is a promise fullfillment, ϱ is the solution for the lock, and P_i is the party involved in the promise with the tumbler P_t. Additionally, for clarity of exposition, we denote by $\mathsf{rand}(\cdot)$ and $\mathsf{derand}(\cdot)$ the randomization and the corresponding de-randomization functions, which given as input a (possibly randomized) value, return the (de-)randomized version of it. These functions are used inside the PuzzlePromise and PuzzleSolver interfaces as defined in Figure 4.

$\mathcal{F}_{\mathsf{A^2L}}$ provides five interfaces. The KGen interface allows the tumbler and the other party to establish a link between themselves. The PuzzlePromise interface allows a party to obtain a promise and a lock from the tumbler. The PuzzleSolver interface allows a party to acquire the solution of a given lock. The Open interface allows a party to fulfill a promise. Finally, the Verify interface verifies that the promise and the promise fullfillment match each other.

Alternative Approaches for Ideal Functionality. Naturally, there exist alternative approaches to model anonymous atomic locks. We could define two separate ideal functionalities, one for the puzzle promise phase and the other one for the puzzle solver phase, similar to the model in [27]. However, the ideal functionalities in [27] by themselves only satisfy a property that is called fairness, which is analogous to our atomicity notion. In order to achieve any meaningful privacy notion one has to assume that the puzzle input given to their payment functionality is blinded beforehand. In contrast, our ideal functionality achieves unlinkability by design, as a lock gets randomized inside the ideal functionality itself.

Another possible approach is to construct a single 2-of-2 signature ideal functionality, and then instantiate it with different signatures that satisfy the desired properties. However, it is harder to fit this approach into our setting, as our primary property is unlinkability, hence, we need a way to correlate the signature from the promise protocol with the signature from the payment protocol in an unlinkable fashion. It is not obvious how to do this with a 2-of-2 signature functionality.

Discussion. We introduce the security and privacy notions of interest for our $\mathcal{F}_{\mathsf{A^2L}}$ functionality.

Atomicity: Loosely speaking, the system should ensure that a lock can only be opened if there has been a payment for it before. This protects the tumbler from a malicious receiver. This is enforced by $\mathcal{F}_{\mathsf{A^2L}}$ because it keeps track of each promise along with the solution of the lock and the promise fullfillment. $\mathcal{F}_{\mathsf{A^2L}}$ checks whether the solution given to the Open interface corresponds to one of the existing entries in the list $\mathcal{L}$. Since a party obtains a solution only from a call to the PuzzleSolver interface and $\mathcal{F}_{\mathsf{A^2L}}$ is trusted, this ensures that PuzzleSolver is called before Open in order for Open to succeed.

Additionally, the system should ensure that if a payment can be received by the tumbler then the receiver can fullfil a matching promise previously issued by the tumbler. This protects the sender from a malicious tumbler. Assume that the PuzzleSolver interface is invoked on a lock ℓ previously issued by a PuzzlePromise. If $\mathcal{F}_{\mathsf{A^2L}}$ does not abort, then $\mathcal{F}_{\mathsf{A^2L}}$ ensures that it returns the correct solution ϱ matching the promise Π. In other words, if Open is invoked on input Π and ϱ, $\mathcal{F}_{\mathsf{A^2L}}$ ensures there is an entry in $\mathcal{L}$ containing both.

Unlinkability: Intuitively, unlinkability means that the tumbler does not learn information that allows it to associate the sender and the receiver of a payment. This property is enforced by $\mathcal{F}_{\mathsf{A^2L}}$ since the lock ℓ that is created by the tumbler in the PuzzlePromise interface gets randomized by $\mathcal{F}_{\mathsf{A^2L}}$ within the PuzzleSolver interface before it is sent back to the tumbler.

Additionally, since we assume the existence of an anonymous communication channel between parties (i.e., the $\mathcal{F}_{\mathsf{anon}}$ functionality), the intermediary cannot use the network information to correlate sender and receiver. We remark that this assumption is indispensable for unlinkability and is commonly adopted in the PCH-related literature [26, 27].

4 OUR PROTOCOLS

4.1 Trilero: Our PCH Instantiation

System Assumptions. We assume a constant amount for every payment, as otherwise it becomes trivial to link P_s and P_r in a payment. Moreover, as in [27], we assume that the protocols are run in epochs, optimizing the anonymity set within an epoch.

Protocol Description. A^{2}L, in combination, with a blockchain $\mathcal{B}$ can be used to realize a fully-fledged PCH. Here we assume that both the sender (P_s) and receiver (P_r) have already carried out the key generation procedure and have set up the payment channels with the tumbler (P_t). We denote those channels as $(P_s, P_t)^{x_s, x_t}$ and $(P_t, P_r)^{x'_t, x'_r}$, respectively. A payment of amt coins between P_s and P_r through P_t is realized by updating both channels such that P_t gets additional amt coins in $(P_s, P_t)^{x_s, x_t}$ if and only if P_r gets amt coins in $(P_t, P_r)^{x'_t, x'_r}$. In order to ensure this invariant, Trilero relies on two contracts built upon the A^{2}L functionality.

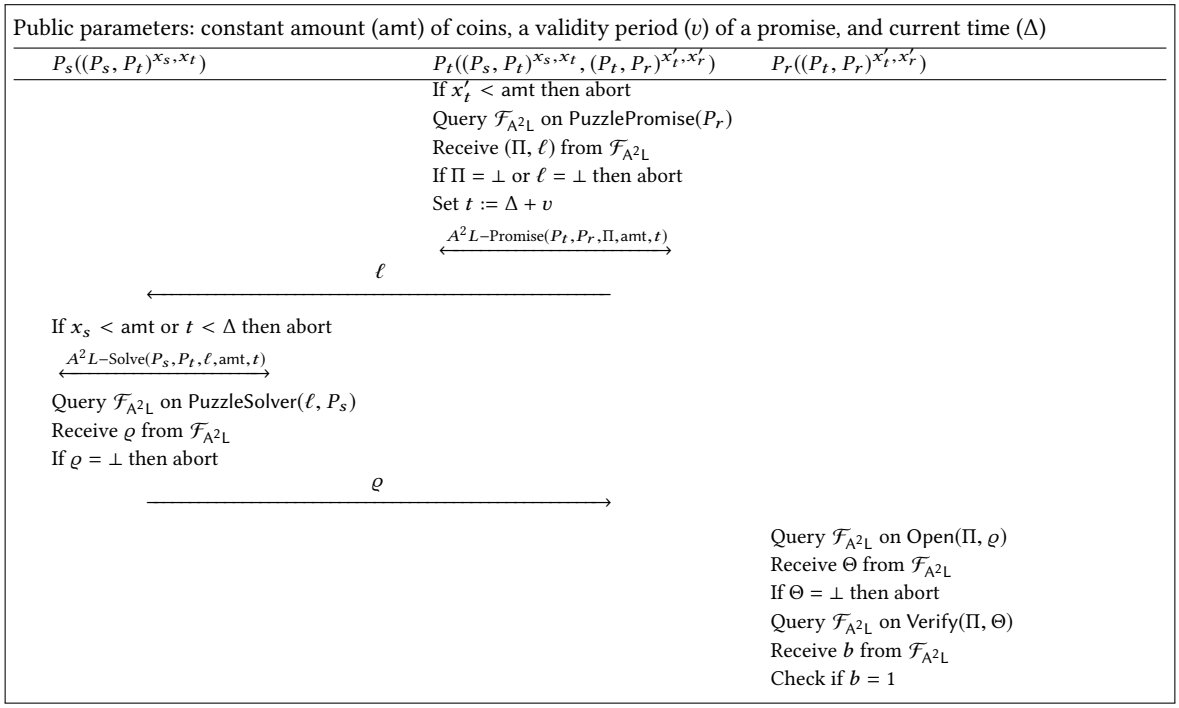

Figure 5: Trilero protocol in the ($\mathcal{F}_{\mathsf{A^2L}}, \mathcal{F}_{\mathsf{B}}, \mathcal{F}_{\mathsf{smt}}, \mathcal{F}_{\mathsf{syn}}$)-hybrid model.

In a bit more detail, first P_t and P_r execute the PuzzlePromise protocol from A^{2}L to get the input required to establish the following A^2L-Promise(P_t, P_r, Π, amt, t) contract:

(1) If P_r produces the promise fullfillment data Θ, so that Verify(Π, Θ) = 1 before time t expires, then $(P_t, P_r)^{x'_t,x'_r}$ is updated as $(P_t, P_r)^{x'_t-\text{amt},x'_r+\text{amt}}$ (i.e., Tumbler pays receiver amt coins).
(2) If timeout t expires, $(P_t, P_r)^{x'_t,x'_r}$ remains unchanged (i.e., Tumbler regains control over amt coins).

Here, Π is the output (along with ℓ) of the PuzzlePromise protocol in A^{2}L, t is an expiration time (validity period) of the promise, which is properly set to give P_r the time it needs to reveal the solution ϱ. In case this does not happen, then P_t gets back the money, thereby avoiding an indefinite locking of money in the channel. Notice that we require that $\mathcal{B}$ supports the Verify algorithm and time management in its scripting language. This is the case in practice as Verify is implemented as the unmodified verification algorithm from Schnorr, ECDSA or DLSAG digital signature scheme, and virtually all cryptocurrencies natively implement a timelock mechanism where time is measured as the number of blocks in the blockchain.

Second, P_r sends the lock ℓ (as output by the PuzzlePromise protocol) to P_s. Then, P_s and P_t execute the PuzzleSolver protocol to get the input required to establish the following A^2L-Solve(P_s, P_t, ℓ, amt, t') contract:

(1) If before t', P_t sends P_s the solution ϱ to the cryptographic challenge encoded in ℓ, $(P_s, P_t)^{x_s,x_t}$ is updated as $(P_s, P_t)^{x_s-\text{amt},x_t+\text{amt}}$ (i.e., the sender pays Tumbler amt coins).
(2) Otherwise, $(P_s, P_t)^{x_s,x_t}$ remains unchanged (i.e., the sender regains control over amt coins).

Finally, P_s gets the solution ϱ to the challenge encoded in the lock ℓ. Then, P_s sends ϱ to P_r who can then complete the A^2L-Promise contract with the promise fullfillment $\Theta := \text{Open}(\Pi, \varrho)$.

Security Analysis. Here we argue that the system as described in Figure 5, UC-realizes the functionality $\mathcal{F}_{\mathsf{PCH}}$ as defined in Figure 3. Due to space constraints, we defer the proof to Appendix D.

Theorem 4.1. *The system described in Figure 5, UC-realizes $\mathcal{F}_{\mathsf{PCH}}$ (as defined in Figure 3) in the ($\mathcal{F}_{\mathsf{A^2L}}, \mathcal{F}_{\mathsf{B}}, \mathcal{F}_{\mathsf{smt}}, \mathcal{F}_{\mathsf{syn}}$)- hybrid model.*

4.2 A^{2}L Instantiations

In this section, we present our A^{2}L instantiations. In particular, we give an overall intuition in Section 2.1, we discuss the building blocks in Section 4.2.1, we detail the Schnorr-based instantiation in Section 4.2.2 and the ECDSA-based instantiation in Section 4.3. Due to lack of space, we defer our DLSAG-based instantiation to Appendix F.

4.2.1 Cryptographic Building Blocks. We denote by 1^λ, for $\lambda \in \mathbb{N}$, the security parameter. We assume that the security parameter is given as an implicit input to every function. We review here the cryptographic primitives used in our protocols.

Commitment Scheme. A commitment scheme COM consists of a commitment algorithm (com, decom) $\leftarrow \mathsf{P}_{\mathsf{COM}}(m)$ and a verification algorithm $\{0, 1\} \leftarrow \mathsf{V}_{\mathsf{COM}}(\text{com}, \text{decom}, m)$. The commitment algorithm allows a prover to commit to a message m without

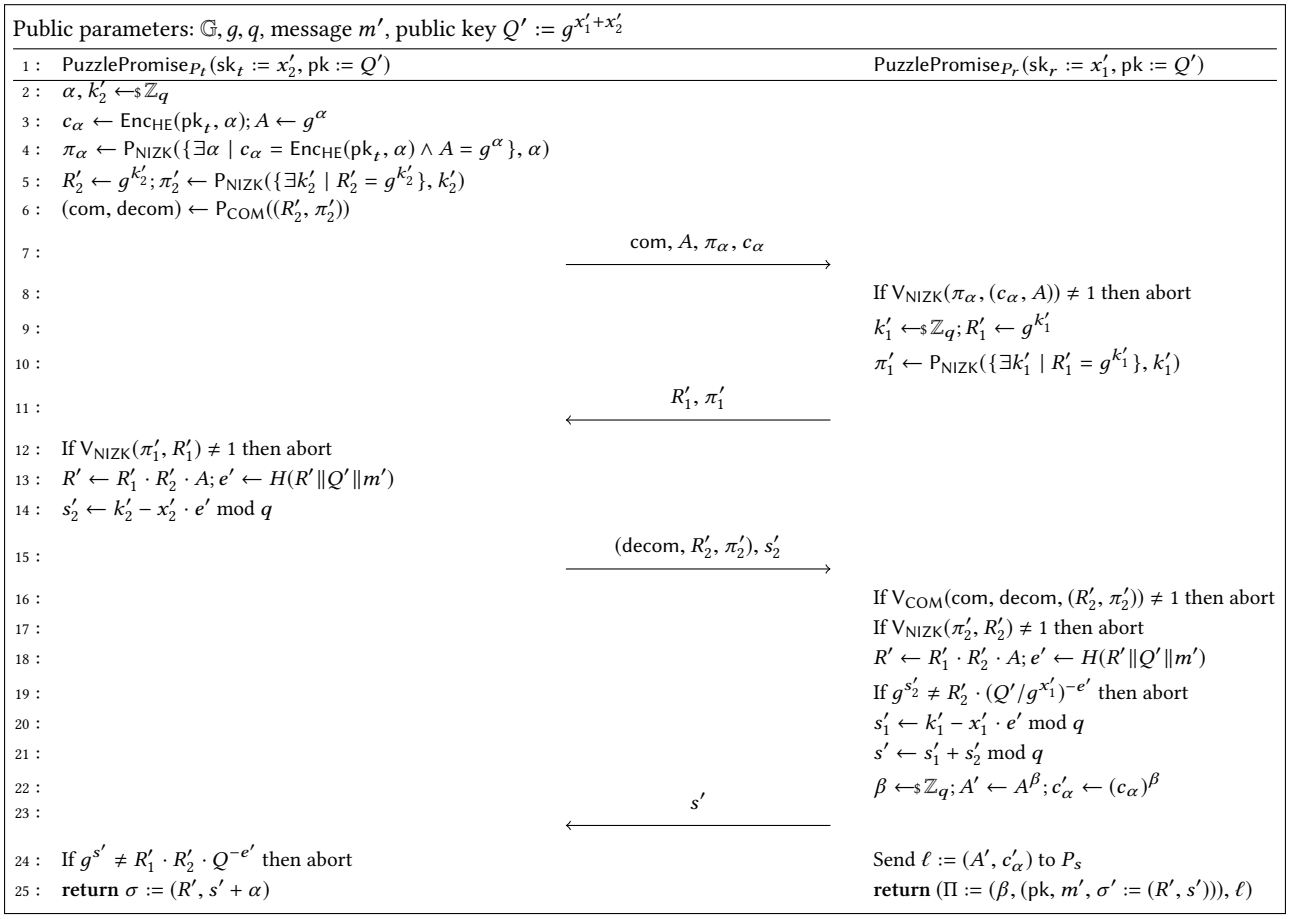

Figure 6: Puzzle promise protocol of Schnorr-based construction.

revealing it. The verification algorithm allows a verifier to be convinced that the message m was committed using the revealed decommitment information decom. The security of COM is modeled by the ideal functionality $\mathcal{F}_{\mathsf{COM}}$ [7].

Non-Interactive Zero-Knowledge. Let R be an NP relation, and let L be a set of positive instances corresponding to the relation R (i.e., $L = \{x \mid \exists w \text{ s.t. } R(x, w) = 1\}$). A non-interactive zero-knowledge proof scheme NIZK [4] consists of a prover algorithm $\pi \leftarrow \mathsf{P}_{\mathsf{NIZK}}(x, w)$ and a verification algorithm $\{0, 1\} \leftarrow \mathsf{V}_{\mathsf{NIZK}}(x, \pi)$. A NIZK scheme allows a prover to convince a verifier about the existence of a witness w for a statement x without revealing any information apart from the fact that it actually knows the witness w. We model the security of a NIZK scheme using the ideal functionality $\mathcal{F}_{\mathsf{NIZK}}$, as described in Appendix C.

Homomorphic Encryption. A linearly homomorphic encryption scheme HE is composed of the algorithms $(\mathsf{KGen}_{\mathsf{HE}}, \mathsf{Enc}_{\mathsf{HE}}, \mathsf{Dec}_{\mathsf{HE}})$, where $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KGen}_{\mathsf{HE}}()$, $c \leftarrow \mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}, m)$, and $m \leftarrow \mathsf{Dec}_{\mathsf{HE}}(\mathsf{sk}, c)$, for a secret/public key pair $(\mathsf{sk}, \mathsf{pk})$, a message m, and a ciphertext c. In our construction, we use the homomorphic encryption scheme by Castagnos-Laguillaumie (CL) [12], more precisely, the variant of the scheme described in [10, 13] (henceforth HSM-CL). It supports homomorphic operations over ciphertexts of the form $\mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}, m_1) \cdot \mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}, m_2) = \mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}, m_1 + m_2)$ and $\mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}, m_1)^{m_2} = \mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}, m_1 \cdot m_2)$. We rely on the assumption that HSM-CL encryption scheme is IND-CPA secure. We refer the reader to Appendix B for more information about the HSM-CL encryption scheme and its advantages over the other homomorphic encryption schemes.

ECDSA Signature. The ECDSA signature scheme is composed of the algorithms $(\mathsf{KGen}_{\mathsf{ECDSA}}, \mathsf{Sig}_{\mathsf{ECDSA}}, \mathsf{Vf}_{\mathsf{ECDSA}})$, where $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KGen}_{\mathsf{ECDSA}}()$, $\sigma := (r, s) \leftarrow \mathsf{Sig}_{\mathsf{ECDSA}}(\mathsf{sk}, m)$, and $\{0, 1\} \leftarrow \mathsf{Vf}_{\mathsf{ECDSA}}(\mathsf{pk}, \sigma, m)$, for a signing/public (verification) key pair $(\mathsf{sk}, \mathsf{pk})$, a message m and a signature σ. We use the two-party ECDSA protocol of Castagnos et al. [10], which provides distributed key generation and signing. An ideal functionality $\mathcal{F}^{\mathsf{ECDSA}}_{\mathsf{KGen}}$ that securely computes distributed key generation for two-party ECDSA is given in Appendix C. A comparison of different threshold ECDSA schemes is discussed in Section 6.

Schnorr Signature. The Schnorr signature scheme is defined using the algorithms $(\mathsf{KGen}_{\mathsf{Schnorr}}, \mathsf{Sig}_{\mathsf{Schnorr}}, \mathsf{Vf}_{\mathsf{Schnorr}})$, such that $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KGen}_{\mathsf{Schnorr}}()$, $\sigma := (e, s) \leftarrow \mathsf{Sig}_{\mathsf{Schnorr}}(\mathsf{sk}, m)$, and $\{0, 1\} \leftarrow \mathsf{Vf}_{\mathsf{Schnorr}}(\mathsf{pk}, \sigma, m)$, for a signing/public (verification) key pair $(\mathsf{sk}, \mathsf{pk})$, a message m and a signature σ. In our protocol we make use of a two-party Schnorr signature with distributed key

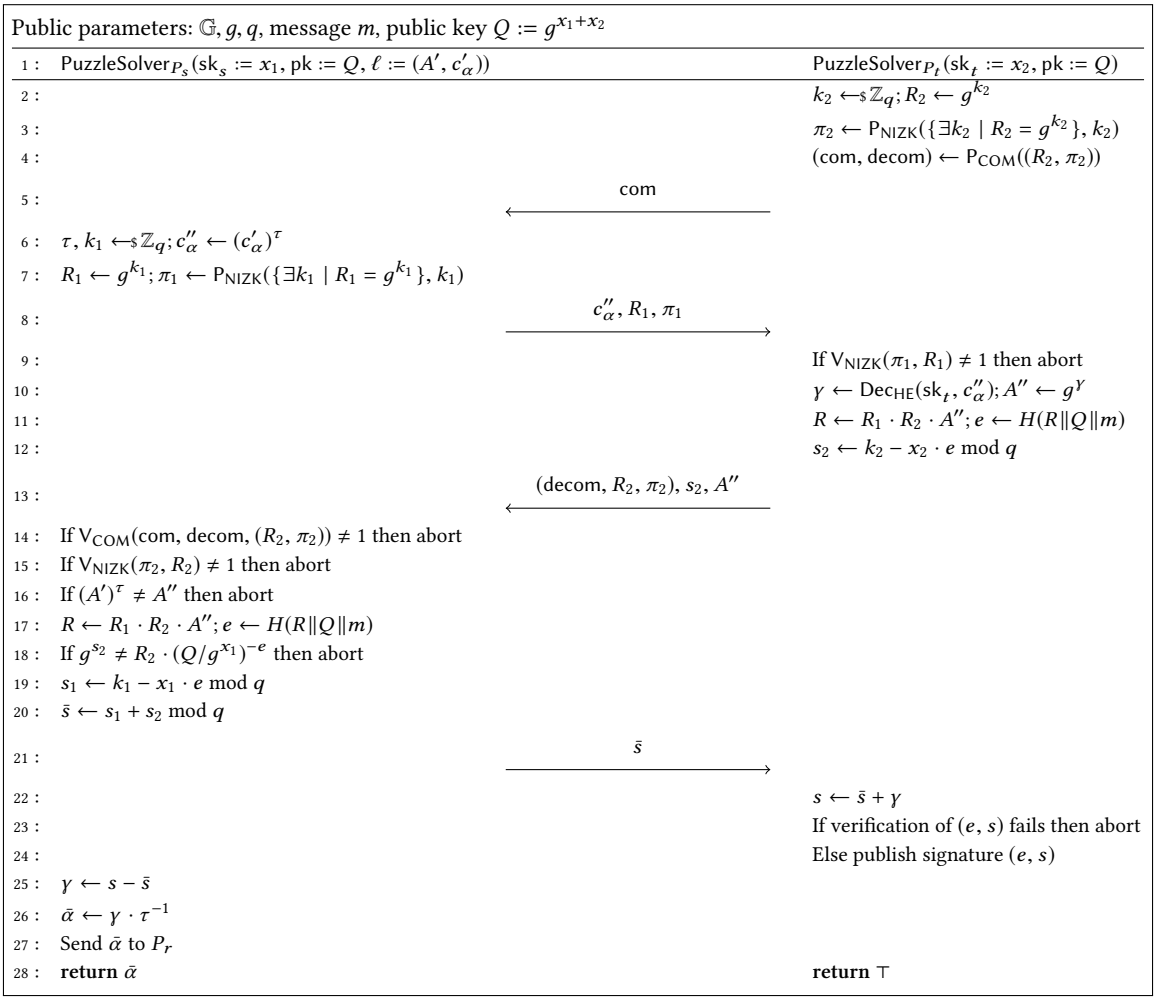

Figure 7: Puzzle solver protocol of Schnorr-based construction.

