LedgerLocks: A Security Framework for Blockchain Protocols Based on Adaptor Signatures

Erkan Tairi
TU Wien
Christian Doppler Laboratory
Blockchain Technologies for the Internet of Things
Vienna, Austria
erkan.tairi@tuwien.ac.at

Pedro Moreno-Sanchez[*]
IMDEA Software Institute
VISA Research
Madrid, Spain
pedro.moreno@imdea.org

Clara Schneidewind
MPI-SP
Bochum, Germany
clara.schneidewind@mpi-sp.org

ABSTRACT

The scalability and interoperability challenges in current cryptocurrencies have motivated the design of cryptographic protocols that enable efficient applications on top and across widely used cryptocurrencies such as Bitcoin or Ethereum. Examples of such protocols include (virtual) payment channels, atomic swaps, oracle-based contracts, deterministic wallets, and coin mixing services. Many of these protocols are built upon minimal core functionalities supported by a wide range of cryptocurrencies. Most prominently, adaptor signatures (AS) have emerged as a powerful tool for constructing blockchain protocols that are (mostly) agnostic to the specific logic of the underlying cryptocurrency. Even though AS-based protocols are built upon the same cryptographic principles, there exists no modular and faithful way to reason about their security. Instead, all the works analyzing such protocols focus on reproving how adaptor signatures are used to cryptographically link transactions while considering highly simplified blockchain models that do not capture security-relevant aspects of transaction execution in blockchain-based consensus.

To help this, we present LedgerLocks, a framework for the secure design of AS-based blockchain applications in the presence of a realistic blockchain. LedgerLocks defines the concept of AS-locked transactions, transactions whose publication is bound to the knowledge of a cryptographic secret. We argue that AS-locked transactions are the common building block of AS-based blockchain protocols and we define $\mathcal{G}_{\mathsf{LedgerLocks}}$, a realistic ledger model in the Universal Composability framework with built-in support for AS-locked transactions. As LedgerLocks abstracts from the cryptographic realization of AS-locked transactions, it allows protocol designers to focus on the blockchain-specific security considerations instead.

1 INTRODUCTION

Blockchain-based cryptocurrencies such as Bitcoin, enable mutually distrusting users to perform financial transactions without relying on a trusted third party. However, for their large-scale adoption, cryptocurrencies face major interoperability and scalability challenges. These challenges can be tackled with the help of cryptographic protocols that form a more flexible application layer on top of the core cryptocurrency functionalities. Prominent examples are atomic swaps [39] for users to trade their coins across different cryptocurrencies or payment channels [1] to perform an unlimited number of fast bilateral payments while publishing only a small number of transactions on the blockchain.

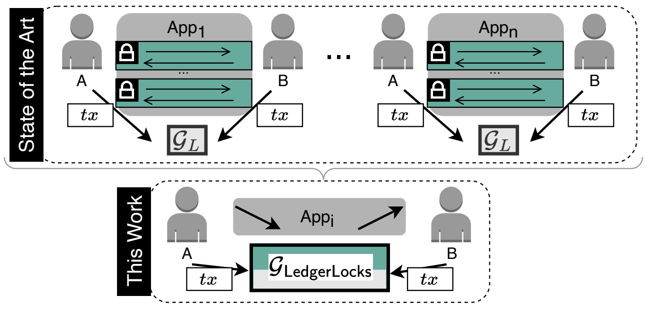


Figure 1: Overview of the LedgerLocks framework. Cryptographic protocols for creating AS-locked transactions in the state of the art (in green) are modeled by an ideal functionality $\mathcal{G}_{\mathsf{LedgerLocks}}$.

To ease the interoperability across cryptocurrencies, these protocols are usually realized upon simple core operations supported by most cryptocurrencies (e.g., payment authorization with a digital signature from the sender). This endeavor has been facilitated by the recent discovery of *adaptor signatures (AS)* [1, 18], which allow for conditioning the creation of a digital signature on the knowledge of a cryptographic secret.

Despite the multitude of cryptographic blockchain protocols relying on adaptor signatures [2, 5, 11, 19, 22, 29–32, 39], the security analysis of these protocols is usually incomplete. This is due to the fact that the security of these protocols does not only rely on the correct usage of cryptographic primitives used in the message exchanges between protocol participants but also on the guarantees that stem from the underlying blockchain consensus. In spite of that, all current works proposing new AS-based blockchain protocols study their security in the context of highly simplified ledger models, defined in an ad-hoc manner [1, 3, 4, 22, 31, 36, 39].

However, the subtleties of the ledger model have a significant influence on the blockchain protocol security and neglecting these aspects can easily result in undetected security issues as we will show in §2. Consequently, it is highly desirable to build an infrastructure that facilitates the reasoning about AS-based blockchain protocols in the presence of a realistic ledger.