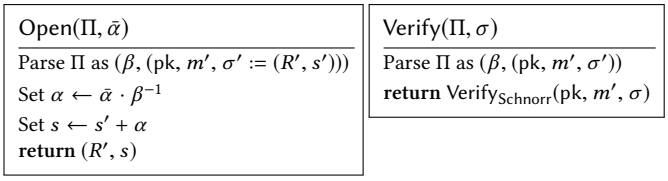

Figure 8: Open and verify algorithms of Schnorr-based construction.

generation and signing. An ideal functionality $\mathcal{F}_{\mathsf{KGen}}^{\mathsf{Schnorr}}$ that securely computes distributed key generation for two-party Schnorr is given in Appendix C.

4.2.2 Schnorr-based Construction. Let $\mathbb{G}$ be an elliptic curve group of prime order q with a generator g, and let $H : \{0,1\}^* \to \mathbb{Z}_q$ be a collision resistant hash function. Additionally, let COM, NIZK and HE be a commitment scheme, a non-interactive zero-knowledge scheme, and a homomorphic encryption scheme, respectively. The Schnorr-based puzzle promise and puzzle solver protocols are shown in Figures 6 and 7, respectively.

Each pair of parties (P_1, P_2) generates a shared Schnorr public key $\mathsf{pk} := g^{x_1+x_2}$ via the $\mathcal{F}_{\mathsf{KGen}}^{\mathsf{Schnorr}}$ ideal functionality, where we assume that $P_2 := P_t$ (tumbler) in both protocols, and $P_1 := P_r$ (receiver) in the puzzle promise protocol, whereas $P_1 := P_s$ (sender) in the puzzle solver protocol. The Schnorr-based distributed key generation functionality $\mathcal{F}_{\mathsf{KGen}}^{\mathsf{Schnorr}}$ is described in Appendix C.

The puzzle promise protocol is run between the parties (P_t, P_r). They initially agree on a message encoding a transaction that transfers coins from P_t to P_r. Additionally, P_t chooses a secret value α, encrypts it under its own public key using the homomorphic encryption scheme, and sends the ciphertext c_α along with $A := g^\alpha$ to P_r (lines 1-7 in Figure 6). Here we require a zero-knowledge proof (denoted by π_α in the puzzle promise protocol) proving that the ciphertext c_α encrypts the discrete logarithm of A (line 4 in Figure 6). If we do not have such a proof, then P_t can perform the following attack to link a potential payer and payee. At a particular epoch, P_t chooses a payee P_r^* it wants to attack, and when

performing the puzzle solver protocol with this party it encrypts a value that is different from the discrete logarithm of A. Then, during the puzzle solver protocol, when a payer P_s^* performs the protocol with P_t, the check $(A')^\tau = A''$ (line 16 in Figure 7) will fail, and P_s^* will cause an abort. Although, in this case (due to our atomicity property) no payment will go through, P_t can still link a payee P_r^* of its choice with its corresponding potential payer P_s^* in a given epoch.

Once this initial setup procedure is prepared by P_t, the parties execute a coin tossing protocol to agree on a randomness $R' = k_1' + k_2' + \alpha$, where α is unknown to P_r. The randomness here is composed additively due to the linear structure of Schnorr. The randomness R' is computed by parties exchanging $g^{k_1'}$ and $g^{k_2'}$, and additionally making use of the value A. The computation of R' together with the corresponding consistency proof is piggybacked in the coin tossing (lines 5-13 in Figure 6). At this point, P_t computes its side of the two-party Schnorr signature, but does not include the secret α into the signature (line 14 in Figure 6). Now, P_r is able to validate this partial signature that it receives from P_t, and also to compute an "almost valid" signature by performing its part of the two-party signature. This means that P_r computes a tuple $(e', s' := k_1' + k_2' - e' \cdot (x_1' + x_2'))$, and that the complete signature is of the form $(e', s' + \alpha)$ (lines 18-21 in Figure 6). However, P_r does not have α, so it cannot complete the signature. Nevertheless, P_r receives $c_\alpha := \mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}_t, \alpha)$ and $A := g^\alpha$ from P_t at the beginning of the puzzle promise protocol, and at the end of the protocol P_r samples a random value β, which it uses to randomize the values as $c_\alpha' := (c_\alpha)^\beta$ and $A' := A^\beta$. This is possible due to the homomorphic properties of the HSM-CL encryption scheme that we are using. The puzzle promise protocol finishes with P_r sending these randomized values to P_s (lines 22 and 24 in Figure 6).

The puzzle solver protocol is executed between the parties (P_s, P_t). At the beginning of the protocol, P_s samples a random value τ, and randomizes the values it received from P_r, as $c_\alpha'' := (c_\alpha')^\tau$ and $A'' := (A')^\tau$ (line 6 in Figure 7). Once this is done, P_s and P_t perform a coin tossing protocol similar to the one performed between P_r and P_t in the puzzle promise protocol, but additionally P_s sends c_α'' to P_t (lines 7-11 in Figure 7). At this point, P_t decrypts c_α'' to obtain the value $\gamma := \alpha \cdot \beta \cdot \tau$ (line 10 in Figure 7). The rest of the protocol continues similar to the puzzle promise protocol, where P_t and P_s compute a common randomness, and then perform a two-party Schnorr signature. However, this time P_t incorporates the decrypted value γ as part of the randomness. After the two-party Schnorr signature completes and P_t publishes it (allowing P_t to receive the payment from P_s), P_s is able to extract the value γ from the published signature (lines 24-25 in Figure 7). It removes her part of the randomization from γ as $\bar{\alpha} := \gamma \cdot \tau^{-1}$, and sends this value to P_r (lines 26-27 in Figure 7), who can also remove its part of the randomization and obtain the initial $\alpha := \bar{\alpha} \cdot \beta^{-1}$. Once P_r obtains α, it can complete the "almost valid" signature that it computed at the end of the puzzle promise protocol, as seen in Figure 8, which allows it to claim the coins that were promised by P_t.

Security Analysis. We define the security of the Schnorr-based construction in Theorem 4.2, and formally prove it in Appendix C.

Theorem 4.2. *Let* COM *be a secure commitment scheme and let* NIZK *be a non-interactive zero-knowledge scheme. If Schnorr signature is strongly existentially unforgeable and HSM-CL encryption is* IND-CPA *secure, then the construction in Figures 6, 7 and 8, UC-realizes the ideal functionality* $\mathcal{F}_{\mathsf{A^2L}}$ *in the* $(\mathcal{F}_{\mathsf{KGen}}^{\mathsf{Schnorr}}, \mathcal{F}_{\mathsf{anon}}, \mathcal{F}_{\mathsf{smt}}, \mathcal{F}_{\mathsf{syn}})$*-hybrid model.*

4.3 ECDSA-based Construction

The ECDSA signature does not have a linear structure as Schnorr, making the design of our protocol more challenging. Let $\mathbb{G}$ be an elliptic curve group of order q with a generator g, and let $H : \{0,1\}^* \to \mathbb{Z}_q$ be a collision resistant hash function. Additionally, let COM, NIZK, and HE be a commitment scheme, a non-interactive zero-knowledge scheme, and a homomorphic encryption scheme, respectively. The ECDSA-based puzzle promise and puzzle solver protocols are shown in Figures 9 and 10, respectively.

Our ECDSA-based instantiation shares similar ideas with our Schnorr-based instantiation. Hence, we only describe the differences compared to the Schnorr-based variant here. Each pair of parties (P_1, P_2) generates a shared ECDSA public key $\mathsf{pk} := g^{x_1 \cdot x_2}$ via the $\mathcal{F}_{\mathsf{KGen}}^{\mathsf{ECDSA}}$ ideal functionality, where, as before, $P_2 := P_t$ (tumbler) in both protocols, whereas $P_1 := P_r$ (receiver) in the puzzle promise protocol and $P_1 := P_s$ (sender) in the puzzle solver protocol. Because ECDSA does not have the linear structure of Schnorr, the distributed key generation is also more convoluted, and it requires additionally exchanging a HSM-CL encrypted secret key. More precisely, P_1 receives a HSM-CL secret key sk and its share x_1, whereas P_2 receives its share x_2 and the HSM-CL encryption c_{key} of x_1. The ECDSA-based distributed key generation functionality $\mathcal{F}_{\mathsf{KGen}}^{\mathsf{ECDSA}}$ is described in Appendix C.

The puzzle promise protocol runs similarly to the Schnorr-based puzzle promise protocol, except that the randomness is composed multiplicatively due to the structure of ECDSA. More precisely, the parties agree on a randomness $R' = k_1' \cdot k_2' \cdot \alpha$, where α is unknown to P_r (lines 4-14 in Figure 9). Once the randomness is computed, P_t performs its side of the two-party ECDSA signature using c'_{key} (the encryption of x_1') and the homomorphic properties of HSM-CL encryption scheme. However, P_t does not embed the inverse of α into the signature (line 16 in Figure 9). Now, P_r is able to compute an "almost valid" signature by decrypting the ciphertext that it received from P_t and performing his part of the signature. This means that P_r computes a tuple $(r', s' := \frac{r' \cdot x_1' \cdot x_2' + H(m')}{k_1' \cdot k_2'})$, and that the complete signature is of the form $(r', s' \cdot \alpha^{-1})$ (lines 21-25 in Figure 9). Since P_r does not have α, he cannot complete the signature. Exactly as in the Schnorr-based construction, P_r receives $c_\alpha := \mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}_t, \alpha)$ and $A := g^\alpha$ from P_t at the beginning of the puzzle promise protocol, and at the end of the protocol P_r samples a random value β, and randomizes the values c_α and A using β. The puzzle promise protocol finishes with P_r sending these randomized values to P_s (lines 25-26 and 28 in Figure 9).

The puzzle solver protocol is similar to Schnorr-based puzzle solver protocol, with the sole difference that P_s and P_t compute a two-party ECDSA signature instead of a two-party Schnorr signature..

Public parameters: $\mathbb{G}, g, q$, message m', public key $Q' := g^{x'_1 \cdot x'_2}$

	$\mathsf{PuzzlePromise}_{P_t}(\mathsf{sk}_t := x'_2, \mathsf{pk} := Q', c'_{key} = \mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}_r, x'_1))$		$\mathsf{PuzzlePromise}_{P_r}(\mathsf{sk}_r := x'_1, \mathsf{pk} := Q')$
2 :	$\alpha, k'_2 \leftarrow_{\$} \mathbb{Z}_q; c_\alpha \leftarrow \mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}_t, \alpha); A \leftarrow g^\alpha$		
3 :	$\pi_\alpha \leftarrow \mathsf{P}_{\mathsf{NIZK}}(\{\exists \alpha \mid c_\alpha = \mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}_t, \alpha) \wedge A = g^\alpha\}, \alpha)$		
4 :	$R'_2 \leftarrow g^{k'_2}; \pi'_2 \leftarrow \mathsf{P}_{\mathsf{NIZK}}(\{\exists k'_2 \mid R'_2 = g^{k'_2}\}, k'_2)$		
5 :	$(\mathsf{com}, \mathsf{decom}) \leftarrow \mathsf{P}_{\mathsf{COM}}((R'_2, \pi'_2))$		
6 :		$\mathsf{com}, A, \pi_\alpha, c_\alpha \longrightarrow$	
7 :			If $\mathsf{V}_{\mathsf{NIZK}}(\pi_\alpha, (c_\alpha, A)) \neq 1$ then abort
8 :			$k'_1 \leftarrow \mathbb{Z}_q$
9 :			$R'_1 \leftarrow g^{k'_1}; \pi'_1 \leftarrow \mathsf{P}_{\mathsf{NIZK}}(\{\exists k'_1 \mid R'_1 = g^{k'_1}\}, k'_1)$
10 :		$\longleftarrow R'_1, \pi'_1$	
11 :	If $\mathsf{V}_{\mathsf{NIZK}}(\pi'_1, R'_1) \neq 1$ then abort		
12 :	$R'_c \leftarrow (R'_2)^\alpha$		
13 :	$\pi'_c \leftarrow \mathsf{P}_{\mathsf{NIZK}}(\{\exists \alpha \mid R_c = (R'_2)^\alpha\}, \alpha)$		
14 :	$\pi'_a \leftarrow \mathsf{P}_{\mathsf{NIZK}}(\{\exists \alpha \mid A = g^\alpha \wedge R_c = (R'_2)^\alpha\}, \alpha)$		
15 :	$R' \leftarrow (R'_1)^{k'_2 \cdot \alpha}; R' := (r'_x, r'_y)$; Set $r' \leftarrow r'_x \bmod q$		
16 :	$c' \leftarrow \mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}_r, (k'_2)^{-1} \cdot r' \cdot x'_1 \cdot x'_2 + (k'_2)^{-1} \cdot H(m') + \rho q)$		
17 :		$(\mathsf{decom}, R'_2, \pi'_2), c', R'_c, \pi'_c, \pi'_a \longrightarrow$	
18 :			If $\mathsf{V}_{\mathsf{COM}}(\mathsf{com}, \mathsf{decom}, (R'_2, \pi'_2)) \neq 1$ then abort
19 :			If $\mathsf{V}_{\mathsf{NIZK}}(\pi'_2, R'_2) \neq 1 \vee \mathsf{V}_{\mathsf{NIZK}}(\pi'_c, R'_c) \neq 1$
20 :			$\vee\, \mathsf{V}_{\mathsf{NIZK}}(\pi'_a, (A, R'_c)) \neq 1$ then abort
21 :			$R' \leftarrow (R'_c)^{k'_1}; R' := (r'_x, r'_y)$
22 :			Set $r' \leftarrow r'_x \bmod q$
23 :			$s'_2 \leftarrow \mathsf{Dec}_{\mathsf{HE}}(\mathsf{sk}_r, c')$
24 :			If $(R'_2)^{s'_2 \bmod q} \neq (Q')^{r'} \cdot g^{H(m')}$ then abort
25 :			$s' \leftarrow s'_2 \cdot (k'_1)^{-1} \bmod q; \beta \leftarrow_{\$} \mathbb{Z}_q$
26 :			$A' \leftarrow A^\beta; c'_\alpha \leftarrow (c_\alpha)^\beta$
27 :		$\longleftarrow s'$	
28 :	If $(R'_1)^{k'_2 \cdot s'} \neq (Q')^{r'} \cdot g^{H(m')}$ then abort		Send $\ell := (A', c'_\alpha)$ to P_r
29 :	**return** $\sigma := (r', s' \cdot \alpha^{-1})$		**return** $(\Pi := (\beta, (\mathsf{pk}, m', \sigma' := (r', s'))), \ell)$

Figure 9: Puzzle promise protocol of ECDSA-based construction.

Security Analysis. We define the security of the ECDSA-based construction in Theorem 4.3, and formally prove it in Appendix C.

Theorem 4.3. *Let* COM *be a secure commitment scheme and let* NIZK *be a non-interactive zero-knowledge scheme. If ECDSA is strongly existentially unforgeable and HSM-CL encryption is* IND-CPA *secure, then the construction in Figures 9 to 11 UC-realizes the ideal functionality* $\mathcal{F}_{\mathsf{A^2L}}$ *in the* $(\mathcal{F}_{\mathsf{KGen}}^{\mathsf{ECDSA}}, \mathcal{F}_{\mathsf{anon}}, \mathcal{F}_{\mathsf{smt}}, \mathcal{F}_{\mathsf{syn}})$*-hybrid model.*

Discussions. Our protocol achieves interoperability and fungibility. Interoperability is achieved due to the minimal cryptographic requirements of our construction. More precisely, we only require a digital signature that can be turned into an adaptor signature, and a timelock mechanism from the underlying cryptocurrency, two functionalities provided by virtually all cryptocurrencies today. As a matter of fact, we can also adapt our approach to cryptocurrencies that totally lack a scripting language, such as Monero, as we demonstrate in Appendix F. Furthermore, the tumbler is able to mediate payments in different cryptocurrencies, by running the puzzle promise and puzzle solver protocols in different cryptocurrencies. For example, this can be achieved by instantiating our constructions with the same elliptic curve group, and using one construction for the puzzle promise phase, and the other one for the puzzle solver phase [39], thereby enabling cross-chain applications like exchanges. Additionally, the output of our protocol results in accepting a channel update with a single signature verifiable by a single public key, which is essentially indistinguishable from any other payment. This in turn preserves fungibility, a crucial privacy property to avoid tainted coins and to meet GDPR requirements.

Public parameters: $\mathbb{G}, g, q$, message m, public key $Q := g^{x_1 \cdot x_2}$

	$\text{PuzzleSolver}_{P_s}(\text{sk}_s := x_1, \text{pk} := Q, \ell := (A', c'_\alpha))$		$\text{PuzzleSolver}_{P_t}(\text{sk}_t := x_2, \text{pk} := Q, c_{key} = \text{Enc}_{\text{HE}}(\text{pk}_s, x_1))$
2 :			$k_2 \leftarrow\$ \mathbb{Z}_q; R_2 \leftarrow g^{k_2}$
3 :			$\pi_2 \leftarrow \text{P}_{\text{NIZK}}(\{\exists k_2 \mid R_2 = g^{k_2}\}, k_2)$
4 :			$(\text{com}, \text{decom}) \leftarrow \text{P}_{\text{COM}}((R_2, \pi_2))$
5 :		$\xleftarrow{\text{com}}$	
6 :	$\tau, k_1 \leftarrow\$ \mathbb{Z}_q; c''_\alpha \leftarrow (c'_\alpha)^\tau; R_1 \leftarrow g^{k_1}$		
7 :	$\pi_1 \leftarrow \text{P}_{\text{NIZK}}(\{\exists k_1 \mid R_1 = g^{k_1}\}, k_1)$		
8 :		$\xrightarrow{c''_\alpha, R_1, \pi_1}$	
9 :			If $\text{V}_{\text{NIZK}}(\pi_1, R_1) \neq 1$ then abort
10 :			$\gamma \leftarrow \text{Dec}_{\text{HE}}(\text{sk}_t, c''_\alpha); A'' \leftarrow g^\gamma; R_c \leftarrow (R_2)^\gamma$
11 :			$\pi_c \leftarrow \text{P}_{\text{NIZK}}(\{\exists \gamma \mid R_c = (R_2)^\gamma\}, \gamma)$
12 :			$\pi_\gamma \leftarrow \text{P}_{\text{NIZK}}(\{\exists \gamma \mid A'' = g^\gamma \wedge R_c = (R_2)^\gamma\}, \gamma)$
13 :			$R \leftarrow (R_1)^{k_2 \cdot \gamma}; R := (r_x, r_y);$ Set $r \leftarrow r_x \bmod q$
14 :			$c \leftarrow \text{Enc}_{\text{HE}}(\text{pk}_s, (k_2)^{-1} \cdot r \cdot x_2 \cdot x_1 + (k_2)^{-1} \cdot H(m) + \rho q)$
15 :		$\xleftarrow{(\text{decom}, R_2, \pi_2), c, A'', R_c, \pi_c, \pi_\gamma}$	
16 :	If $\text{V}_{\text{COM}}(\text{com}, \text{decom}, (R_2, \pi_2)) \neq 1$ then abort		
17 :	If $\text{V}_{\text{NIZK}}(\pi_c, R_c) \neq 1$ then abort		
18 :	If $\text{V}_{\text{NIZK}}(\pi_\gamma, (A'', R_c)) \neq 1$ then abort		
19 :	If $(A')^\tau \neq A''$ then abort		
20 :	$R \leftarrow (R_c)^{k_1}; R := (r_x, r_y);$ Set $r \leftarrow r_x \bmod q$		
21 :	$s_2 \leftarrow \text{Dec}_{\text{HE}}(\text{sk}_s, c)$		
22 :	If $(R_2)^{s_2 \bmod q} \neq Q^r \cdot g^{H(m)}$ then abort		
23 :	$\bar{s} \leftarrow s_2 \cdot (k_1)^{-1} \bmod q$		
24 :		$\xrightarrow{\bar{s}}$	
25 :			$s \leftarrow (\gamma)^{-1} \cdot \bar{s}$
26 :			If verification of (r, s) fails then abort
27 :			Else publish signature (r, s)
28 :	$\gamma \leftarrow (s \cdot (\bar{s})^{-1})^{-1}; \bar{\alpha} \leftarrow \gamma \cdot \tau^{-1}$		
29 :	Send $\bar{\alpha}$ to P_r		
30 :	**return** $\bar{\alpha}$		**return** $\top$

Figure 10: Puzzle solver protocol of ECDSA-based construction.

$\text{Open}(\Pi, \bar{\alpha})$

Parse Π as $(\beta, (\text{pk}, m', \sigma' := (r', s')))$
Set $\alpha \leftarrow \bar{\alpha} \cdot \beta^{-1}$
Set $s \leftarrow s' \cdot \alpha^{-1}$
return (r', s)

$\text{Verify}(\Pi, \sigma)$

Parse Π as $(\beta, (\text{pk}, m', \sigma'))$
return $\text{Verify}_{\text{ECDSA}}(\text{pk}, m', \sigma)$

Figure 11: Open and verify algorithms of ECDSA-based construction.

5 PERFORMANCE ANALYSIS

5.1 Implementation Details

We implemented our protocols and evaluated their performance. The implementation is written in C, and it relies on the RELIC library [2] for the cryptographic operations, and on the PARI library [50] for the arithmetic operations in class groups. All our constructions have been instantiated over the elliptic curve *secp256k1*, which is also used in Bitcoin. HSM-CL encryption scheme has been instantiated for 128-bit security level as described in [10, Section 4]. Zero-knowledge proofs for discrete logarithm and Diffie-Hellman tuple have been implemented using Σ-protocols [16] and made non-interactive using the Fiat-Shamir heuristic [22]. Lastly, we have instantiated the commitment scheme using the hash function SHA-256.

We replaced the distributed key generation by randomly assigning keys to every party. Key generation is a one-time operation at setup (e.g., opening a payment channel). We refer to [10, 24] for a detailed performance evaluation of distributed ECDSA and Schnorr key generation, respectively. In the following, we focus on the remaining operations as they are the ones defined for the first time in this work. The source code is available at [23].

5.2 Evaluation

Testbed. We used three EC2 instances from Amazon AWS, where P_t was a m5a.2xlarge instance (2.50GHz AMD EPYC 7571 processor with 8 cores, 32GB RAM) located in Frankfurt, whereas P_s and P_r were m5a.large instances (2.50GHz AMD EPYC 7571 processor with 2 cores, 8GB RAM) located in Singapore and Oregon, respectively. In order to show that network latency is the biggest bottleneck in running times, we also measured performance in a LAN network. The benchmarks for a LAN network were taken on a machine with 2.80GHz Intel Xeon E3-1505M v5 processor with 8 cores, 32GB RAM. All the machines were running Ubuntu 18.04 LTS. We measured the average runtimes over 100 runs each. The results of our performance evaluation are reported in Table 2.

Computation Time. All our protocols complete in < 5 seconds, where the running time is dominated by network latency. The impact of network latency is obvious when we look at the running time for the LAN setting. In that case, our ECDSA-based construction finishes in ~1 second, whereas our Schnorr- and DLSAG-based constructions takes ~0.5 and ~0.7 second, respectively. The reason for this is that ECDSA-based two-party signing has a more complex structure, as it requires additional ciphertext computations and homomorphic operations.

Next, we compare our constructions with the state-of-the-art payment hub TumbleBit [27]. In order to have more precise results, we performed the comparison in a LAN setting without any network latency. TumbleBit requires ~0.6 second to complete, hence, our Schnorr-based construction is slightly faster, whereas our DLSAG- and ECDSA-based constructions are slightly slower.

Communication Overhead. We measured the communication overhead as the amount of information that parties need to exchange during the execution of the protocols. Hence, the bandwidth column in our table corresponds to the combined total amount of messages exchanged for the specific protocol. The ECDSA-based construction has a higher communication overhead compared to DLSAG- and Schnorr-based constructions. This is due to the fact that ECDSA-based two-party signing requires a HSM-CL ciphertext, as explained in [10], and we have to perform two-party ECDSA signing in both puzzle promise and puzzle solver protocols. TumbleBit requires 326KB of bandwidth, hence, our ECDSA-, DLSAG- and Schnorr-based constructions incur ~19x, ~23x and ~30x less communication overhead, respectively.

In summary, we highlight three points. First, our constructions highly reduce the communication complexity while retaining a computation time comparable to TumbleBit. Interestingly, results for TumbleBit [27] are shown for a security level of 80 bits, whereas we run our experiments with a security parameter that provides 128 bits of security. Thus, our construction is more efficient even when providing a higher level of security.

Second, the reduction in communication overhead is not due to a more efficient implementation, but because A^{2}L is *asymptotically* more efficient. In a bit more detail, TumbleBit relies on the cut-and-choose technique, which implies that parties need to compute and exchange messages composed of $\binom{m+n}{m}$ elements, where m and n are the parameters for the cut-and-choose game. For instance, authors of TumbleBit used $m = 15$ and $n = 285$ to achieve 80 bits of security. Instead, A^{2}L requires to compute and exchange messages composed of constant number of elements.

Third, we note that the main bottleneck with respect to computation and communication in our constructions is CL encryption [12] and CLDL zero-knowledge argument of knowledge [11] (denoted as π_α in our constructions). In our implementation a single CL ciphertext has size of 3KB and takes ~130 milliseconds to compute, while a CLDL proof has size of 2.5KB and takes ~140 milliseconds to compute. A possible optimization is for Tumbler to generate many random α values, along with their corresponding ciphertext c_α and proof π_α during its idle time, so that during the actual protocol run these values do not need to be computed, which results in a significant saving in computation time during the puzzle promise protocol.

6 RELATED WORK

On-Chain Tumblers. Several prior works exist where a centralized tumbler assists users to mix their coins [3, 5, 25, 28, 40, 42, 46–49, 52, 53]. However, all these constructions heavily rely on on-chain transactions to operate, thus, hindering scalability. A^{2}L operates instead with off-chain payments, aiding the scalability of current blockchains. Moreover, while the aforementioned systems are restricted to one (or few) cryptocurrencies, A^{2}L relies only on widely deployed cryptographic primitives such as digital signatures and timelocks, paving the way to interoperable cross-chain applications.

Payment-Channel Networks (PCNs). In a PCN [45], parties performs payments through a path of opened channels between sender and receiver. Recent works have studied their security, privacy, and concurrency guarantees [38, 39]. We consider this research line as orthogonal to our work, since the underlying protocol requires to reveal the predecessor and successor nodes in the path to the intermediaries, which is exactly the privacy notion in a PCH, with only one intermediary (i.e., the tumbler).

Threshold ECDSA Protocols. Subsequent to Lindell's two-party ECDSA construction [36], Doerner et al. [17, 18] and Lindell et al. [37] provided a threshold variant of ECDSA signing, which can also be used in the 2-of-2 signature setting that we require. However, [37] performs worse with respect to both communication and computation in this setting. On the other hand, although [17] and [18] perform better with respect to computation in the 2-of-2 setting, they require more communication. Since we are in the WAN network, we want to minimize the communication, hence, they are not suitable for our scenario. Recently, Castagnos et al. [10] proposed a generalized approach to Lindell's original two-party ECDSA protocol [36] using hash proof systems. Their approach gets rid of the Paillier-EC assumption, and additionally requires less communication compared to other protocols. Furthermore, their homomorphic encryption scheme works over a group of the same order as the order of the elliptic curve group for ECDSA. Hence, they naturally avoid all the difficulties and inefficiencies arising from using a different moduli for the encryption scheme and the order of the elliptic curve (which was the case in Lindell's construction [36]). Due to the aforementioned reasons we opted for the two-party ECDSA construction of Castagnos et al. [10], and their underlying homomorhpic encryption scheme [12] as building block in our protocols.

Table 2: Performance of Schnorr-, ECDSA- and DLSAG-based constructions. Time is shown in seconds.

	WAN[1]			LAN			Bandwidth		
	Schnorr	ECDSA	DLSAG	Schnorr	ECDSA	DLSAG	Schnorr	ECDSA	DLSAG
Puzzle Promise	1.622	1.985	1.550	0.417	0.659	0.468	6.53KB	9.81KB	8.21KB
Puzzle Solver	1.102	1.579	1.230	0.111	0.349	0.167	4.11KB	7.39KB	5.92KB
Open	1.139	1.147	1.155	0.040	0.042	0.058	0.16KB	0.16KB	0.16KB
Total	3.863	4.711	3.935	0.568	1.050	0.693	10.80KB	17.36KB	14.29KB

[1]Payment Hub (Singapore-Frankfurt-Oregon)

7 CONCLUSION

This paper presents Trilero, a new cryptographic protocol to realize a secure, privacy-preserving, interoperable, and fungibility-preserving PCHs. The core building block of Trilero is A^{2}L, a novel three-party protocol to synchronize the updates between the payment channels involved in a PCH. We developed three instantiations of A^{2}L, based on Schnorr, ECDSA and DLSAG signatures. We defined and proved security and privacy for Trilero and A^{2}L in the UC framework. We further demonstrated that Trilero is the most efficient Bitcoin-compatible PCH, showing that our ECDSA-, DLSAG- and Schnorr-based instantiations require ~19x, ~23x and ~30x less bandwidth, respectively, than the state-of-the-art PCH TumbleBit, even when providing a higher level of security. Moreover, Trilero provides fungibility and interoperability with virtually all cryptocurrencies today. For instance, this was shown with our DLSAG-based instantiation, a novel linkable ring signature proposed to add support for payment channels that is being considered in the Monero community [43].

As a future work, it would be interesting to generalize our construction to multi-hop payment hubs and, ultimately, to interface PCHs with payment channel networks. Finally, we intend to explore techniques to achieve value privacy guarantees and, possibly, the inherent trade-offs between interoperability and value privacy.

ACKNOWLEDGEMENTS

We gratefully thank Lloyd Fournier and Ida Tucker for the helpful discussions. This work has been partially supported by the European Research Council (ERC) under the European Unions Horizon 2020 research (grant agreement No 771527-BROWSEC); by Netidee through the project EtherTrust (grant agreement 2158) and PROFET (grant agreement P31621); by the Austrian Research Promotion Agency through the Bridge-1 project PR4DLT (grant agreement 13808694); by COMET K1 SBA, ABC; by Chaincode Labs; by the Austrian Science Fund (FWF) through the Meitner program and project W1255-N23.

REFERENCES

[1] Lightning Network Specifications. Github project. https://github.com/lightningnetwork/lightning-rfc.
[2] D. F. Aranha and C. P. L. Gouvêa. 2020. RELIC is an Efficient LIbrary for Cryptography. Github Project. (2020). https://github.com/relic-toolkit/relic.
[3] George Bissias, A. Pinar Ozisik, Brian N. Levine, and Marc Liberatore. 2014. Sybil-Resistant Mixing for Bitcoin. In *WPES*. ACM, 149–158. https://doi.org/10.1145/2665943.2665955
[4] Manuel Blum, Paul Feldman, and Silvio Micali. 1988. Non-interactive Zero-knowledge and Its Applications. In *STOC*. ACM, 103–112. http://doi.acm.org/10.1145/62212.62222
[5] Joseph Bonneau, Arvind Narayanan, Andrew Miller, Jeremy Clark, Joshua A. Kroll, and Edward W. Felten. 2014. Mixcoin: Anonymity for Bitcoin with Accountable Mixes. In *Financial Cryptography and Data Security*. 486–504.
[6] Jan Camenisch and Anna Lysyanskaya. 2005. A Formal Treatment of Onion Routing. In *Advances in Cryptology – CRYPTO 2005*. 169–187.
[7] Ran Canetti. 2000. Universally Composable Security: A New Paradigm for Cryptographic Protocols. Cryptology ePrint Archive, Report 2000/067. https://eprint.iacr.org/2000/067.
[8] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. 2007. Universally Composable Security with Global Setup. In *Theory of Cryptography*. 61–85.
[9] Ran Canetti and Tal Rabin. 2003. Universal composition with joint state. In *Annual International Cryptology Conference*. 265–281.
[10] Guilhem Castagnos, Dario Catalano, Fabien Laguillaumie, Federico Savasta, and Ida Tucker. 2019. Two-Party ECDSA from Hash Proof Systems and Efficient Instantiations. In *Advances in Cryptology – CRYPTO*. 191–221.
[11] Guilhem Castagnos, Dario Catalano, Fabien Laguillaumie, Federico Savasta, and Ida Tucker. 2020. Bandwidth-efficient threshold EC-DSA. Cryptology ePrint Archive, Report 2020/084. (2020). https://eprint.iacr.org/2020/084.
[12] Guilhem Castagnos and Fabien Laguillaumie. 2015. Linearly Homomorphic Encryption from DDH. In *Topics in Cryptology — CT-RSA*. Cham, 487–505.
[13] Guilhem Castagnos, Fabien Laguillaumie, and Ida Tucker. 2018. Practical Fully Secure Unrestricted Inner Product Functional Encryption Modulo p. In *Advances in Cryptology – ASIACRYPT*. 733–764.
[14] Ronald Cramer and Victor Shoup. 2002. Universal Hash Proofs and a Paradigm for Adaptive Chosen Ciphertext Secure Public-Key Encryption. In *Advances in Cryptology — EUROCRYPT 2002*, Lars R. Knudsen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 45–64.
[15] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, Dawn Song, and Roger Wattenhofer. 2016. On Scaling Decentralized Blockchains. In *Financial Cryptography and Data Security*.
[16] I. Damgård. 2002. On the σ-protocols. Lecture Notes, University of Aarhus, Department for Computer Science. (2002).
[17] Jack Doerner, Yashvanth Kondi, Eysa Lee, and abhi shelat. 2018. Secure Multi-party Threshold ECDSA from ECDSA Assumptions. In *Oakland S&P'2018*.
[18] Jack Doerner, Yashvanth Kondi, Eysa Lee, and abhi shelat. 2019. Threshold ECDSA from ECDSA Assumptions: The Multiparty Case. In *Oakland S&P'2019*.
[19] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, Julia Hesse, and Kristina Hostáková. 2019. Multi-party Virtual State Channels. In *Advances in Cryptology – EUROCRYPT 2019*, Yuval Ishai and Vincent Rijmen (Eds.). Springer International Publishing, Cham, 625–656.
[20] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, and Daniel Malinowski. 2017. Perun: Virtual Payment Hubs over Cryptocurrencies. Cryptology ePrint Archive, Report 2017/635. (2017). https://eprint.iacr.org/2017/635.
[21] Stefan Dziembowski, Sebastian Faust, and Kristina Hostakova. 2018. General State Channel Networks. In *CCS*.
[22] Amos Fiat and Adi Shamir. 1987. How To Prove Yourself: Practical Solutions to Identification and Signature Problems. In *Advances in Cryptology — CRYPTO' 86*. 186–194.
[23] Anonymized for submission. 2020. Source code for Trilero implementation. Github repository. (2020). https://github.com/trileroa2l/evaluation.
[24] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. 2007. Secure Distributed Key Generation for Discrete-Log Based Cryptosystems. *Journal of Cryptology* (2007), 51–83.
[25] gmaxwell (pseudonym). 2013. CoinSwap: Transaction Graph Disjoint Trustless Trading. Forum post. (2013). https://bitcointalk.org/index.php?topic=321228.0.
[26] Matthew Green and Ian Miers. 2017. Bolt: Anonymous Payment Channels for Decentralized Currencies. In *CCS*.
[27] Ethan Heilman, Leen Alshenibr, Foteini Baldimtsi, Alessandra Scafuro, and Sharon Goldberg. 2017. TumbleBit: An Untrusted Bitcoin-Compatible Anonymous Payment Hub. In *NDSS*.
[28] Ethan Heilman, Foteini Baldimtsi, and Sharon Goldberg. 2016. *Blindly Signed Contracts: Anonymous On-Blockchain and Off-Blockchain Bitcoin Transactions.*

Technical Report 056.
[29] George Kappos, Haaroon Yousaf, Mary Maller, and Sarah Meiklejohn. 2018. An Empirical Analysis of Anonymity in Zcash. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. 463–477.
[30] Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. 2013. Universally Composable Synchronous Computation. In *Theory of Cryptography*. 477–498.
[31] Rami Khalil, Arthur Gervais, and Guillaume Felley. 2018. NOCUST - A Securely Scalable Commit-Chain. Cryptology ePrint Archive, Report 2018/642. (2018). https://eprint.iacr.org/2018/642.
[32] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. 2016. Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing. In *USENIX Security Symposium*. 279–296.
[33] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. 2018. OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. 583–598.
[34] Joshua Lind, Oded Naor, Ittay Eyal, Florian Kelbert, Peter R. Pietzuch, and Emin Gün Sirer. 2018. Teechain: Reducing Storage Costs on the Blockchain With Offline Payment Channels. In *ACM SYSTOR*. 125.
[35] Yehuda Lindell. 2011. Highly-efficient Universally-composable Commitments Based on the DDH Assumption. In *Proceedings of the 30th Annual International Conference on Theory and Applications of Cryptographic Techniques: Advances in Cryptology (EUROCRYPT'11)*. Springer-Verlag, Berlin, Heidelberg, 446–466. http://dl.acm.org/citation.cfm?id=2008684.2008718
[36] Yehuda Lindell. 2017. Fast Secure Two-Party ECDSA Signing. Cryptology ePrint Archive, Report 2017/552. (2017). https://eprint.iacr.org/2017/552.
[37] Yehuda Lindell and Ariel Nof. 2018. Fast Secure Multiparty ECDSA with Practical Distributed Key Generation and Applications to Cryptocurrency Custody. In *CCS*. ACM, 1837–1854. https://doi.org/10.1145/3243734.3243788
[38] Giulio Malavolta, Pedro Moreno-Sanchez, Aniket Kate, Matteo Maffei, and Srivatsan Ravi. 2017. Concurrency and Privacy with Payment-Channel Networks. In *CCS*. ACM, 455–471. https://doi.org/10.1145/3133956.3134096
[39] Giulio Malavolta, Pedro Moreno-Sanchez, Clara Schneidewind, Aniket Kate, and Matteo Maffei. 2019. Anonymous Multi-Hop Locks for Blockchain Scalability and Interoperability. (2019). https://www.ndss-symposium.org/ndss-paper/anonymous-multi-hop-locks-for-blockchain-scalability-and-interoperability/
[40] Sarah Meiklejohn and Rebekah Mercer. 2017. *Möbius: Trustless Tumbling for Transaction Privacy*. Technical Report 881.
[41] Pedro Moreno-Sanchez, Randomrun, Duc V. Le, Sarang Noether, Brandon Goodell, and Aniket Kate. 2019. DLSAG: Non-Interactive Refund Transactions For Interoperable Payment Channels in Monero. Cryptology ePrint Archive, Report 2019/595. (2019). https://eprint.iacr.org/2019/595.
[42] Pedro Moreno-Sanchez, Tim Ruffing, and Aniket Kate. 2017. PathShuffle: Credit Mixing and Anonymous Payments for Ripple. *PoPETs* 2017, 3 (2017), 110. https://doi.org/10.1515/popets-2017-0031
[43] Sarang Noether and Brandon Goodell. 2018. Dual linkable ring signatures. Monero Research Bulleting. (2018). https://web.getmonero.org/resources/research-lab/pubs/MRL-0008.pdf.
[44] Andrew Poelstra. 2020. Documentation about scriptless scripts. Github repository. (2020). https://github.com/ElementsProject/scriptless-scripts.
[45] Joseph Poon and Thaddeus Dryja. 2016. The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments. Technical Report. (2016). https://lightning.network/lightning-network-paper.pdf.
[46] Tim Ruffing and Pedro Moreno-Sanchez. 2017. ValueShuffle: Mixing Confidential Transactions for Comprehensive Transaction Privacy in Bitcoin. In *Financial Cryptography and Data Security -BITCOIN*, Vol. 10323. Springer, 133–154.
[47] Tim Ruffing, Pedro Moreno-Sanchez, and Aniket Kate. 2014. CoinShuffle: Practical Decentralized Coin Mixing for Bitcoin. In *ESORICS'14*, Vol. 8713. Springer, 345–364.
[48] Tim Ruffing, Pedro Moreno-Sanchez, and Aniket Kate. 2017. P2P Mixing and Unlinkable Bitcoin Transactions. In *NDSS*. The Internet Society.
[49] István András Seres, Dániel A. Nagy, Chris Buckland, and Péter Burcsi. 2019. *MixEth: Efficient, Trustless Coin Mixing Service for Ethereum*. Technical Report 341.
[50] The PARI Group 2019. *PARI/GP version* `2.12.0`. The PARI Group, Univ. Bordeaux. available from http://pari.math.u-bordeaux.fr/.
[51] Manny Trillo. 2013. Stress Test Prepares VisaNet for the Most Wonderful Time of the Year. http://www.visa.com/blogarchives/us/2013/10/10/stress-test-prepares-visanet-for-the-most-wonderful-time-of-the-year/index.html. (2013). Accessed: 2017-08-07.
[52] Luke Valenta and Brendan Rowan. 2015. Blindcoin: Blinded, Accountable Mixes for Bitcoin. In *Financial Cryptography and Data Security (Lecture Notes in Computer Science)*, Michael Brenner, Nicolas Christin, Benjamin Johnson, and Kurt Rohloff (Eds.). Springer Berlin Heidelberg, 112–126.
[53] Jan Henrik Ziegeldorf, Fred Grossmann, Martin Henze, Nicolas Inden, and Klaus Wehrle. 2015. CoinParty: Secure Multi-Party Mixing of Bitcoins. In *CODASPY*. ACM, 75–86. https://doi.org/10.1145/2699026.2699100

A CORRECTNESS DEFINITIONS

A.1 Payment Channel Hub (PCH)

Intuitively, a PCH is correct if the receiver gets the money paid by the sender through the tumbler with overwhelming probability.