Towards this goal, we propose LedgerLocks, a framework for separating the reasoning about ledger-specific aspects of AS-based

[*]This work has been done while in employment of IMDEA Software Institute.

blockchain protocols from the cryptographic operations. We observe that adaptor signatures are used in these protocols to encode a generic building block that we call *AS-locked transactions*. AS-locked transactions are transactions whose publication on the blockchain is bound to the knowledge of a cryptographic secret in two ways: (i) knowing the cryptographic secret is a prerequisite for a party holding the AS-locked transaction to publish it on the ledger, and (ii) the publication of the AS-locked transaction on the ledger reveals the secret to all parties holding the AS-locked transaction. By synthesizing this building block and integrating it into a realistic ledger functionality, we can describe many blockchain protocols in terms of this functionality without the cryptographic interactions between the protocol participants. We illustrate this approach in Figure 1: In the state of the art, AS-based blockchain protocols are mainly given through the exchange of cryptographic messages among the participants. These interactions shall ensure that the protocol participants can construct valid transactions to be published on the blockchain (given through a simplified ledger functionality $\mathcal{G}_L$) in compliance with the protocol goals. The cryptographic reasoning for showing the security of these interactions is essentially the same throughout the state-of-the-art protocols (denoted by the green parts of protocols App_1 to App_n). In LedgerLocks, we define a realistic ledger functionality $\mathcal{G}_{\mathsf{LedgerLocks}}$ that supports generic AS-locked transactions and, thus, subsumes the cryptographic aspects of these protocols. In this way, AS-based blockchain protocols can be described in terms of AS-locked transactions without further need for cryptographic interactions between the protocol participants. Consequently, the subsequent security analysis of such protocols does not require cryptographic reasoning but can focus on the ledger-specific security arguments. Lifting the burden of concurrently reasoning about both cryptographic and ledger-specific security aspects paves the ground for the security analysis of blockchain protocols in realistic ledger models.

Constructing $\mathcal{G}_{\mathsf{LedgerLocks}}$ comes with multiple technical challenges: The logic of protocols using AS-locked transactions usually relies on relating these transactions through the structure of their cryptographic conditions. Therefore, for truly modular reasoning we need a general model of cryptographic conditions that integrates with $\mathcal{G}_{\mathsf{LedgerLocks}}$ and is adaptable to the protocol needs. Further, to show that $\mathcal{G}_{\mathsf{LedgerLocks}}$ is realizable by adaptor signatures in the presence of such a model of cryptographic conditions, a novel composable notion of adaptor signature security is needed. Finally, to facilitate flexible reasoning in a faithful ledger model, we need to model $\mathcal{G}_{\mathsf{LedgerLocks}}$ to expose provably realistic ledger behavior while supporting a generic notion of AS-locked transactions.

In this work, we overcome these challenges as follows:

- We model cryptographic conditions as a standalone (global) ideal functionality $\mathcal{G}_{\mathsf{Cond}}$, which encodes operations over conditions, such as their composition. $\mathcal{G}_{\mathsf{Cond}}$ can be easily extended to account for other operations in a modular fashion, that is, without modifying the several other functionalities using it in a shared manner to keep conditions consistent across them (§5).
- We model (two-party) adaptor signatures as an ideal functionality $\mathcal{F}_{\mathsf{AdaptSig}}$ and prove that it is UC-realized by any two-party adaptor signature with aggregatable public keys generated from an identification scheme [18], a class encompassing all the digital signatures used in current AS-based applications (§6).
- Based on the ledger functionality $\mathcal{G}_{\mathsf{Ledger}}$ from Badertscher et al. [8], which has been proven to be realizable by the Bitcoin backbone protocol [8] as well as the proof of stake-based protocol Ouroboros Genesis [7], we propose $\mathcal{G}_{\mathsf{LedgerLocks}}$, an ideal functionality that models a ledger with generic AS-locked transactions. We provide a protocol $\Pi_{\mathsf{LedgerLocks}}$ that UC-realizes $\mathcal{G}_{\mathsf{LedgerLocks}}$ in the presence of $\mathcal{G}_{\mathsf{Ledger}}$ from [8] and $\mathcal{F}_{\mathsf{AdaptSig}}$ (§7).
- We demonstrate the flexibility of our framework, by using it to describe an enhanced atomic swap protocol $\Pi_{\mathsf{AtomicSwap}}$ and a multi-hop payment protocol Π_{MultiHop} over payment channels Π_{Channel}, all of them protocols relying on AS-locked transactions. To this end, we instantiate $\mathcal{G}_{\mathsf{LedgerLocks}}$ with support for transaction timelocks, which are crucial for atomic swap and multi-hop payment security. The description of $\Pi_{\mathsf{AtomicSwap}}$, Π_{Channel} and Π_{MultiHop} does not involve additional cryptography, and hence, can focus on the delicate task of adjusting the protocol timelocks to provide security in the presence of realistic blockchains as modeled by $\mathcal{G}_{\mathsf{LedgerLocks}}$ (§8).