Definition A.1 (Correctness of PCHs). Let $\mathcal{P}$ be a PCH, $\lambda \in \mathbb{N}$ and $n \in \mathsf{poly}(\lambda)$. Let $\mathbb{V} = \{P_1, \ldots, P_n, P_t\}$ be a set of parties, and P_t be the intermediary (tumbler).

For any $P_s, P_r \in \mathbb{V}$, any $x_{P_s}, x_{P_t}, x'_{P_t}, x'_{P_r} \in \mathbb{R}_{>0}$, let

$$(P_s, P_t)^{x_{P_s}, x_{P_t}} \leftarrow \mathsf{OpenChannel}(P_s, P_t, x_{P_s}, x_{P_t})$$

and

$$(P_t, P_r)^{x'_{P_t}, x_{P_r}} \leftarrow \mathsf{OpenChannel}(P_t, P_r, x'_{P_t}, x'_{P_r})$$

such that $(P_s, P_t)^{x_{P_s}, x_{P_t}} \neq \bot$ and $(P_t, P_r)^{x'_{P_t}, x'_{P_r}} \neq \bot$.

We say that $\mathcal{P}$ is correct if there exists a negligible function negl, such that for any $\mathsf{amt} \in \mathbb{R}_{>0}$, where $x_{P_s} \geq \mathsf{amt}$ and $x'_{P_t} \geq \mathsf{amt}$, the following holds

$$\Pr[\mathsf{Pay}(P_s, P_r, \mathsf{amt}) = 1] \geq 1 - \mathsf{negl}(\lambda)\,.$$

A.2 Anonymous Atomic Locks (A^{2}L)

Intuitively, A^{2}L is correct if opening of a promise using a valid opening information verifies with overwhelming probability.

Definition A.2 (Correctness of A^{2}Ls). Let $\mathcal{L}$ be an A^{2}L, $\lambda \in \mathbb{N}$ and $n \in \mathsf{poly}(\lambda)$. Let P_t be the intermediary, $(P_1, \ldots, P_n) \in \mathbb{P}^n$ be a vector of parties, $(\mathsf{sk}_1, \ldots, \mathsf{sk}_n, \mathsf{sk}_t)$ be a vector of secret keys, and $(\mathsf{pk}_{1,t}, \ldots, \mathsf{pk}_{n,t})$ be a vector of public keys, such that for all $1 \leq i \leq n$, it holds that

$$\{(\mathsf{sk}_i, \mathsf{pk}_{i,t}), (\mathsf{sk}_t, \mathsf{pk}_{i,t})\} \leftarrow \langle \mathsf{KGen}_{P_i}(1^\lambda), \mathsf{KGen}_{P_t}(1^\lambda)\rangle.$$

Furthermore, let $(\Pi_1, \ldots, \Pi_n)$ be a vector of promises, $(\ell_1, \ldots, \ell_n)$ be a vector of locks, and $(\varrho_1, \ldots, \varrho_n)$ be a vector of opening information, such that for all $1 \leq i, j \leq n$, it holds that

$$\{\cdot, (\Pi_i, \ell_i)\} \leftarrow \langle \mathsf{PuzzlePromise}_{P_t}(\mathsf{sk}_t, \mathsf{pk}_{i,t}), \mathsf{PuzzlePromise}_{P_i}(\mathsf{sk}_i, \mathsf{pk}_{i,t})\rangle$$

and

$$\{\varrho_i, \cdot\} \leftarrow \langle \mathsf{PuzzleSolver}_{P_j}(\mathsf{sk}_j, \mathsf{pk}_{j,t}, \ell_i), \mathsf{PuzzleSolver}_{P_t}(\mathsf{sk}_t, \mathsf{pk}_{j,t})\rangle.$$

We say that $\mathbb{L}$ is correct if there exists a negligible function negl, such that for all $1 \leq i \leq n$, the following holds

$$\Pr[\mathsf{Verify}(\Pi_i, \mathsf{Open}(\Pi_i, \varrho_i)) = 1] \geq 1 - \mathsf{negl}(\lambda)\,.$$

B HASH PROOF SYSTEMS AND CASTAGNOS-LAGUILLAUMIE ENCRYPTION SCHEME

In our constructions we use the homomorphic encryption scheme by Castagnos-Laguillaumie (CL) [12], more specifically, the variant of the scheme described in [10, 13] (called HSM-CL), and its instantiation based on class groups of an imaginary quadratic field. HSM-CL follows the paradigm of Cramer-Shoup [14] for building IND-CPA secure encryption scheme from hash proof systems (HPS), which were also introduced in the same paper. Consider a set of

words $\mathcal{X}$, and NP language $\mathcal{L} \subset \mathcal{X}$, such that $\mathcal{L} := \{x \in \mathcal{X} \mid w \in \mathcal{W} : (x, w) \in R\}$, where R is the relation defining the language, $\mathcal{L}$ is the language of true statements in $\mathcal{X}$, and for $(x, w) \in R$, $w \in \mathcal{W}$ is a witness for $x \in \mathcal{L}$. The tuple $(\mathcal{X}, \mathcal{L}, \mathcal{W}, R)$ defines an instance of a subset membership problem (i.e., the problem of deciding if an element $x \in \mathcal{X}$ is in $\mathcal{L}$ or $\mathcal{X} \setminus \mathcal{L}$). A HPS associates a projective hash family (PHF) to such a subset membership problem. It defines a key generation algorithm PHF.KGen, which outputs a secret hashing key hk sampled from distribution of hashing keys $\mathcal{D}_{\mathsf{hk}}$ over a hash key space K_{hk}, and a public projection key $\mathsf{hp} \leftarrow \omega(\mathsf{hk})$ in projection key space K_{hp}, where $\omega : K_{\mathsf{hk}} \rightarrow K_{\mathsf{hp}}$ is an efficient auxiliary function. The secret hashing key hk defines a hash function $\mathcal{H}_{\mathsf{hk}} : \mathcal{X} \rightarrow \Pi$, where Π is typically an algebraic group, and the public projection key hp allows for the public evaluation of the hash function on words $x \in \mathcal{L}$, more precisely, $\mathcal{H}_{\mathsf{hp}}(x, w) = \mathcal{H}_{\mathsf{hk}}(x)$ for $(x, w) \in R$. Thus, a projective hash family PHF is defined by $(\{\mathcal{H}_{\mathsf{hk}}\}_{\mathsf{hk} \in K_{\mathsf{hk}}}, K_{\mathsf{hk}}, \mathcal{X}, \mathcal{L}, \Pi, K_{\mathsf{hp}}, \omega)$ [10]. Cramer-Shoup [14] defined an IND-CPA secure encryption scheme from HPS, which is composed of the algorithms (KGen, Enc, Dec), where KGen runs PHF.KGen and sets the secret key $\mathsf{sk} := \mathsf{hk} \in K_{\mathsf{hk}}$, and the associated public key $\mathsf{pk} := \mathsf{hp} \leftarrow \omega(\mathsf{hk})$. Encryption of a plaintext message $m \in \mathbb{Z}_q$ is done by sampling a random pair $(u, w) \in R$, and computing $c := (u, \mathcal{H}_{\mathsf{hp}}(u, w) \cdot \mathsf{Encode}(m)) \leftarrow \mathsf{Enc}(\mathsf{hp}, m)$. To decrypt a ciphertext $(u, e) \in \mathcal{X} \times \Pi$, one computes $m := \mathsf{Decode}(\frac{e}{\mathcal{H}_{\mathsf{hk}}(u)}) \leftarrow \mathsf{Dec}(\mathsf{hk}, (u, e))$. The resulting encryption scheme also support homomorphic operations as described in [10]. Applying this approach to a HPS generated from the hard subset membership problem that is based on class groups of an imaginary quadratic fields results in HSM-CL encryption scheme [10, 13].

The main reason for choosing the HSM-CL encryption scheme as opposed to any other linearly homomorphic encryption scheme is that it can be instantiated to work over $\mathbb{Z}_q$, for a q that is the same as the order of the elliptic curve group used for Schnorr, ECDSA and DLSAG signature schemes. If one uses an encryption scheme with a plaintext space larger than q, then several problems appear. For example, two-party ECDSA construction of Lindell [36] uses Paillier, which has a plaintext space $\mathbb{Z}_N$, for a composite N much larger than q. In that case to enforce correctness and security of the protocol the value of N needs to be chosen large enough, so that no wrap around occurs, and one needs to prove that the encrypted value is within the right range, which requires an expensive range proof. We can avoid these issues by using the HSM-CL encryption scheme instantiated with the plaintext space $\mathbb{Z}_q$. Another advantage of HSM-CL is that in the security proofs challenger's access to the secret key does not compromise the indistinguishability of ciphertexts, as it relies on a computational assumption and a statistical argument. For more information about the problems arising from using an encryption scheme with a larger modulus than the elliptic curve group order, and how these problems are addressed by the HSM-CL encryption scheme we refer the reader to [10].

C SECURITY ANALYSIS OF A^{2}L

Throughout this section we denote by $\mathsf{poly}(\lambda)$ any function that is bounded by a polynomial in λ, where $\lambda \in \mathbb{N}$ is the security parameter. We denote any function that is negligible in the security parameter by $\mathsf{negl}(\lambda)$. We say an algorithm is PPT if it is modeled as a probabilistic Turing machine whose running time is bounded by some function $\mathsf{poly}(\lambda)$.

We prove security according to the UC framework [7], and in the presence of *malicious adversaries* with *static corruptions*.

Key Generation Functionalities. Our protocols build on top of key generation funtionalities for Schnorr, ECDSA and DLSAG. The key generation functionalities for Schnorr and ECDSA are taken from [39]. Ideal functionality for key generation of Schnorr signature $\mathcal{F}_{\mathsf{KGen}}^{\mathsf{Schnorr}}$ is defined below (it models a distributed key generation for discrete logarithm-based schemes).

$\mathsf{KeyGen}(\mathbb{G}, g, q)$

Upon invocation by both P_1 and P_2 on input $(\mathbb{G}, g, q)$:
sample $x \leftarrow\!\!\$\, \mathbb{Z}_q$ and compute $Q = g^x$
set $\mathsf{sk}_{P_1,P_2} = x$
sample $x_1, x_2 \leftarrow\!\!\$\, \mathbb{Z}_q$ and a hash function $H : \{0,1\}^* \rightarrow \mathbb{Z}_q$
send (x_1, Q, H) to P_1 and (x_2, Q, H) to P_2
ignore future calls by (P_1, P_2)

The ideal functionality for key generation of ECDSA signature $\mathcal{F}_{\mathsf{KGen}}^{\mathsf{ECDSA}}$ is defined as follows:

$\mathsf{KeyGen}(\mathbb{G}, g, q)$

Upon invocation by both P_1 and P_2 on input $(\mathbb{G}, g, q)$:
sample $x \leftarrow\!\!\$\, \mathbb{Z}_q$ and compute $Q = g^x$
set $\mathsf{sk}_{P_1,P_2} = x$
sample $x_1, x_2 \leftarrow\!\!\$\, \mathbb{Z}_q$ and a hash function $H : \{0,1\}^* \rightarrow \mathbb{Z}_q$
sample a key pair $(\mathsf{sk}_{P_1,P_2}, \mathsf{pk}_{P_1,P_2}) \leftarrow \mathsf{KGen}_{\mathsf{HE}}()$
compute $c \leftarrow \mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}, \tilde{x_1})$ for a random $\tilde{x_1} \leftarrow\!\!\$\, \mathbb{Z}_q$
send (x_1, Q, H, sk) to P_1 and (x_2, Q, H, c) to P_2
ignore future calls by (P_1, P_2)

Lastly, we define the ideal functionality for key generation of DLSAG signature $\mathcal{F}_{\mathsf{KGen}}^{\mathsf{DLSAG}}$, which is similar to the Schnorr-based key generation, however, in DLSAG we require dual keys, hence, we have to account for that in our ideal functionality.

$\mathsf{KeyGen}(\mathbb{G}, g, q)$

Upon invocation by both P_1 and P_2 on input $(\mathbb{G}, g, q)$:
sample $x, x' \leftarrow\!\!\$\, \mathbb{Z}_q$ and compute $Q = g^x$, $Q' = g^{x'}$
set $\mathsf{sk}_{P_1,P_2,0} = x$ and $\mathsf{sk}_{P_1,P_2,1} = x'$
sample $x_{P_1,0}, x_{P_1,1}, x_{P_2,0}, x_{P_2,1} \leftarrow\!\!\$\, \mathbb{Z}_q$, a hash function $H : \{0,1\}^* \rightarrow \mathbb{Z}_q$
send $(x_{P_1,0}, x_{P_1,1}, Q, Q', H)$ to P_1 and $(x_{P_2,0}, x_{P_2,1}, Q, Q', H)$ to P_2
ignore future calls by (P_1, P_2)

We stress that the copies of these functionalities that are invoked as subroutines are fresh independent instances, and hence, the composition theorem [7] directly applies to our setting.

Schnorr-based Construction. Here we prove Theorem 4.2.

Proof. The proof is composed of a series of hybrids, where we gradually modify the initial experiment.

$\mathcal{H}_0$: Is identical to the construction as described in Section 4.2.2.

$\mathcal{H}_1$: All the calls to the commitment scheme COM are replaced with calls to the ideal functionality $\mathcal{F}_{\mathsf{COM-ZK}}$. which is defined for a relation R as described in [36] and shown below.

Commit(sid, x, w)

Upon invocation by P_i, where $i \in \{1,2\}$, on input (x, w) :
if some (sid, $\cdot$, $\cdot$) is already recorded or $(x, w) \notin R$
then ignore the message
else record (sid, i, x) and send (com, sid) to P_{3-i}

Decommit(sid)

Upon invocation by P_i, where $i \in \{1,2\}$:
if (sid, i, x) is recorded, then send (decom, sid, x) to P_{3-i}
else ignore the message

We have to use the $\mathcal{F}_{\mathsf{COM-ZK}}$ functionality, while in our protocol the parties commit to non-interactive zero-knowledge proofs. This functionality is securely realized by having the prover commit to a non-interactive zero-knowledge proof using an ideal commitment functionality $\mathcal{F}_{\mathsf{COM}}$, such as the one from [35]. Instead of calling the commitment algorithm COM with (x, w), the parties send a message of the form Commit(sid, x, w) to the ideal functionality $\mathcal{F}_{\mathsf{COM-ZK}}$. Similarly, the decommitment is replaced with a message of the form Decommit(sid). The verifying party records the messages from $\mathcal{F}_{\mathsf{COM-ZK}}$.

$\mathcal{H}_2$: All the calls to the non-interactive zero-knowledge scheme NIZK are replaced with calls to the ideal functionality $\mathcal{F}_{\mathsf{NIZK}}$, which works with a relation R and is defined as follows.

Prove(sid, x, w)

Upon invocation by P_i, where $i \in \{1,2\}$, on input (x, w) :
if $(x, w) \notin R$, then send (proof, sid, x) to P_{3-i}
else ignore the message

Instead of calling the non-interactive zero-knowledge scheme NIZK with input (x, w), the proving party queries the ideal functionality $\mathcal{F}_{\mathsf{NIZK}}$ with message Prove(sid, x, w). The verifier records the messages from $\mathcal{F}_{\mathsf{NIZK}}$.

$\mathcal{H}_3$: Consider the following ensemble of variables in the interaction with $\mathcal{A}$: key pairs $(\mathsf{sk}_s, \mathsf{pk}_{s,t})$, $(\mathsf{sk}_t, \mathsf{pk}_{s,t})$, $(\mathsf{sk}_r, \mathsf{pk}_{r,t})$, and $(\mathsf{sk}_t, \mathsf{pk}_{r,t})$, a pair $(\bar{\alpha}, (\Pi, \ell))$ such that

$$\{\cdot, (\Pi, \ell)\} \leftarrow \langle \mathsf{PuzzlePromise}_{P_r}(\mathsf{sk}_r, \mathsf{pk}_{r,t}), \mathsf{PuzzlePromise}_{P_t}(\mathsf{sk}_t, \mathsf{pk}_{r,t})\rangle$$

and

$$\{\bar{\alpha}, \cdot\} \leftarrow \langle \mathsf{PuzzleSolver}_{P_s}(\mathsf{sk}_s, \mathsf{pk}_{s,t}, \ell), \mathsf{PuzzleSolver}_{P_t}(\mathsf{sk}_t, \mathsf{pk}_{s,t})\rangle.$$

If for any set of these variables, the adversary returns some $\sigma := (R, s)$, such that $\mathsf{Verify}(\Pi, \sigma) = 1$, but $s \neq \mathsf{Open}(\Pi, \bar{\alpha})[s]$, then the experiment aborts.

$\mathcal{H}_4$: Consider the following ensemble of variables in the interaction with $\mathcal{A}$: key pairs $(\mathsf{sk}_s, \mathsf{pk}_{s,t})$, $(\mathsf{sk}_t, \mathsf{pk}_{s,t})$, $(\mathsf{sk}_r, \mathsf{pk}_{r,t})$, and $(\mathsf{sk}_t, \mathsf{pk}_{r,t})$, a pair $(\bar{\alpha}, (\Pi, \ell))$ such that

$$\{\cdot, (\Pi, \ell)\} \leftarrow \langle \mathsf{PuzzlePromise}_{P_r}(\mathsf{sk}_r, \mathsf{pk}_{r,t}), \mathsf{PuzzlePromise}_{P_t}(\mathsf{sk}_t, \mathsf{pk}_{r,t})\rangle$$

and

$$\{\bar{\alpha}, \cdot\} \leftarrow \langle \mathsf{PuzzleSolver}_{P_s}(\mathsf{sk}_s, \mathsf{pk}_{s,t}, \ell), \mathsf{PuzzleSolver}_{P_t}(\mathsf{sk}_t, \mathsf{pk}_{s,t})\rangle.$$

If for any set of these variables, the adversary returns some $\sigma := (R, s)$, such that $\mathsf{Verify}(\Pi, \sigma) = 1$, before P_s outputs $\bar{\alpha}$ from puzzle solver protocol with P_t, such that $\mathsf{Verify}(\Pi, \mathsf{Open}(\Pi, \bar{\alpha})) = 1$ then the experiment aborts.

$\mathcal{S}$: The actions of the simulator $\mathcal{S}$ are dictated by interacting with $\mathcal{F}$. If $\mathcal{A}$ interacts with an honest user, then the simulator queries the corresponding interface of $\mathcal{F}$. More precisely, it is queried by $\mathcal{F}$ on the following set of inputs:

- PuzzlePromise: The simulator initiates the puzzle promise procedure with the adversary and replies with $\perp$ if the execution is not successful, otherwise replies with a valid promise and lock.
- PuzzleSolver: The simulator initiates the puzzle solver procedure with the adversary and replies with $\perp$ if the execution is not successful, otherwise it releases the opening information of the corresponding lock.
- Open: The simulator returns the opened lock data.

Next, we prove the indistinguishability of the neighboring experiments for the environment $\mathcal{E}$.

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

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

Proof. The proof follows directly from the security of the commitment scheme COM. □

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

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

Proof. The proof follows directly from the security of the non-interactive zero-knowledge scheme NIZK. □

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

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

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

$\mathcal{H}_2^*$: All the calls to the puzzle promise and puzzle solver protocols are replaced with calls to $\mathcal{F}_{\mathsf{Sign}}$ ideal functionality, which is defined as follows.

Sign(sid, m, α)

Upon invocation by P_1 and P_2 on input (sid, m, α) :
if some (sid, $\cdot$, $\cdot$) is already recorded, then ignore the message
else record (sid, m, α)
compute $(R, s) \leftarrow \mathsf{Sig}_{\mathsf{Schnorr}}(\mathsf{sk}_{P_1,P_2}, m)$
return $(R, s - \alpha)$

We note that the key sk_{P_1,P_2} refers to the previously established key between the parties P_1 and P_2 in the call to the $\mathcal{F}_{\mathsf{KGen}}^{\mathsf{Schnorr}}$. During the puzzle promise protocol $P_1 := P_r$ (receiver), whereas during

the puzzle solver protocol $P_1 := P_s$ (sender), and in both protocols $P_2 := P_t$ (tumbler).

Lemma C.4. *For all* PPT *distinguisher* $\mathcal{E}$ *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 simulators for the interactive puzzle promise and puzzle solver protocols. Since the puzzle promise protocol is executed between P_t and P_r, we describe two simulators depending on whether the adversary is playing the role of P_t or P_r. Later, we do the same thing for the puzzle solver protocol, however, there the simulator takes the role of P_t or P_s. We start with the simulation of the puzzle promise protocol.

1) P_r corrupted: After agreeing on a message m, the simulator $\mathcal{S}$ samples a random $\alpha^* \leftarrow_\$ \mathbb{Z}_q$, and queries Sign on input $(\mathsf{sid}, m, \alpha^*)$, for a random sid, and obtains $\sigma' := (R', s')$. $\mathcal{S}$ computes $A = g^{\alpha^*}$ and $c_\alpha = \mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}_t, \alpha^*)$, where pk_t is the HSM-CL encryption public key of P_t. $\mathcal{S}$ sends $((\mathsf{com}, \mathsf{sid}), A, c_\alpha, (\mathsf{proof}, \mathsf{sid}, \{\exists \alpha^* \mid c_\alpha = \mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}_t, \alpha^*) \wedge A = g^{\alpha^*}\}))$ to $\mathcal{A}$. At some point of the execution $\mathcal{A}$ sends $(R_1', (\mathsf{prove}, \{\exists k_1' \mid R_1' = g^{k_1'}\}, k_1'))$. $\mathcal{S}$ verifies that $R_1' = g^{k_1'}$, and if this is not the case $\mathcal{S}$ simulates P_t aborting. $\mathcal{S}$ replies with

$$\left(\begin{array}{l} \mathsf{decom}, \mathsf{sid}, \left(\begin{array}{l} R_2' = R'/(R_1' \cdot A), \\ \mathsf{proof}, \mathsf{sid}, \\ \{\exists k_2' \mid R_2' = g^{k_2'}\} \end{array}\right), \\ (s' - k_1' + e \cdot x_1') \end{array}\right)$$

where $e = H(\mathsf{pk} \| R' \| m)$, and x_1' is the value returned by the key generation to $\mathcal{A}$. The rest of the execution is unchanged.

The simulator is efficient, and the distribution induced by the simulator is identical to the real execution except for the way c_α is computed. Therefore, the indistinguishability reduces to to the IND-CPA security of the underlying encryption scheme. We prove this indistinguishability by closely following the indistinguishability proof given in [10], and using the games they have defined, which can be seen in Table 3. We note that here we use the notation for the encryption scheme explained in Appendix B, and we also implicitly assume that the messages have correct encoding, and do not explicitly call an Encode function on plaintexts as in [10]. In order to prove indistinguishability, we prove the following lemma regarding the games specified in Table 3.

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

$$\mathsf{EXEC}_{\mathsf{Game}_0,\mathcal{A},\mathcal{E}} \approx \mathsf{EXEC}_{\mathsf{Game}_1,\mathcal{A},\mathcal{E}},$$
$$\mathsf{EXEC}_{\mathsf{Game}_1,\mathcal{A},\mathcal{E}} \approx \mathsf{EXEC}_{\mathsf{Game}_2,\mathcal{A},\mathcal{E}},$$
$$\mathsf{EXEC}_{\mathsf{Game}_2,\mathcal{A},\mathcal{E}} \approx \mathsf{EXEC}_{\mathsf{Game}_3,\mathcal{A},\mathcal{E}},$$
$$\mathsf{EXEC}_{\mathsf{Game}_3,\mathcal{A},\mathcal{E}} \approx \mathsf{EXEC}_{\mathsf{Game}_4,\mathcal{A},\mathcal{E}},$$
$$\mathsf{EXEC}_{\mathsf{Game}_4,\mathcal{A},\mathcal{E}} \approx \mathsf{EXEC}_{\mathsf{Game}_5,\mathcal{A},\mathcal{E}}.$$

Proof. We demonstrate that the games depicted in Table 3 are indistinguishable from the view of the adversary $\mathcal{A}$. We denote by E_i the probability that an algorithm interacting with the simulator in Game_i outputs 1.

Game_0 to Game_1: The only difference between the two games is how c_α is computed. More precisely, in Game_1 we use the secret hashing key hk instead of the projection key hp and the witness w to compute c_α. Although, the values are computed differently, they are distributed identically, and hence, are perfectly indistinguishable. Thus,

$$|\Pr[\mathsf{E}_1] - \Pr[\mathsf{E}_0]| = 0.$$

Game_1 to Game_2: Let $\mathcal{D}$ be a distinguisher that can distinguish between Game_1 and Game_2 with a non-negligible advantage. Then, we can device $\tilde{\mathcal{S}}$ that can use $\mathcal{D}$ to break the hard subset membership assumption. $\tilde{\mathcal{S}}$ takes as input a hard subset membership challenge x^* (which is either an element of $\mathcal{L}$, or an element of $\mathcal{X} \setminus \mathcal{L}$), and instead of sampling $(u, w) \in R$ as $\mathcal{S}$ does in Game_1, it sets $u := x^*$ and computes $c_\alpha \leftarrow (u, \mathcal{H}_{\mathsf{hk}}(u) \cdot \alpha)$. When $\mathcal{D}$ returns a bit b, $\tilde{\mathcal{S}}$ returns the same bit, where 0 represents the case $x^* \in \mathcal{L}$, and 1 represents the case $x^* \in \mathcal{X} \setminus \mathcal{L}$. We can distinguish two cases here: a) Case $x^* \in \mathcal{L}$: There exists $w \in \mathcal{W}$ such that $(x^*, w) \in R$ and $\mathcal{H}_{\mathsf{hp}}(x^*, w) = \mathcal{H}_{\mathsf{hk}}(x^*)$. Hence, $c_\alpha = (u, e)$ is an encryption of α as in Game_1. b) Case $x^* \in \mathcal{X} \setminus \mathcal{L}$: The ciphertext is $(x^*, \mathcal{H}_{\mathsf{hk}}(x^*) \cdot \alpha)$, which is the distribution obtained in Game_2. Therefore, the advantage of $\tilde{\mathcal{S}}$ breaking the hard subset membership assumption is at least that of $\mathcal{D}$ distinguishing both games. Thus,

$$|\Pr[\mathsf{E}_2] - \Pr[\mathsf{E}_1]| \leq \delta_{\mathcal{L}},$$

where $\delta_{\mathcal{L}}$ is the maximal advantage of any PPT adversary in hard subset membership problem, as defined in [10].

Game_2 to Game_3: Set $\alpha^* := \alpha + r \bmod q$. Under the assumption that the hash proof system is δ_s-smooth over $\mathcal{X}$ (we refer the reader to [10] for smoothness definition of HPS), it holds that the distribution of $(x^*, \mathcal{H}_{\mathsf{hk}}(x^*) \cdot \alpha)$ and of $(x^*, \mathcal{H}_{\mathsf{hk}}(x^*) \cdot \alpha \cdot r = \mathcal{H}_{\mathsf{hk}}(x^*) \cdot \alpha^*))$ for some random $\alpha^* \in \mathbb{Z}_q$ are δ_s-close. Hence, replacing $(x^*, \mathcal{H}_{\mathsf{hk}}(x^*) \cdot \alpha)$ by $(x^*, \mathcal{H}_{\mathsf{hk}}(x^*) \cdot \alpha^*)$ cannot be noticed by any PPT adversary with advantage greater than δ_s, so we have that

$$|\Pr[\mathsf{E}_3] - \Pr[\mathsf{E}_2]| \leq \delta_s.$$

Game_3 to Game_4: As the changes are identical to that between Game_1 and Game_2, indistinguishability follows from the hardness of the subset membership problem on which the hash proof system relies, and we get that

$$|\Pr[\mathsf{E}_4] - \Pr[\mathsf{E}_3]| \leq \delta_{\mathcal{L}}.$$

Game_4 to Game_5: The changes are identical to that between Game_0 and Game_1, hence, both games are perfectly indistinguishable. Thus,

$$|\Pr[\mathsf{E}_5] - \Pr[\mathsf{E}_4]| = 0.$$

Putting it all together, we get that

$$|\Pr[\mathsf{E}_5] - \Pr[\mathsf{E}_0]| \leq 2\delta_{\mathcal{L}} + \delta_s,$$

and hence, by the hardness of the subset membership problem underlying the hash proof system, and the smoothness of the hash proof system, it holds that c_α is indistinguishable. □

Game_0	Game_1	Game_2	Game_3	Game_4	Game_5
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\mathsf{hk} \leftarrow\!\!\$\ \mathcal{D}_{\mathsf{hk}}$	$\mathsf{hk} \leftarrow\!\!\$\ \mathcal{D}_{\mathsf{hk}}$	$\mathsf{hk} \leftarrow\!\!\$\ \mathcal{D}_{\mathsf{hk}}$	$\mathsf{hk} \leftarrow\!\!\$\ \mathcal{D}_{\mathsf{hk}}$	$\mathsf{hk} \leftarrow\!\!\$\ \mathcal{D}_{\mathsf{hk}}$	$\mathsf{hk} \leftarrow\!\!\$\ \mathcal{D}_{\mathsf{hk}}$
$\mathsf{hp} \leftarrow \omega(\mathsf{hk})$	$\mathsf{hp} \leftarrow \omega(\mathsf{hk})$	$\mathsf{hp} \leftarrow \omega(\mathsf{hk})$	$\mathsf{hp} \leftarrow \omega(\mathsf{hk})$	$\mathsf{hp} \leftarrow \omega(\mathsf{hk})$	$\mathsf{hp} \leftarrow \omega(\mathsf{hk})$
	Sample $(u, w) \in R$	$\tilde{u} \leftarrow\!\!\$\ \mathcal{X} \setminus \mathcal{L}$	$\tilde{u} \leftarrow\!\!\$\ \mathcal{X} \setminus \mathcal{L}, r \leftarrow\!\!\$\ \mathbb{Z}_q$	Sample $(u, w) \in R, r \leftarrow\!\!\$\ \mathbb{Z}_q$	$r \leftarrow\!\!\$\ \mathbb{Z}_q$
$c_\alpha \leftarrow \mathsf{Enc}(\mathsf{hp}, \alpha)$	$c_a \leftarrow (u, \mathcal{H}_{\mathsf{hk}}(u) \cdot \alpha)$	$c_\alpha \leftarrow (\tilde{u}, \mathcal{H}_{\mathsf{hk}}(\tilde{u}) \cdot \alpha)$	$c_\alpha \leftarrow (\tilde{u}, \mathcal{H}_{\mathsf{hk}}(\tilde{u}) \cdot \alpha + r)$	$c_\alpha \leftarrow (u, \mathcal{H}_{\mathsf{hk}}(u) \cdot \alpha + r)$	$c_\alpha \leftarrow \mathsf{Enc}(\mathsf{hp}, \alpha + r)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Table 3: Sequence of games for the indistinguishability of c_α using HSM-CL encryption scheme [10].

Consequently, by the above indistinguishability result, we obtain that the distribution induced by the simulator and the real execution is indistinguishable.

2) $\underline{P_t \text{ corrupted}}$: After agreeing on a message m, the simulator $\mathcal{S}$ is given

$$\left(\text{com, sid, } \left(R_2', \begin{array}{c}\text{prove, sid,}\\ \{\exists k_2' \mid R_2' = g^{k_2'}\}, k_2'\end{array}\right), \left(A, c_\alpha, \begin{array}{c}\text{prove, sid,}\\ \{\exists \alpha \mid c_\alpha = \mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}_t, \alpha) \wedge\\ A = g^\alpha\}, \alpha\end{array}\right)\right)$$

by $\mathcal{A}$. $\mathcal{S}$ verifies that $c_\alpha = \mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}_t, \alpha)$ and $A = g^\alpha$. If the verification fails, $\mathcal{S}$ simulates P_r aborting. $\mathcal{S}$ queries Sign on input $(\mathsf{sid}, m, \alpha)$, and obtains $\sigma' := (R', s')$. $\mathcal{S}$ sends $(R_1' = R'/(R_2' \cdot A), (\mathsf{proof}, \mathsf{sid}, \{\exists k_1' \mid R_1' = g^{k_1'}\}))$ to $\mathcal{A}$, and receives $((\mathsf{decom}, \mathsf{sid}), s_2' = k_2' - e' \cdot x_2')$, where $e' = H(\mathsf{pk} \| R' \| m)$, and x_2' is the value returned by the key generation to $\mathcal{A}$. $\mathcal{S}$ verifies that $R_2' = g^{k_2'}$, and if this fails simulates P_r aborting. The rest of the execution is unchanged.

The simulator is efficient and the distribution induced by the simulated view is identical to the one of the original protocol.

Next, we continue with the simulation of the puzzle solver protocol. Similar to the simulation of the puzzle promise protocol, we define two simulators.

1) $\underline{P_s \text{ corrupted}}$: After agreeing on a message m, $\mathcal{S}$ sends $(\mathsf{com}, \mathsf{sid})$ to $\mathcal{A}$, for a random sid. At some point of the execution $\mathcal{A}$ sends $(c_\alpha'', R_1, (\mathsf{prove}, \{\exists k_1 \mid R_1 = g^{k_1}\}, k_1))$. If $R_1 \neq g^{k_1}$, then $\mathcal{S}$ simulates P_t aborting. $\mathcal{S}$ decrypts c_α'' to obtain γ, and computes $A'' = g^\gamma$. $\mathcal{S}$ queries Sign on input $(\mathsf{sid}, m, \gamma)$, and receives $\sigma := (R, s)$. $\mathcal{S}$ sends

$$\left(\mathsf{decom}, \mathsf{sid}, \left(\begin{array}{l} R_2 = R/(R_1 \cdot A''),\\ \mathsf{proof}, \mathsf{sid},\\ \{\exists k_2 \mid R_2 = g^{k_2}\}\end{array}\right), \begin{array}{l}(s - k_1 + e \cdot x_1), A''\end{array}\right)$$

to $\mathcal{A}$, where $e = H(\mathsf{pk} \| R \| m)$, and x_1 is the value returned by the key generation to $\mathcal{A}$. The rest of the execution is unchanged.

The simulator is efficient and the distribution induced by the simulated view is identical to the one of the original protocol.

2) $\underline{P_t \text{ corrupted}}$: Prior to the interaction the simulator $\mathcal{S}$ is given A' and c_α'. After agreeing on a message m, the simulator $\mathcal{S}$ receives

$$\left(\mathsf{com}, \mathsf{sid}, \left(R_2, \begin{array}{c}\mathsf{prove}, \mathsf{sid},\\ \{\exists k_2 \mid R_2 = g^{k_2}\}, k_2\end{array}\right)\right)$$

by $\mathcal{A}$. $\mathcal{S}$ decrypts c_α' to obtain $\alpha \cdot \beta$, then samples $\tau^* \leftarrow\!\!\$\ \mathbb{Z}_q$, computes $\gamma^* = \alpha \cdot \beta \cdot \tau^*$, $A'' = (A')^{\tau^*}$, and $c_\alpha'' = (c_\alpha')^{\tau^*}$, and it queries Sign on input $(\mathsf{sid}, m, \gamma^*)$. The simulator receives $\sigma := (R, s)$, and sends $(c_\alpha'', R_1 = R/(R_2 \cdot A''), (\mathsf{proof}, \mathsf{sid}, \{\exists k_1 \mid R_1 = g^{k_1}\}))$ to $\mathcal{A}$. $\mathcal{S}$ receives $((\mathsf{decom}, \mathsf{sid}), s_2 = k_2 - e \cdot x_2, A'')$, where $e' = H(\mathsf{pk} \| R \| m)$, and x_2 is the value returned by the key generation to $\mathcal{A}$. If $R_2 \neq g^{k_2}$, then $\mathcal{S}$ simulates P_s aborting, else the rest of the execution is unchanged.

The simulator is efficient and the distribution induced by the simulator is identical to the real execution except for the way c_α'' is computed. However, the same argument about the indistinguishability of HSM-CL encryption scheme from the simulation of the puzzle promise protocol applies here as well. Hence, the distribution induced by the simulated view is identical to the one of the original protocol.

□

Next, we continue with the proof of Lemma C.3. Let cheat be the event that triggers an abort of the experiment in $\mathcal{H}_3$. Assume towards contradiction that $\Pr[\text{cheat} \mid \mathcal{H}_2^*] \geq \frac{1}{\mathsf{poly}(\lambda)}$, then we can construct the following reduction against the strong existential unforgeability of Schnorr signature. The reduction receives as input a public key pk, and samples an index $j \in [1, q]$, where $q \in \mathsf{poly}(\lambda)$ is a bound on the total number of interactions. Let Q be the key generated in the j-th interaction, the reduction sets $Q = \mathsf{pk}$. All the calls to the signing algorithm are redirected to the signing oracle. If the event cheat happens, the reduction returns the corresponding $(\mathsf{pk}^*, m^*, \sigma^* := (R^*, s^*))$, otherwise it aborts.

The reduction is clearly efficient. Assume that j is the index of the interaction where cheat happens. Note that in the case the guess of the reduction is correct we have that $\mathsf{pk}^* = \mathsf{pk}_{r,t}$. Since cheat happens we have that $\mathsf{Verify}_{\mathsf{Schnorr}}(\mathsf{pk}^*, m^*, \sigma^*) = 1$, but $s^* \neq \mathsf{Open}(\Pi, \bar{\alpha})[s]$, where Π and $\bar{\alpha}$ are returned from the puzzle promise and puzzle solver protocols, respectively. Recall that $\bar{\alpha} = \alpha \cdot \beta$ and Open parses Π as (R', s'), where $s' = s_j - \alpha$, for some $\alpha \in \mathbb{Z}_q$, where s_j is the answer of the oracle on the j-th session on input m_j.

Substituting we get

$$\begin{aligned} s^* &\neq \mathsf{Open}(\Pi, \bar{\alpha})[s] \\ &\neq s' + (\bar{\alpha} \cdot \beta^{-1}) \\ &\neq s_j - \alpha + \alpha \cdot \beta \cdot \beta^{-1} \\ &\neq s_j \end{aligned}$$

as expected. Since each message uniquely identifies a session, this implies that $(\mathsf{pk}^*, m^*, \sigma^*)$ is a valid forgery. By assumption this happens with probability at least $\frac{1}{q \cdot \mathsf{poly}(\lambda)}$, which is a contradiction and proves that $\Pr[\text{cheat} \mid \mathcal{H}_2^*] \leq \mathsf{negl}(\lambda)$. □

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

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

Proof. Let $q \in \mathsf{poly}(\lambda)$ be a bound on the total number of interactions. Let cheat denote an event that triggers an abort in $\mathcal{H}_4$, but not in $\mathcal{H}_3$. We prove the indistinguishability of $\mathcal{H}_3$ and $\mathcal{H}_4$ by showing that $\Pr[\text{cheat} \mid \mathcal{H}_3] \leq \mathsf{negl}(\lambda)$. Assume that the converse is true, then we can construct the following reduction against the discrete logarithm problem: On input some $A^* \in \mathbb{G}$ and a public key pk, the reduction guesses a session $j \in [1, q]$. The reduction replaces A from the first message of the puzzle promise protocol with A^*. If P_s is requested to call the puzzle solver protocol, the reduction aborts. At some point of the execution $\mathcal{A}$ outputs some $(\mathsf{pk}^*, m^*, \sigma^* := (R^*, s^*))$. The reduction returns $g^{s^* - s'}$, where s' is part of the output of the puzzle promise protocol.

The reduction is clearly efficient, and whenever j is guessed correctly, the reduction does not abort, and we also have that $\mathsf{pk}^* = \mathsf{pk}_{r,t}$. The event cheat happens only in the case where $\mathsf{Verify}_{\mathsf{Schnorr}}(\mathsf{pk}^*, m^*, \sigma^*) = 1$, but puzzle solver protocol has not been executed. Recall that $s' = s_j - \alpha$ and $A = g^\alpha$, for some $\alpha \in \mathbb{Z}_q$, where s_j is the answer of the oracle on the j-th session on input m_j. We note that we replaced A with the input A^* of the reduction, hence $A = A^*$ in this case. As argued in the proof of Lemma C.3, if $s^* \neq s_j$, then we have an attacker against the strong unforgeability of the signature scheme. Hence, it follows that $s^* = s_j$ with all but negligible probability. Substituting we have

$$\begin{aligned} g^{s^* - s'} &= g^{s^* - (s_j - \alpha)} \\ &= g^\alpha \\ &= A \end{aligned}$$

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

This concludes the proof of Theorem 4.2. □

ECDSA-based Construction. Here we prove Theorem 4.3.

Proof. The sequence of hybrids that we need are identical to the ones used in the proof of the Schnorr-based construction. Hence, here we only prove the indistinguishability of the neighboring experiments which require modifications in the argument. If the argument is the same, then the proof is omitted.

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

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

Proof. Similar to the proof of the Schnorr-based construction, we define an intermediate hybrid.

$\mathcal{H}_2^*$: All the calls to the puzzle promise and puzzle solver protocols are substituted with calls to $\mathcal{F}_{\mathsf{Sign}}$ ideal functionality, defined as follows.

Sign(sid, m, α)

Upon invocation by P_1 and P_2 on input (sid, m, α) :
if some (sid, ·, ·) is already recorded, then ignore the message
else record (sid, m, α)
compute $(r, s) \leftarrow \mathsf{Sig}_{\mathsf{ECDSA}}(\mathsf{sk}_{P_1,P_2}, m)$
return $(r, \min(s \cdot \alpha, -s \cdot \alpha))$

Recall that the key sk_{P_1,P_2} refers to the key established between parties P_1 and P_2 in the call to the $\mathcal{F}_{\mathsf{KGen}}^{\mathsf{ECDSA}}$ functionality.

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

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

Proof. We start with the simulation of the puzzle promise protocol. We define two simulators, depending on whether the adversary is playing the role of P_t or P_r.

1) P_r corrupted: After agreeing on a message m, the simulator $\mathcal{S}$ samples a random $\alpha^* \leftarrow\!\!\$\, \mathbb{Z}_q$, and queries Sign on input (sid, m, α^*), for a random sid, obtains $\sigma' := (r', s')$ and sets $R' = g^{H(m) \cdot (s')^{-1}} \cdot Q^{r' \cdot (s')^{-1}}$. $\mathcal{S}$ computes $A = g^{\alpha^*}$ and $c_\alpha = \mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}_t, \alpha^*)$, where pk_t is the HSM-CL encryption public key of P_t. $\mathcal{S}$ sends ((com, sid), A, c_α, (proof, sid, $\{\exists \alpha^* \mid c_\alpha = \mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}_t, \alpha^*) \wedge A = g^{\alpha^*}\}$)) to $\mathcal{A}$. At some point of the execution $\mathcal{A}$ sends $(R_1', (\text{prove}, \{\exists k_1' \mid R_1' = g^{k_1'}\}, k_1'))$. $\mathcal{S}$ verifies that $R_1' = g^{k_1'}$, and if this is not the case $\mathcal{S}$ simulates P_t aborting. $\mathcal{S}$ computes $c' \leftarrow \mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}_t, k_1' \cdot s' \bmod q)$. $\mathcal{S}$ provides $\mathcal{A}$ with

$$\left(\text{decom}, sid, \left(\begin{array}{l} R_c = (R')^{(k_1')^{-1}}, \\ R_2 = (R_c)^{(\alpha^*)^{-1}}, \\ (\text{proof, sid,} \\ \{\exists k_2 \mid R_2 = g^{k_2}\}), \\ (\text{proof, sid,} \\ \{\exists \alpha^* \mid R_c = R_2^{\alpha^*}\}), \\ (\text{proof, sid,} \\ \quad \{\exists \alpha^* \mid A = g^{\alpha^*} \wedge \\ R_c = R_2^{\alpha^*}\}) \end{array} \right), c' \right).$$

The rest of the execution is unchanged.

The simulator is efficient and the distribution induced by the simulator is identical to the real execution except for the way c_α is computed. The same argument about the distribution of c_α from the proof of Lemma C.4 applies here as well. Hence, indistinguishability follows.

2) P_t corrupted: After agreeing on a message m, the simulator $\mathcal{S}$ is given

$$\left(\text{com}, sid, \begin{pmatrix} R_2', & \text{prove, sid,} \\ & \{\exists k_2' \mid R_2' = g^{k_2'}\}, k_2' \end{pmatrix}, \begin{pmatrix} A, c_\alpha & \text{prove, sid,} \\ & \{\exists \alpha \mid c_\alpha = \text{Enc}_{\text{HE}}(\text{pk}_t, \alpha) \wedge \\ & \quad A = g^\alpha\}, \alpha \end{pmatrix}\right)$$

by $\mathcal{A}$. $\mathcal{S}$ verifies that $c_a = \text{Enc}_{\text{HE}}(\text{pk}_t, \alpha)$ and $A = g^\alpha$. If the verification fails, $\mathcal{S}$ simulates P_r aborting. $\mathcal{S}$ queries Sign on input (sid, m, α), obtains $\sigma' := (r', s')$ and sets $R' = g^{H(m)\cdot(s')^{-1}} \cdot Q^{r'\cdot(s')^{-1}}$. $\mathcal{S}$ sends $(R_1 = (R')^{(k_2')^{-1}\cdot\alpha^{-1}}$, $(\text{proof, sid}, \{\exists k_1 \mid R_1 = g^{k_1}\}))$ to $\mathcal{A}$, and receives

$$\left(\text{decom}, sid, \begin{pmatrix} R_c', & \text{prove, sid,} \\ & \{\exists \alpha \mid R_c' = (R_2')^\alpha\}, \alpha \end{pmatrix}, \begin{pmatrix} A, R_c', & \text{prove, sid,} \\ & \{\exists \alpha \mid A = g^\alpha \wedge \\ & \quad R_c' = (R_2')^\alpha\}, \alpha \end{pmatrix}, c'\right).$$

$\mathcal{S}$ verifies that $R_2' = g^{k_2'}$, $R_c' = (R_2')^\alpha$ and $A = g^\alpha$. If the verification fails $\mathcal{S}$ simulates P_r aborting. $\mathcal{S}$ checks if

$$\text{Dec}_{\text{HE}}(\text{sk}_r, c') = r' \cdot x_2' \cdot \tilde{x_1} \cdot (k_2')^{-1} + H(m) \cdot (k_2')^{-1} \bmod q,$$

where $\tilde{x_1}$ was sampled in the key generation algorithm, and sk_r (the secret key of P_r for the encryption scheme) corresponds to the secret hashing key hk in HSM-CL encryption scheme as described in Appendix B. If the check holds, then the rest of the execution proceeds unchanged, else $\mathcal{S}$ simulates P_r aborting.

The simulator is efficient and the distribution induced by the simulated view is identical to the one of the original protocol due to indistinguishability of the HSM-CL encryption scheme.

Next, we simulate the puzzle solver protocol. We define two simulators, one when P_s is corrupted, and the other one when P_t is corrupted.

1) P_s corrupted: After agreeing on a message m, $\mathcal{S}$ sends (com, sid) to $\mathcal{A}$, for a random sid. At some point of the execution $\mathcal{A}$ sends $(c_\alpha'', R_1, (\text{prove}, \{\exists k_1 \mid R_1 = g^{k_1}\}, k_1))$. If $R_1 \neq g^{k_1}$, then $\mathcal{S}$ simulates P_t aborting. $\mathcal{S}$ decrypts c_α'' to obtain γ, and computes $A'' = g^\gamma$. $\mathcal{S}$ queries Sign on input (sid, m, γ), receives $\sigma := (r, s)$, and sets $R = g^{H(m)\cdot s^{-1}} \cdot Q^{r\cdot s^{-1}}$. $\mathcal{S}$ computes $c \leftarrow \text{Enc}_{\text{HE}}(\text{pk}_t, k_1 \cdot s \bmod q)$, and sends

$$\left(\text{decom}, sid, \begin{pmatrix} R_c = R^{(k_1)^{-1}}, \\ R_2 = (R_c)^{\alpha^{-1}}, \\ (\text{proof, sid,} \\ \{\exists k_2 \mid R_2 = g^{k_2}\}), \\ (\text{proof, sid,} \\ \{\exists \alpha \mid R_c = R_2^\alpha\}), \\ (\text{proof, sid,} \\ \{\exists \alpha \mid A = g^\alpha \wedge \\ R_c = R_2^\alpha\}) \end{pmatrix}, c\right).$$

to $\mathcal{A}$. The rest of the execution is unchanged.

The simulator is efficient and the distribution induced by the simulator is identical to the real execution.

2) P_t corrupted: Prior to the interaction the simulator $\mathcal{S}$ is given A' and c_α'. After agreeing on a message m, the simulator $\mathcal{S}$ receives

$$\left(\text{com}, \text{sid}, \begin{pmatrix} R_2, & \text{prove, sid,} \\ & \{\exists k_2 \mid R_2 = g^{k_2}\}, k_2 \end{pmatrix}\right)$$

by $\mathcal{A}$. $\mathcal{S}$ decrypts c_α' to obtain $\alpha \cdot \beta$, then samples $\tau^* \leftarrow\!\!\$\, \mathbb{Z}_q$, computes $\gamma^* = \alpha \cdot \beta \cdot \tau^*$, $A'' = (A')^{\tau^*}$, and $c_\alpha'' = (c_\alpha')^{\tau^*}$, and it queries Sign on input $(\text{sid}, m, \gamma^*)$. $\mathcal{S}$ receives $\sigma := (r, s)$, and sets $R = g^{H(m)\cdot s^{-1}} \cdot Q^{r\cdot s^{-1}}$. $\mathcal{S}$ sends $(R_1 = R^{(k_2)^{-1}\cdot(\gamma^*)^{-1}}, (\text{proof, sid}, \{\exists k_1 \mid R_1 = g^{k_1}\}))$ to $\mathcal{A}$. $\mathcal{S}$ receives

$$\left(\text{decom}, \text{sid}, \begin{pmatrix} R_c, & \text{prove, sid,} \\ & \{\exists \gamma^* \mid R_c = (R_2)^{\gamma^*}\}, \gamma^* \end{pmatrix}, \begin{pmatrix} A'', R_c, & \text{prove, sid,} \\ & \{\exists \gamma^* \mid A'' = g^{\gamma^*} \wedge \\ & \quad R_c = (R_2)^{\gamma^*}\}, \gamma^* \end{pmatrix}, c\right).$$

$\mathcal{S}$ verifies that $R_2 \neq g^{k_2}$, $R_c = (R_2')^{\gamma^*}$ and $A'' = g^{\gamma^*}$. If the verification fails, then $\mathcal{S}$ simulates P_s aborting. $\mathcal{S}$ checks if

$$\text{Dec}_{\text{HE}}(\text{sk}_s, c) = r \cdot x_2 \cdot \tilde{x_1} \cdot (k_2)^{-1} + H(m) \cdot (k_2)^{-1} \bmod q,$$

where $\tilde{x_1}$ was sampled in the key generation algorithm, and sk_s (the secret key of P_s for the encryption scheme) corresponds to the secret hashing key hk in HSM-CL encryption scheme as described in Appendix B. If the check holds, then the rest of the execution proceeds unchanged, else $\mathcal{S}$ simulates P_s aborting.

The simulator is efficient and the distribution induced by the simulated view is identical to the one of the original protocol due to indistinguishability of the HSM-CL encryption scheme.

□

Next, we continue with the proof of Lemma C.7. Let cheat be the event that triggers an abort of the experiment in $\mathcal{H}_3$. Assume towards contradiction that $\Pr[\text{cheat} \mid \mathcal{H}_2^*] \geq \frac{1}{\text{poly}(\lambda)}$, then we can construct the following reduction against the strong existential unforgeability of ECDSA signature. The reduction receives as input a public key pk, and samples an index $j \in [1, q]$, where $q \in \text{poly}(\lambda)$ is a bound on the total number of interactions. Let Q be the key generated in the j-th interaction, the reduction sets $Q = \text{pk}$. All the calls to the signing algorithm are redirected to the signing oracle. If the event cheat happens, the reduction returns the corresponding $(\text{pk}^*, m^*, \sigma^* := (r^*, s^*))$, otherwise it aborts.

The reduction is clearly efficient. Assume that j is the index of the interaction where cheat happens. Note that in the case the guess of the reduction is correct we have that $\text{pk}^* = \text{pk}_{r,t}$. Since cheat happens we have that $\text{Verify}_{\text{ECDSA}}(\text{pk}^*, m^*, \sigma^*) = 1$, but $s^* \neq \text{Open}(\Pi, \bar{\alpha})[s]$, where Π and $\bar{\alpha}$ are returned from the puzzle promise and puzzle solver protocols, respectively. Recall that $\bar{\alpha} = \alpha \cdot \beta$ and Open parses Π as (r', s'), where $s' = s_j \cdot \alpha$, for some $\alpha \in \mathbb{Z}_q$, where s_j is the answer of the oracle on the j-th session on input m_j.

Substituting we get

$$\begin{aligned} s^* &\neq \mathsf{Open}(\Pi, \bar{\alpha})[s] \\ &\neq s' \cdot (\bar{\alpha} \cdot \beta^{-1})^{-1} \\ &\neq s_j \cdot \alpha \cdot (\alpha \cdot \beta \cdot \beta^{-1})^{-1} \\ &\neq s_j \cdot \alpha \cdot \beta \cdot \beta^{-1} \cdot \alpha^{-1} \\ &\neq s_j \end{aligned}$$

as expected. Since each message uniquely identifies a session, this implies that $(\mathsf{pk}^*, m^*, \sigma^*)$ is a valid forgery. By assumption this happens with probability at least $\frac{1}{q \cdot \mathsf{poly}(\lambda)}$, which is a contradiction and proves that $\Pr[\text{cheat} \mid \mathcal{H}_2^*] \leq \mathsf{negl}(\lambda)$. □

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

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

Proof. Let $q \in \mathsf{poly}(\lambda)$ be a bound on the total number of interactions. Let cheat denote an event that triggers an abort in $\mathcal{H}_4$, but not in $\mathcal{H}_3$. We prove the indistinguishability of $\mathcal{H}_3$ and $\mathcal{H}_4$ by showing that $\Pr[\text{cheat} \mid \mathcal{H}_3] \leq \mathsf{negl}(\lambda)$. Assume that the converse is true, then we can construct the following reduction against the discrete logarithm problem: On input some $A^* \in \mathbb{G}$ and a public key pk, the reduction guesses a session $j \in [1, q]$. The reduction replaces A from the first message of the puzzle promise protocol with A^*. If P_s is requested to call the puzzle solver protocol, the reduction aborts. At some point of the execution $\mathcal{A}$ outputs some $(\mathsf{pk}^*, m^*, \sigma^* := (R^*, s^*))$. The reduction returns $g^{(s^*)^{-1} \cdot s'}$, where s' is part of the output of the puzzle promise protocol.

The reduction is clearly efficient, and whenever j is guessed correctly, the reduction does not abort, and we also have that $\mathsf{pk}^* = \mathsf{pk}_{r,t}$. The event cheat happens only in the case where $\mathsf{Verify}_{\mathsf{ECDSA}}(\mathsf{pk}^*, m^*, \sigma^*) = 1$, but puzzle solver protocol has not been executed. Recall that $s' = s_j \cdot \alpha$ and $A = g^{\alpha}$, for some $\alpha \in \mathbb{Z}_q$, where s_j is the answer of the oracle on the j-th session on input m_j. We note that we replaced A with the input A^* of the reduction, hence $A = A^*$ in this case. As argued in the proof of Lemma C.7, if $s^* \neq s_j$, then we have an attacker against the strong unforgeability of the signature scheme. Hence, it follows that $s^* = s_j$ with all but negligible probability. Substituting we have

$$\begin{aligned} g^{(s^*)^{-1} \cdot s'} &= g^{(s^*)^{-1} \cdot (s_j \cdot \alpha)} \\ &= g^{\alpha} \\ &= A \end{aligned}$$

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

This concludes the proof of Theorem 4.3. □

DLSAG-based Construction. Here we prove Theorem F.1.

Proof. The sequence of hybrids that we need are identical to the ones used in the previous proofs. Therefore, we only prove the indistinguishability of the neighboring experiments which require modifications in the argument.

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

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

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

$\mathcal{H}_2^*$: All the calls to the puzzle promise and puzzle solver protocols are replaced with calls to $\mathcal{F}_{\mathsf{Sign}}$ ideal functionality, which is defined as follows.

$\mathsf{Sign}(\mathsf{sid}, \vec{\mathsf{pk}}, b, m, \alpha)$

Upon invocation by P_1 and P_2 on input $(\mathsf{sid}, \vec{\mathsf{pk}}, b, m, \alpha)$:

if some $(\mathsf{sid}, \cdot, \cdot, \cdot, \cdot)$ is already recorded, then ignore the message

else record $(\mathsf{sid}, \vec{\mathsf{pk}}, b, m, \alpha)$

compute $(s_0, s_1, \ldots, s_{n-1}, R, \hat{\mathcal{J}}, \mathcal{J}, b) \leftarrow \mathsf{Sig}_{\mathsf{DLSAG}}(\mathsf{sk}_{P_1, P_2, b}, \vec{\mathsf{pk}}, m)$

return $(s_0 - \alpha, s_1, \ldots, s_{n-1}, R, \hat{\mathcal{J}}, \mathcal{J}, b)$

We note that the key $\mathsf{sk}_{P_1, P_2, b}$ refers to the part of the previously established dual key between the parties P_1 and P_2 in the call to the $\mathcal{F}_{\mathsf{KGen}}^{\mathsf{DLSAG}}$. Similar to the previous proofs, we have that $P_1 := P_r$ (receiver) in the puzzle promise protocol, whereas $P_1 := P_s$ (sender) in the puzzle solver protocol, and in both protocols $P_2 := P_t$ (tumbler).

Lemma C.11. *For all* PPT *distinguisher* $\mathcal{E}$ *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 simulators for the interactive puzzle promise and puzzle solver protocols. We start with the puzzle promise protocol, and since it is executed between P_t and P_r, we describe two simulators depending on whether the adversary is playing the role of P_t or P_r.

1) P_r corrupted: Prior to the interaction the simulator $\mathcal{S}$ is given the ring $\vec{\mathsf{pk}}' := ((\mathsf{pk}'_{1,0}, \mathsf{pk}'_{1,1}, k'_1), \ldots, (\mathsf{pk}'_{n-1,0}, \mathsf{pk}'_{n-1,1}, k'_{n-1}), (\mathsf{pk}_{r,t,0}, \mathsf{pk}_{r,t,1}, k_{r,t}))$, the bit b, and sets $Q' := \mathsf{pk}_{r,t,1-b}$. After agreeing on a message m, the simulator $\mathcal{S}$ samples a random $\alpha^* \leftarrow_{\$} \mathbb{Z}_q$, queries Sign on input $(\mathsf{sid}, \vec{\mathsf{pk}}', b, m, \alpha^*)$, for a random sid, and obtains $\sigma' := (s'_0, s'_1, \ldots, s'_{n-1}, R', \hat{\mathcal{J}}', \mathcal{J}', b)$. $\mathcal{S}$ computes $A = g^{\alpha^*}$, $A^* = (Q')^{\alpha^* \cdot k_{r,t}}$, and $c_\alpha = \mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}_t, \alpha^*)$, where pk_t is the HSM-CL encryption public key of P_t. The simulator $\mathcal{S}$ sends $((\mathsf{com}, \mathsf{sid}), A, A^*, c_\alpha, (\mathsf{proof}, \mathsf{sid}, \{\exists \alpha^* \mid c_\alpha = \mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}_t, \alpha^*) \wedge A = g^{\alpha^*}\}), (\mathsf{proof}, \mathsf{sid}, \{\exists \alpha^* \mid A = g^{\alpha^*} \wedge A^* = (Q'^{k_{r,t}})^{\alpha^*}\}))$ to $\mathcal{A}$. At some point of the execution $\mathcal{A}$ sends $(\mathcal{J}'_1, \hat{\mathcal{J}}'_1, R'_1, (\mathsf{prove}, \{\exists [s'_0]_r \mid R'_1 = g^{[s'_0]_r} \wedge \hat{\mathcal{J}}'_1 = (Q'^{k_{r,t}})^{[s'_0]_r}\}, [s'_0]_r))$. $\mathcal{S}$ verifies that $R'_1 = g^{[s'_0]_r}$ and $\hat{\mathcal{J}}'_1 = (Q'^{k_{r,t}})^{[s'_0]_r}$. If the verification fails, $\mathcal{S}$ simulates P_t aborting. $\mathcal{S}$ replies with

$$\left(\begin{array}{l} \mathsf{decom}, \mathsf{sid}, \left(\begin{array}{l} (s'_1, \ldots, s'_{n-1}), \\ \mathcal{J}'_2 = \mathcal{J}'/\mathcal{J}'_1, \\ \hat{\mathcal{J}}'_2 = \hat{\mathcal{J}}'/(\hat{\mathcal{J}}'_1 \cdot A^*), \\ R'_2 = R'/(R'_1 \cdot A), \\ \mathsf{proof}, \mathsf{sid}, \\ \{\exists [s'_0]_t \mid R'_2 = g^{[s'_0]_t} \wedge \\ \qquad \hat{\mathcal{J}}'_2 = (Q'^{k_{r,t}})^{[s'_0]_t}\} \end{array} \right), \\ (s'_0 - ([s'_0]_r - h'_{n-1} \cdot \mathsf{sk}_{r,b}), s'_1, \ldots, s'_{n-1}, \mathcal{J}'_2) \end{array} \right)$$

where h'_{n-1} is computed by the simulator as in the original protocol, and $\mathsf{sk}_{r,b} := x_{1,b}$ is the value returned by the key generation to $\mathcal{A}$. The rest of the execution is unchanged.

The simulator is efficient, and the distribution induced by the simulator is identical to the real execution except for the way c_α is computed. However, the same argument from Theorem C.4 about the distribution of c_α applies here. Hence, we obtain that the distribution induced by the simulator and the real execution is indistinguishable.

2) P_t corrupted: Prior to the interaction the simulator $\mathcal{S}$ is given the ring $\vec{\mathsf{pk}}' := ((\mathsf{pk}'_{1,0}, \mathsf{pk}'_{1,1}, k'_1), \ldots, (\mathsf{pk}'_{n-1,0}, \mathsf{pk}'_{n-1,1}, k'_{n-1}), (\mathsf{pk}_{r,t,0}, \mathsf{pk}_{r,t,1}, k_{r,t}))$, the bit b, and sets $Q' := \mathsf{pk}_{r,t,1-b}$. After agreeing on a message m, the simulator $\mathcal{S}$ is given

$$\left(\begin{array}{l} \left(\begin{array}{ll} & \text{prove, sid,} \\ \mathcal{J}'_2, \hat{\mathcal{J}}'_2, R'_2, & \{\exists [s'_0]_t \mid R'_2 = g^{[s'_0]_t} \wedge \\ & \qquad \hat{\mathcal{J}}'_2 = (Q'^{k_{t,r}})^{[s'_0]_t}\}, [s'_0]_t \end{array}\right), \\ \mathsf{com}, \mathsf{sid}, \left(\begin{array}{ll} & \text{prove, sid,} \\ & \{\exists \alpha \mid c_\alpha = \mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}_t, \alpha) \wedge \\ A, A^*, c_\alpha, & \qquad A = g^\alpha\}, \alpha \\ & \text{prove, sid,} \\ & \{\exists \alpha \mid A = g^\alpha \wedge \\ & \qquad A^* = (Q'^{k_{t,r}})^\alpha\}, \alpha \end{array}\right) \end{array}\right)$$

by $\mathcal{A}$. $\mathcal{S}$ verifies that $c_\alpha = \mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}_t, \alpha)$, $A = g^\alpha$, and $A^* = (Q'^{k_{r,t}})^\alpha$. If the verification fails, $\mathcal{S}$ simulates P_r aborting. $\mathcal{S}$ queries Sign on input $(\mathsf{sid}, \vec{\mathsf{pk}}', b, m, \alpha)$, and obtains $\sigma' := (s'_0, s'_1, \ldots, s'_{n-1}, R', \hat{\mathcal{J}}', \mathcal{J}', b)$. $\mathcal{S}$ sends $(R'_1 = R'/(R'_2 \cdot A), \hat{\mathcal{J}}'_1 = \hat{\mathcal{J}}'/(\hat{\mathcal{J}}'_2 \cdot A^*), \mathcal{J}'_1 = \mathcal{J}'/\mathcal{J}'_2, (\text{proof, sid, } \{\exists [s'_0]_r \mid R'_1 = g^{[s'_0]_r} \wedge \hat{\mathcal{J}}'_1 = (Q'^{k_{r,t}})^{[s'_0]_r}\}))$ to $\mathcal{A}$, and receives $((\text{decom, sid}), \sigma'_2 := ([s_0]_t, s'_1, \ldots, s'_2, \mathcal{J}'_2))$. $\mathcal{S}$ verifies that $R'_2 = g^{[s'_0]_t}$ and $\hat{\mathcal{J}}'_2 = (Q'^{k_{r,t}})^{[s'_0]_t}$. If the verification fails, $\mathcal{S}$ simulates P_r aborting, otherwise $\mathcal{S}$ continues rest of the execution unchanged.

The simulator is efficient and the distribution induced by the simulated view is identical to the one of the original protocol.

Next, we continue with the simulation of the puzzle solver protocol. Similar to the simulation of the puzzle promise protocol, we define two simulators.

1) P_s corrupted: Prior to the interaction the simulator $\mathcal{S}$ is given the ring $\vec{\mathsf{pk}} := ((\mathsf{pk}_{1,0}, \mathsf{pk}_{1,1}, k_1), \ldots, (\mathsf{pk}_{n-1,0}, \mathsf{pk}_{n-1,1}, k_{n-1}), (\mathsf{pk}_{s,t,0}, \mathsf{pk}_{s,t,1}, k_{s,t}))$, the bit b, and sets $Q := \mathsf{pk}_{s,t,1-b}$. After agreeing on a message m, $\mathcal{S}$ sends (com, sid) to $\mathcal{A}$, for a random sid. At some point of the execution $\mathcal{A}$ sends $(c''_\alpha, \mathcal{J}_1, \hat{\mathcal{J}}_1, R_1, (\text{prove}, \{\exists [s'_0]_s \mid R_1 = g^{[s'_0]_s} \wedge \hat{\mathcal{J}}_1 = (Q^{k_{s,t}})^{[s'_0]_s}\}, [s'_0]_s))$. If $R_1 \neq g^{[s'_0]_s}$ or $\hat{\mathcal{J}}_1 \neq (Q^{k_{s,t}})^{[s'_0]_s}$, then $\mathcal{S}$ simulates P_t aborting. $\mathcal{S}$ decrypts c''_α to obtain γ, and computes $A'' = g^\gamma, (A^*)'' = (Q^{k_{s,t}})^\gamma$. $\mathcal{S}$ queries Sign on input $(\mathsf{sid}, \vec{\mathsf{pk}}, b, m, \gamma)$, and obtains $\sigma := (s_0, s_1, \ldots, s_{n-1}, R, \hat{\mathcal{J}}, \mathcal{J}, b)$. $\mathcal{S}$ sends

$$\left(\begin{array}{l} \mathsf{decom}, \mathsf{sid}, \left(\begin{array}{l} (s_1, \ldots, s_{n-1}), \\ \mathcal{J}_2 = \mathcal{J}/\mathcal{J}_1 \\ \hat{\mathcal{J}}_2 = \hat{\mathcal{J}}/(\hat{\mathcal{J}}_1 \cdot (A^*)''), \\ R_2 = R/(R_1 \cdot A''), \\ \text{proof, sid,} \\ \{\exists [s'_0]_t \mid R_1 = g^{[s'_0]_t} \wedge \\ \qquad \hat{\mathcal{J}}_1 = (Q^{k_{s,t}})^{[s'_0]_t}\} \end{array}\right), A'', (A^*)'', \\ (s_0 - ([s'_0]s - h_{n-1} \cdot \mathsf{sk}_{s,b}), s_1, \ldots, s_{n-1}, \mathcal{J}_2) \end{array}\right)$$

to $\mathcal{A}$, where h'_{n-1} is computed by the simulator as in the original protocol, and the key $\mathsf{sk}_{s,b} := x_{1,b}$ is the one returned by the key generation to $\mathcal{A}$. The rest of the execution is unchanged.

The simulator is efficient and the distribution induced by the simulated view is identical to the one of the original protocol.

2) P_t corrupted: Prior to the interaction the simulator $\mathcal{S}$ is given A', the ciphertext c'_α, the ring $\vec{\mathsf{pk}} := ((\mathsf{pk}_{1,0}, \mathsf{pk}_{1,1}, k_1), \ldots, (\mathsf{pk}_{n-1,0}, \mathsf{pk}_{n-1,1}, k_{n-1}), (\mathsf{pk}_{s,t,0}, \mathsf{pk}_{s,t,1}, k_{s,t}))$, the bit b, and sets $Q := \mathsf{pk}_{s,t,1-b}$. After agreeing on a message m, the simulator $\mathcal{S}$ receives

$$\left(\mathsf{com}, \mathsf{sid}, \left(\begin{array}{ll} & \text{prove, sid,} \\ \mathcal{J}_2, \hat{\mathcal{J}}_2, R_2, & \{\exists [s'_0]_t \mid R_2 = g^{[s'_0]_t} \wedge \\ & \qquad \hat{\mathcal{J}}_2 = (Q^{k_{s,t}})^{[s'_0]_t}\}, [s'_0]_t \end{array}\right)\right)$$

by $\mathcal{A}$. $\mathcal{S}$ decrypts c'_α to obtain $\alpha \cdot \beta$, then samples a random $\tau^* \leftarrow_\$ \mathbb{Z}_q$, computes $\gamma^* = \alpha \cdot \beta \cdot \tau^*$, and $c''_\alpha = (c'_\alpha)^{\tau^*}$, and it queries Sign on input $(\mathsf{sid}, \vec{\mathsf{pk}}, b, m, \gamma^*)$. The simulator receives $\sigma := (s_0, s_1, \ldots, s_{n-1}, R, \hat{\mathcal{J}}, \mathcal{J}, b)$, and sends $(c''_\alpha, R_1 = R/(R_2 \cdot A''), \hat{\mathcal{J}}_1 = \hat{\mathcal{J}}/(\hat{\mathcal{J}}_2 \cdot (A^*)''), \mathcal{J}_1 = \mathcal{J}/\mathcal{J}_2, (\text{proof, sid, } \{\exists [s'_0]_s \mid R_1 = g^{[s'_0]_s} \wedge \hat{\mathcal{J}}_1 = (Q^{k_{s,t}})^{[s'_0]_s}\}))$ to $\mathcal{A}$. $\mathcal{S}$ receives $((\text{decom, sid}), \sigma_2 := ([s_0]_t, s_1, \ldots, s_{n_1}, \mathcal{J}_2))$, $\mathcal{S}$ verifies that $R_2 = g^{[s'_0]_t}$ and $\hat{\mathcal{J}}_2 = (Q^{k_{s,t}})^{[s'_0]_t}$. If the verification fails, then $\mathcal{S}$ simulates P_s aborting. $\mathcal{S}$ replies with $(s_0 - [s_0]_t, s_1, \ldots, s_{n-1}, \mathcal{J}_1)$. The rest of the execution is unchanged.

The simulator is efficient and the distribution induced by the simulator is identical to the real execution except for the way c''_α is computed. However, the same argument about the indistinguishability of HSM-CL encryption scheme from the simulation of the puzzle promise protocol applies here as well. Hence, the distributions induced by the simulated view is identical to the one of the original protocol.

□

Next, we continue with the proof of Lemma C.10. Let cheat be the event that triggers an abort of the experiment in $\mathcal{H}_3$. Assume towards contradiction that $\Pr[\text{cheat} \mid \mathcal{H}_2^*] \geq \frac{1}{\mathsf{poly}(\lambda)}$, then we can construct the following reduction against the existential unforgeability of DLSAG signature. The reduction receives as input a ring $\vec{\mathsf{pk}}$, and a public key pk, such that $\mathsf{pk} \in \vec{\mathsf{pk}}$ and samples an index $j \in [1, q]$, where $q \in \mathsf{poly}(\lambda)$ is a bound on the total number of interactions. Let Q be the key generated in the j-th interaction, the reduction sets $Q = \mathsf{pk}$. All the calls to the signing algorithm are redirected to the signing oracle. If the event cheat happens, the reduction returns the corresponding $(\mathsf{pk}^*, m^*, \sigma^* := (s_0^*, s_1^*, \ldots, s_{n-1}^*, R^*, \hat{\mathcal{J}}^*, \mathcal{J}^*, b^*))$, otherwise it aborts.

The reduction is clearly efficient. Assume that j is the index of the interaction where cheat happens. Note that in the case the guess of the reduction is correct we have that $\mathsf{pk}^* = \mathsf{pk}_{r,t,1-b^*}$ and $\mathsf{pk}^* \in \vec{\mathsf{pk}}$.

Since cheat happens we have that $\mathsf{Verify}_{\mathsf{DLSAG}}(\vec{\mathsf{pk}}, m^*, \sigma^*) = 1$, but $s_0^* \neq \mathsf{Open}(\Pi, \bar{\alpha})[s_0]$ (here we replaced s from the hybrid with s_0 as that is the s value of interest in DLSAG signatures), where Π and $\bar{\alpha}$ are returned from the puzzle promise and puzzle solver protocols, respectively. Recall that $\bar{\alpha} = \alpha \cdot \beta$ and Open parses Π as $(s_0', s_1', \ldots, s_{n_1}', R', \hat{\mathcal{J}}', \mathcal{J}, b)$, where $s_0' = s_0^{(j)} - \alpha$, for some $\alpha \in \mathbb{Z}_q$, where $s_0^{(j)}$ comes from the answer of the oracle on the j-th session on input m_j. Substituting we get

$$\begin{aligned} s_0^* &\neq \mathsf{Open}(\Pi, \bar{\alpha})[s_0] \\ &\neq s_0' + (\bar{\alpha} \cdot \beta^{-1}) \\ &\neq s_0^{(j)} - \alpha + \alpha \cdot \beta \cdot \beta^{-1} \\ &\neq s_0^{(j)} \end{aligned}$$

as expected. Since each message uniquely identifies a session, this implies that $(\mathsf{pk}^*, m^*, \sigma^*)$ is a valid forgery. By assumption this happens with probability at least $\frac{1}{q \cdot \mathsf{poly}(\lambda)}$, which is a contradiction and proves that $\Pr[\mathsf{cheat} \mid \mathcal{H}_2^*] \leq \mathsf{negl}(\lambda)$. □

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

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

Proof. Let $q \in \mathsf{poly}(\lambda)$ be a bound on the total number of interactions. Let cheat denote an event that triggers an abort in $\mathcal{H}_4$, but not in $\mathcal{H}_3$. We prove the indistinguishability of $\mathcal{H}_3$ and $\mathcal{H}_4$ by showing that $\Pr[\mathsf{cheat} \mid \mathcal{H}_3] \leq \mathsf{negl}(\lambda)$. Assume that the converse is true, then we can construct the following reduction against the discrete logarithm problem: On input some $A^* \in \mathbb{G}$, a ring $\vec{\mathsf{pk}}$, and a public key pk, such that $\mathsf{pk} \in \vec{\mathsf{pk}}$ the reduction guesses a session $j \in [1, q]$. The reduction replaces A from the first message of the puzzle promise protocol with A^*. If P_S is requested to call the puzzle solver protocol, the reduction aborts. At some point of the execution $\mathcal{A}$ outputs some $(\mathsf{pk}^*, m^*, \sigma^* := (s_0^*, s_1^*, \ldots, s_{n-1}^*, R^*, \hat{\mathcal{J}}^*, \mathcal{J}^*, b^*))$. The reduction returns $g^{s_0^* - s_0'}$, where s_0' is part of the output of the puzzle promise protocol.

The reduction is clearly efficient, and whenever j is guessed correctly, the reduction does not abort, and we also have that $\mathsf{pk}^* = \mathsf{pk}_{r,t,1-b^*}$. The event cheat happens only in the case where $\mathsf{Verify}_{\mathsf{DLSAG}}(\vec{pk}, m^*, \sigma^*) = 1$, but puzzle solver protocol has not been executed. Recall that $s_0' = s_0^{(j)} - \alpha$ and $A = g^\alpha$, for some $\alpha \in \mathbb{Z}_q$, where $s_0^{(j)}$ is part of the answer of the oracle on the j-th session on input m_j. We note that we replaced A with the input A^* of the reduction, hence $A = A^*$ in this case. As argued in the proof of Lemma C.10, if $s_0^* \neq s_0^{(j)}$, then we have an attacker against the unforgeability of the signature scheme. Hence, it follows that $s_0^* = s_0^{(j)}$ with all but negligible probability. Substituting we have

$$\begin{aligned} g^{s_0^* - s_0'} &= g^{s_0^* - (s_0^{(j)} - \alpha)} \\ &= g^\alpha \\ &= A \end{aligned}$$

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

This concludes the proof of Theorem F.1. □

D SECURITY ANALYSIS OF TRILERO

Multi-session Extension. Composition theorem requires that each call of every ideal functionality spawns an independent instance of the corresponding functionality. However, our $\mathcal{F}_{\mathsf{A^2L}}$ functionality formally requires a joint state between sessions. More precisely, the KGen protocols that are used for establishing pairwise links are shared between multiple puzzle promise/puzzle solver instances, which might potentially result in shared keys between the different instances of A^{2}L that realize payment channels. Therefore, we need to rely on composition with joint state (as discussed in [9]), where the authors state a stronger version of the composition theorem, called JUC, which accounts for joint state and randomness across protocol sessions.

In order to satisfy the conditions for the JUC theorem to apply, we must first argue that our protocol realizes a stronger ideal functionality $\tilde{\mathcal{F}}_{\mathsf{A^2L}}$, that makes only independent calls to the underlying interfaces. More precisely, we need to argue for each of the previously presented concrete realizations of $\mathcal{F}_{\mathsf{A^2L}}$ that a parallel composition of those protocols realizes the functionality $\tilde{\mathcal{F}}_{\mathsf{A^2L}}$ (with all instances of the protocols sharing the same KGen, but running independently otherwise). We show this in the following lemmas.

Lemma D.1. *Let* COM *be a secure commitment scheme, let* NIZK *be a non-interactive zero-knowledge scheme, and let* $\widetilde{\mathbb{L}_{\mathsf{Schnorr}}}^{\mathsf{KGen}}$ *be the multi-session extension of the protocol described in Figures 6, 7 and 8, using a shared* KGen *algorithm that realizes* $\mathcal{F}_{\mathsf{KGen}}^{\mathsf{Schnorr}}$. *If Schnorr signatures are strongly existentially unforgeable and HSM-CL encryption is* IND-CPA *secure, then* $\widetilde{\mathbb{L}_{\mathsf{Schnorr}}}^{\mathsf{KGen}}$, *UC-realizes the ideal functionality* $\tilde{\mathcal{F}}_{\mathsf{A^2L}}$ *in the* $(\mathcal{F}_{\mathsf{KGen}}^{\mathsf{Schnorr}}, \mathcal{F}_{\mathsf{anon}}, \mathcal{F}_{\mathsf{smt}}, \mathcal{F}_{\mathsf{syn}})$*-hybrid model.*

Proof. It is trivial to see that the $\mathcal{F}_{\mathsf{KGen}}^{\mathsf{Schnorr}}$ functionality itself is stateless, and therefore, consecutive invocations of $\mathcal{F}_{\mathsf{KGen}}^{\mathsf{Schnorr}}$ are indistinguishable from the invocations of fresh instances of the functionality. Thus, for multiple protocols, it is identical to query the same $\mathcal{F}_{\mathsf{KGen}}^{\mathsf{Schnorr}}$ instance or to work on independent copies (note that the same property carries over to protocols realizing this functionality). Consequently, $\widetilde{\mathbb{L}_{\mathsf{Schnorr}}}^{\mathsf{KGen}}$ is indistinguishable from the multi-session extension of $\mathbb{L}_{\mathsf{Schnorr}}$ using independent KGen copies that realize $\mathcal{F}_{\mathsf{KGen}}^{\mathsf{Schnorr}}$. Hence, the claim follows from the composition theorem [7] and Theorem 4.2. □

Lemma D.2. *Let* COM *be a secure commitment scheme, let* NIZK *be a non-interactive zero-knowledge scheme, and let* $\widetilde{\mathbb{L}_{\mathsf{ECDSA}}}^{\mathsf{KGen}}$ *be the multi-session extension of the protocol described in Figures 9, 10 and 11, using a shared* KGen *algorithm that realizes* $\mathcal{F}_{\mathsf{KGen}}^{\mathsf{ECDSA}}$. *If ECDSA signatures are strongly existentially unforgeable and HSM-CL encryption is* IND-CPA *secure, then* $\widetilde{\mathbb{L}_{\mathsf{ECDSA}}}^{\mathsf{KGen}}$, *UC-realizes the ideal functionality* $\tilde{\mathcal{F}}_{\mathsf{A^2L}}$ *in the* $(\mathcal{F}_{\mathsf{KGen}}^{\mathsf{ECDSA}}, \mathcal{F}_{\mathsf{anon}}, \mathcal{F}_{\mathsf{smt}}, \mathcal{F}_{\mathsf{syn}})$*-hybrid model.*

Proof. $\mathcal{F}_{\mathsf{KGen}}^{\mathsf{ECDSA}}$ satisfies the same independence properties as $\mathcal{F}_{\mathsf{KGen}}^{\mathsf{Schnorr}}$, hence, the same argument as for Lemma D.1 applies. □

Lemma D.3. *Let* COM *be a secure commitment scheme, let* NIZK *be a non-interactive zero-knowledge scheme, and let* $\widehat{\mathbb{L}_{\text{DLSAG}}}^{\text{KGen}}$ *be the multi-session extension of the protocol described in Figures 12, 13 and 14, using a shared* KGen *algorithm that realizes* $\mathcal{F}_{\text{KGen}}^{\text{DLSAG}}$. *If DLSAG signatures are existentially unforgeable and HSM-CL encryption is* IND-CPA *secure, then* $\widehat{\mathbb{L}_{\text{DLSAG}}}^{\text{KGen}}$, *UC-realizes the ideal functionality* $\tilde{\mathcal{F}}_{\text{A}^2\text{L}}$ *in the* $(\mathcal{F}_{\text{KGen}}^{\text{DLSAG}}, \mathcal{F}_{\text{anon}}, \mathcal{F}_{\text{smt}}, \mathcal{F}_{\text{syn}})$*-hybrid model.*

Proof. $\mathcal{F}_{\text{KGen}}^{\text{DLSAG}}$ satisfies the same independence properties as $\mathcal{F}_{\text{KGen}}^{\text{Schnorr}}$, hence, the same argument as for Lemma D.1 applies. □

Now, we are in a position to prove that Trilero UC-realizes $\mathcal{F}_{\text{PCH}}$ as defined in Theorem 4.1.

Proof. The proof consists of the observation that the ideal functionality $\mathcal{F}_{\text{A}^2\text{L}}$ enforces balance security and unlinkability properties of a PCH (that are defined in Section 3.2). Balance security is guaranteed due to the atomicity of $\mathcal{F}_{\text{A}^2\text{L}}$, meaning either all the balances are updated or none of them. This ensures that no party loses or gains more than it should. As was discussed in Section 3.3, $\mathcal{F}_{\text{A}^2\text{L}}$ satisfies the unlinkability property, hence, the same argument for unlinkability applies here too. Also, note that the only information that is sent outside of $\mathcal{F}_{\text{A}^2\text{L}}$ consists of amounts and timeouts, and these values are chosen exactly as described in $\mathcal{F}_{\text{PCH}}$. Furthermore, it is sufficient to argue about the individual copies of $\mathcal{F}_{\text{A}^2\text{L}}$ in isolation by the JUC theorem [9]. As was shown in Lemmas D.1 and D.2, the multi-session extended ideal functionality $\tilde{\mathcal{F}}_{\text{A}^2\text{L}}$ is realized by our instantiations, and therefore, the JUC theorem allows us to complete the analysis assuming independent copies of $\mathcal{F}_{\text{A}^2\text{L}}$ running in parallel. □

E LIMITATIONS OF UNLINKABILITY

In this section, we discuss the unlinkability limitations inherent to the PCH setting, and thus also affecting Trilero. We remark that these limitations are inherent to any tumbler protocol, as shown for instance in TumbleBit [27].

Epoch Anonymity. Assume that P_t executes the puzzle promise protocol with k parties during an epoch. If within the same epoch k payments successfully complete, then the anonymity set is of size k since there exist k compatible interaction graphs, as defined in Section 3.2.

It is however not always the case that k is equal to the total number of parties. The exact anonymity level can be established only at the end of the epoch depending on the number of successful puzzle promise and puzzle solver protocols. For instance, anonymity is reduced by 1 if P_t aborts a a payment made by a party P_s. The payment between P_s and P_r would be the only one failing, thereby showing that P_r was the expected receiver. It is important to note that P_s does not lose coins as P_t obtains a valid channel update only if it cooperates in solving the puzzle.

Tumbler/Receiver Collusion. The tumbler P_t and the receiver P_r can collude to learn the identity of the sender P_s. Intuitively, this type of attack is only useful if P_r can be paid by P_s without learning its true identity. We partially address this collusion in our constructions by letting P_s randomize the puzzle it receives from P_r. However, P_r can still send a maliciously constructed puzzle (more precisely, an invalid puzzle or a non-randomized puzzle) to P_s, which can cause an abort or leak information to P_t during the execution of the puzzle solver protocol between P_s and P_t. This in turn can allow P_t to link that P_s was the party that intended to pay P_r. One possible mitigation to this is to force P_r to give a zero-knowledge proof to P_s that the puzzle it sends is a valid randomized puzzle.

Intersection Attack. The aforementioned k-anonymity notion is broadly used in mixing protocols with an intermediate P_t. However, P_t can further reduce the anonymity set. At any epoch, P_t can record the set of senders and receivers that participate in the puzzle solver and puzzle promise protocols respectively. Then, P_t can correlate this information across epochs to de-anonymize users across epochs (e.g., using frequency analysis).

Ceiling Attack. The amount of payments that a certain P_r can receive during a certain epoch is limited by the balance at the channel between P_t and P_r. If such channel is exhausted (i.e., $(P_t, P_r)^{0, x_{P_r}}$), P_t can deterministically derive the fact that P_r is not a potential receiver within the current epoch.

Attacks with Auxiliary Information.. Our notion of unlinkability does not consider auxiliary information available to P_t. Assume that P_t knows that a certain P_r has a shop online selling a product for a value $2 \cdot \text{amt}$. Further assume that during an epoch, P_t executes the puzzle promise protocol only once on every channel except for one P_r for which the puzzle promise protocol is executed twice. Similarly, P_t could observe that there exists a single P_s executing twice the puzzle solve protocol, allowing P_t to link the pair P_s, P_r. As indicated in [27], this type of attacks (called Potato attack in [27]) could be mitigated by aggregating payments or adding noise à la differential privacy.

F A^{2}L FOR MONERO

In the following, we present how to construct A^{2}L for Monero. Our construction makes use of the recent ring signature scheme, called DLSAG, introduced by Moreno-Sanchez et al. [41]. The puzzle promise and puzzle solver protocols can be seen in Figure 12 and Figure 13, respectively. For simplicity we assume that the ring $\vec{pk}$ and the parity bit b indicating the chosen key are given as public parameters. The values inside brackets denote the shares. For instance, $[s_0']_r$ and $[s_0']_t$ denote the share of P_r and P_t, respectively, for the value s_0. We consider two hash functions: (i) H_s takes as input a bitstring and outputs a scalar (i.e., $H_s\colon \{0,1\}^* \to \mathbb{Z}_q$); (ii) H_p takes as input a bitstring and outputs an element of $\mathbb{G}$ (i.e., $H_p\colon \{0,1\}^* \to \mathbb{G}$). Q' and Q denote the chosen key (according to the bit b) of the dual key created when opening the payment channels between P_r/P_t and P_s/P_t, respectively. For more information about how to open payment channels and make conditional payments in Monero we refer the reader to [41].

We define the security of the DLSAG-based construction in Theorem F.1 and formally prove it in Appendix C.

Theorem F.1. *Let* COM *be a secure commitment scheme and let* NIZK *be a non-interactive zero-knowledge scheme. If DLSAG signature is existentially unforgeable and HSM-CL encryption is* IND-CPA *secure, then the construction in Figures 12, 13 and 14, UC-realizes the ideal functionality* $\mathcal{F}_{\text{A}^2\text{L}}$ *in the* $(\mathcal{F}_{\text{KGen}}^{\text{DLSAG}}, \mathcal{F}_{\text{anon}}, \mathcal{F}_{\text{smt}}, \mathcal{F}_{\text{syn}})$*-hybrid model.*

Public parameters: $\mathbb{G}, g, q$, a bit b, a ring $\vec{\mathsf{pk}}' := ((\mathsf{pk}'_{1,0}, \mathsf{pk}'_{1,1}, k'_1), \ldots, (\mathsf{pk}'_{n-1,0}, \mathsf{pk}'_{n-1,1}, k'_{n-1}), (\mathsf{pk}_{r,t,0}, \mathsf{pk}_{r,t,1}, k_{r,t}))$, a message m'

$\mathsf{PuzzlePromise}_{P_t}(\mathsf{sk}_{t,b}, Q' := \mathsf{pk}_{r,t,1-b})$		$\mathsf{PuzzlePromise}_{P_r}(\mathsf{sk}_{r,b}, Q' := \mathsf{pk}_{r,t,1-b})$
$[s'_0]_t \leftarrow\!\!\$\, \mathbb{Z}_q; \vec{s'} := (s'_1, \ldots, s'_{n-1}) \leftarrow\!\!\$\, \mathbb{Z}_q$		
$\mathcal{J}'_2 \leftarrow (Q')^{\mathsf{sk}_{t,b} \cdot k_{r,t}}; \hat{\mathcal{J}}'_2 \leftarrow (Q')^{[s'_0]_t \cdot k_{r,t}}$		
$R'_2 \leftarrow g^{[s'_0]_t}; \alpha \leftarrow\!\!\$\, \mathbb{Z}_q; c_\alpha \leftarrow \mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}_t, \alpha)$		
$A \leftarrow g^\alpha; A^* \leftarrow (Q')^{\alpha \cdot k_{r,t}}$		
$\pi_\alpha \leftarrow \mathsf{P}_{\mathsf{NIZK}}(\{\exists \alpha \mid c_\alpha = \mathsf{Enc}_{\mathsf{HE}}(\mathsf{pk}_t, \alpha) \wedge A = g^\alpha\}, \alpha)$		
$\pi_A \leftarrow \mathsf{P}_{\mathsf{NIZK}}(\{\exists \alpha \mid A = g^\alpha \wedge A^* = (Q'^{k_{r,t}})^\alpha\}, \alpha)$		
$\pi'_2 \leftarrow \mathsf{P}_{\mathsf{NIZK}}(\{\exists [s'_0]_t \mid R'_2 = g^{[s'_0]_t}$		
$\quad \wedge\ \hat{\mathcal{J}}'_2 = (Q'^{k_{r,t}})^{[s'_0]_t}\}, [s'_0]_t)$		
$(\mathsf{com}, \mathsf{decom}) \leftarrow \mathsf{P}_{\mathsf{COM}}(\vec{s'}, \mathcal{J}'_2, (\hat{\mathcal{J}}'_2, R'_2, \pi'_2))$		
	$\xrightarrow{\mathsf{com}, \pi_\alpha, \pi_A, c_\alpha, A, A^*}$	
		If $\mathsf{V}_{\mathsf{NIZK}}(\pi_A, (A, A^*)) \neq 1$
		$\vee\ \mathsf{V}_{\mathsf{NIZK}}(\pi_\alpha, (c_\alpha, A)) \neq 1$ then abort
		$[s'_0]_r \leftarrow\!\!\$\, \mathbb{Z}_q; R'_1 \leftarrow g^{[s'_0]_r}$
		$\mathcal{J}'_1 \leftarrow (Q')^{\mathsf{sk}_{r,b} \cdot k_{r,t}}; \hat{\mathcal{J}}'_2 \leftarrow (Q')^{[s'_0]_r \cdot k_{r,t}}$
		$\pi'_1 \leftarrow \mathsf{P}_{\mathsf{NIZK}}(\{\exists [s'_0]_r \mid R'_1 = g^{[s'_0]_r}$
		$\quad \wedge\ \hat{\mathcal{J}}'_1 = (Q'^{k_{r,t}})^{[s'_0]_r}\}, [s'_0]_r)$
	$\xleftarrow{\mathcal{J}'_1, \hat{\mathcal{J}}'_1, R'_1, \pi'_1}$	
If $\mathsf{V}_{\mathsf{NIZK}}(\pi'_1, (R'_1, \hat{\mathcal{J}}'_1)) \neq 1$ then abort		
$\mathcal{J}' \leftarrow \mathcal{J}'_1 \cdot \mathcal{J}'_2; R' \leftarrow R'_1 \cdot R'_2 \cdot A; \hat{\mathcal{J}} \leftarrow \hat{\mathcal{J}}'_1 \cdot \hat{\mathcal{J}}'_2 \cdot A^*$		
$h'_0 \leftarrow H_s(m' \Vert R' \Vert \hat{\mathcal{J}}')$		
$\forall i \in \{1, \ldots, n-1\}:$		
$\quad L'_i \leftarrow g^{s'_i} \cdot \mathsf{pk}_{i,b}^{'h'_{i-1}}; R'_i \leftarrow \mathsf{pk}_{i,1-b}^{'s'_i \cdot k'_i} \cdot \mathcal{J}'^{h_{i-1}}$		
$\quad h'_i \leftarrow H_s(m' \Vert L'_i \Vert R'_i)$		
$[s_0]_t \leftarrow [s'_0]_t - h'_{n-1} \cdot \mathsf{sk}_{t,b}$		
	$\xrightarrow{(\mathsf{decom}, \vec{s'}, \mathcal{J}'_2, \hat{\mathcal{J}}'_2, R'_2, \pi'_2), [s_0]_t}$	
		If $\mathsf{V}_{\mathsf{COM}}(\mathsf{com}, \mathsf{decom}, (\vec{s'}, \mathcal{J}'_2, \hat{\mathcal{J}}'_2, R'_2, \pi'_2)) \neq 1$ then abort
		If $\mathsf{V}_{\mathsf{NIZK}}(\pi'_2, (R'_2, \hat{\mathcal{J}}'_2)) \neq 1$ then abort
		$\mathcal{J}' \leftarrow \mathcal{J}'_1 \cdot \mathcal{J}'_2; R' \leftarrow R'_1 \cdot R'_2 \cdot A; \hat{\mathcal{J}}' \leftarrow \hat{\mathcal{J}}'_1 \cdot \hat{\mathcal{J}}'_2 \cdot A^*$
		$h'_0 \leftarrow H_s(m' \Vert R' \Vert \hat{\mathcal{J}}')$
		$\forall i \in \{1, \ldots, n-1\}:$
		$\quad L'_i \leftarrow g^{s'_i} \cdot \mathsf{pk}_{i,b}^{'h'_{i-1}}; R'_i \leftarrow \mathsf{pk}_{i,1-b}^{'s'_i \cdot k'_i} \cdot \mathcal{J}'^{h_{i-1}}$
		$\quad h'_i \leftarrow H_s(m' \Vert L'_i \Vert R'_i)$
		$[s_0]_r \leftarrow [s'_0]_r - h'_{n-1} \cdot \mathsf{sk}_{r,b}$
		$\beta \leftarrow\!\!\$\, \mathbb{Z}_q; c'_\alpha \leftarrow (c_\alpha)^\beta; A' \leftarrow A^\beta$
	$\xleftarrow{[s_0]_r}$	
		Send $\ell := (A', c'_\alpha)$ to P_s
		$\sigma' := ([s_0]_t + [s_0]_r, s'_1, \ldots, s'_{n-1}, R', \hat{\mathcal{J}}', \mathcal{J}', b)$
return $\sigma := ([s_0]_t + [s_0]_r + \alpha, s'_1, \ldots, s'_{n-1}, R', \hat{\mathcal{J}}', \mathcal{J}', b)$		**return** $(\Pi := (\beta, (\vec{\mathsf{pk}}', m', \sigma'), \ell)$

Figure 12: Puzzle promise protocol of DLSAG-based construction.

Public parameters: $\mathbb{G}, g, q$, a bit b, a ring $\vec{\mathsf{pk}} := ((\mathsf{pk}_{1,0}, \mathsf{pk}_{1,1}, k_1), \ldots, (\mathsf{pk}_{n-1,0}, \mathsf{pk}_{n-1,1}, k_{n-1}), (\mathsf{pk}_{s,t,0}, \mathsf{pk}_{s,t,1}, k_{s,t}))$, a message m

$\mathsf{PuzzleSolver}_{P_s}(\mathsf{sk}_{s,b}, Q := \mathsf{pk}_{s,t,1-b}, \ell := (A', c'_\alpha))$		$\mathsf{PuzzleSolver}_{P_t}(\mathsf{sk}_{t,b}, Q := \mathsf{pk}_{s,t,1-b})$
		$[s'_0]_t \leftarrow\$ \mathbb{Z}_q; \vec{s} := (s_1, \ldots, s_{n-1}) \leftarrow\$ \mathbb{Z}_q$
		$\mathcal{J}_2 \leftarrow Q^{\mathsf{sk}_{t,b} \cdot k_{s,t}}; \hat{\mathcal{J}}_2 \leftarrow Q^{[s'_0]_t \cdot k_{s,t}}; R_2 \leftarrow g^{[s'_0]_t}$
		$\pi_2 \leftarrow \mathsf{P}_{\mathsf{NIZK}}(\{\exists [s'_0]_t \mid R_2 = g^{[s'_0]_t} \wedge \hat{\mathcal{J}}_2 = (Q^{k_{s,t}})^{[s'_0]_t}\}, [s'_0]_t)$
		$(\mathsf{com}, \mathsf{decom}) \leftarrow \mathsf{P}_{\mathsf{COM}}(\vec{s}, \mathcal{J}_2, (\hat{\mathcal{J}}_2, R_2, \pi_2))$
	$\xleftarrow{\quad \mathsf{com} \quad}$	
$[s'_0]_s \leftarrow\$ \mathbb{Z}_q; R_1 \leftarrow g^{[s'_0]_s}$		
$\mathcal{J}_1 \leftarrow Q^{\mathsf{sk}_{s,b} \cdot k_{s,t}}; \hat{\mathcal{J}}_1 \leftarrow Q^{[s'_0]_s \cdot k_{s,t}}$		
$\pi_1 \leftarrow \mathsf{P}_{\mathsf{NIZK}}(\{\exists [s'_0]_s \mid R_1 = g^{[s'_0]_s} \wedge \hat{\mathcal{J}}_1 = (Q^{k_{s,t}})^{[s'_0]_s}\}, [s'_0]_s)$		
$\tau \leftarrow\$ \mathbb{Z}_q; c''_\alpha \leftarrow (c'_\alpha)^\tau$		
	$\xrightarrow{\quad \mathcal{J}_1, \hat{\mathcal{J}}_1, R_1, \pi_1, c''_\alpha \quad}$	
		If $\mathsf{V}_{\mathsf{NIZK}}(\pi_1, (R_1, \hat{\mathcal{J}}_1)) \neq 1$ then abort
		$\gamma \leftarrow \mathsf{Dec}_{\mathsf{HE}}(\mathsf{sk}_t, c''_\alpha); A'' \leftarrow g^\gamma; (A^*)'' \leftarrow Q^{\gamma \cdot k_{s,t}}$
		$\pi_A \leftarrow \mathsf{P}_{\mathsf{NIZK}}(\{\exists \gamma \mid A'' = g^\gamma \wedge (A^*)'' = (Q^{k_{s,t}})^\gamma\}, \gamma)$
		$\mathcal{J} \leftarrow \mathcal{J}_1 \cdot \mathcal{J}_2; R \leftarrow R_1 \cdot R_2 \cdot A''; \hat{\mathcal{J}} \leftarrow \hat{\mathcal{J}}_1 \cdot \hat{\mathcal{J}}_2 \cdot (A^*)''$
		$h_0 \leftarrow H_s(m \| R \| \hat{\mathcal{J}})$
		$\forall i \in \{1, \ldots, n-1\}:$
		$\quad L_i \leftarrow g^{s_i} \cdot \mathsf{pk}_{i,b}^{h_{i-1}}; R_i \leftarrow \mathsf{pk}_{i,1-b}^{s_i \cdot k_i} \cdot \mathcal{J}^{h_{i-1}}$
		$\quad h_i \leftarrow H_s(m \| L_i \| R_i)$
		$[s_0]_t \leftarrow [s'_0]_t - h_{n-1} \cdot \mathsf{sk}_{t,b}$
	$\xleftarrow{\quad (\mathsf{decom}, \vec{s}, \mathcal{J}_2, \hat{\mathcal{J}}_2, R_2, \pi_2), A'', (A^*)'', \pi_A, [s_0]_t \quad}$	
If $\mathsf{V}_{\mathsf{COM}}(\mathsf{com}, \mathsf{decom}, (\vec{s}, \mathcal{J}_2, \hat{\mathcal{J}}_2, R_2, \pi_2)) \neq 1$ then abort		
If $\mathsf{V}_{\mathsf{NIZK}}(\pi_2, (R_2, \hat{\mathcal{J}}_2)) \neq 1$		
$\vee\ \mathsf{V}_{\mathsf{NIZK}}(\pi_A, (A'', (A^*)'')) \neq 1$ then abort		
If $(A')^\tau \neq A''$ then abort		
$\mathcal{J} \leftarrow \mathcal{J}_1 \cdot \mathcal{J}_2; R \leftarrow R_1 \cdot R_2 \cdot A''; \hat{\mathcal{J}} \leftarrow \hat{\mathcal{J}}_1 \cdot \hat{\mathcal{J}}_2 \cdot (A^*)''$		
$h_0 \leftarrow H_s(m \| R \| \hat{\mathcal{J}})$		
$\forall i \in \{1, \ldots, n-1\}:$		
$\quad L_i \leftarrow g^{s_i} \cdot \mathsf{pk}_{i,b}^{h_{i-1}}; R_i \leftarrow \mathsf{pk}_{i,1-b}^{s_i \cdot k_i} \cdot \mathcal{J}^{h_{i-1}}$		
$\quad h_i \leftarrow H_s(m \| L_i \| R_i)$		
$[s_0]_s \leftarrow [s'_0]_s - h_{n-1} \cdot \mathsf{sk}_{s,b}$		
	$\xrightarrow{\quad [s_0]_s \quad}$	
		$\sigma := (s_0 := [s_0]_s + [s_0]_t + \gamma, s_1, \ldots, s_{n-1}, R, \hat{\mathcal{J}}, \mathcal{J}, b)$
		If verification of σ fails then abort
		Else publish signature σ
$\gamma \leftarrow s_0 - ([s_0]_s + [s_0]_t); \bar{\alpha} \leftarrow \gamma \cdot \tau^{-1}$		
Send $\bar{\alpha}$ to P_r		
return $\bar{\alpha}$		**return** $\top$

Figure 13: Puzzle solver protocol of DLSAG-based construction.

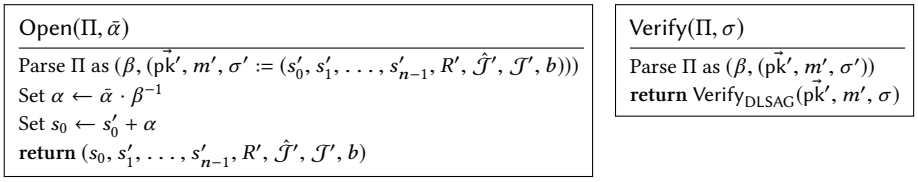

Figure 14: Open and verify algorithms of DLSAG-based construction.