2 STATE-OF-THE-ART BLOCKCHAIN PROTOCOL ANALYSIS

In this section, we overview the existing approaches to analyzing the security of (AS-based) blockchain protocols with an emphasis on the ledger modeling. For this, we first give background on the workings of realistic ledgers and then illustrate the impact of the ledger model on the security analysis using the examples of an atomic swap and a multi-hop off-chain payment protocol. Finally, we discuss the ledger models used in literature and their limitations.

Blockchain workings. In cryptocurrencies built upon a tamper-resistant distributed ledger (the blockchain), network participants conduct transactions by broadcasting them to the network. Specific network nodes, so-called *miners*, group valid transactions into blocks and append them to the blockchain. To ensure fairness, the miner selection process is randomized based on a resource in the possession of miners (usually their computational power or financial stakes). With a majority of the resource being owned by honest miners, it is guaranteed that the system will progress safely: Eventually, the system will reach consensus on a stable prefix of the blockchain and valid transactions are guaranteed to be eventually included in such a stable prefix.

The resulting transaction execution model comes with several peculiarities: Transactions submitted by honest users are not necessarily guaranteed to be included in the blockchain but could still be outrun by (adversarial) transactions that invalidate them, e.g., by consuming the same assets. Also, transactions are already public before their inclusion in the blockchain, possibly leaking sensitive information to a (miner-controlling) attacker.

Atomic swaps. An atomic swap (Figure 2) involves two ledgers $\mathbb{A}$ (blue), $\mathbb{B}$ (orange) and two users Alice (A) and Bob (B), holding assets in $\mathbb{A}$ and $\mathbb{B}$, respectively. A correct atomic swap protocol ensures that Alice receives Bob's assets on $\mathbb{B}$ and Bob receives Alice's assets on $\mathbb{A}$ if both parties are honest. An atomic swap protocol is considered secure if an honest party always either (i) receives the other party's assets; or (ii) keeps their own assets. To set up such an atomic swap, Alice locally creates a cryptographic secret x. Moreover, Alice and Bob deposit their assets into a shared account $\textcircled{AB}$ in the respective chain (through deposit transaction $\boxed{dtx_A}$ and $\boxed{dtx_B}$). The assets in a shared account are set up such that they can be released by the intended receiver showing the secret value x or refunded to the original owner. This functionality is achieved by Alice and Bob jointly creating AS-locked transactions $\boxed{ctx_A}$ and $\boxed{ctx_B}$, which can only be submitted upon the knowledge of secret x and whose publication will release x to the other party. Further, they create the refund transactions $\boxed{rtx_A}$ and $\boxed{rtx_B}$ that allow Alice and Bob to retrieve back their assets in case the other party stops collaborating.

After a successful setup, Alice, who knows the secret x, can claim Bob's assets in $\mathbb{B}$ (using $\boxed{ctx_A}$). Then, Bob can read x from $\mathbb{B}$ and use it to claim the assets in the shared account on $\mathbb{A}$ (using $\boxed{ctx_B}$). Alternatively (e.g., if the other user fails to cooperate), Alice and Bob can refund their assets by publishing $\boxed{rtx_A}$ or $\boxed{rtx_B}$, respectively. Alice's refund transaction $\boxed{rtx_A}$ is equipped with a *timelock* that ensures that it can only be published after time t. This restriction prevents Alice from simultaneously publishing $\boxed{rtx_A}$ and $\boxed{ctx_A}$ to retrieve the assets on both chains. Instead, if Alice has not claimed Bob's assets (through $\boxed{ctx_A}$) until time right before t, Bob can publish $\boxed{rtx_B}$ to be refunded before $\boxed{rtx_A}$ becomes valid at time t.

Ledger model and atomic swap security. Although the idea behind the atomic swap protocol seems simple, it is only secure when assuming a highly simplified ledger model. More precisely, its security relies on the assumption that $\boxed{rtx_B}$ will be included immediately after Bob sent it to the network. In practice, the ledger only guarantees, that $\boxed{rtx_B}$ will be included in the blockchain within a time delay Δ. During this time, other transactions (even if submitted later) may be included in the blockchain, invalidate $\boxed{rtx_B}$ and, thus, prevent its inclusion. Specifically, a malicious Alice could send $\boxed{ctx_B}$ right after observing $\boxed{rtx_B}$ on the network. As a consequence, $\boxed{ctx_B}$ could be included in the blockchain first, canceling $\boxed{rtx_B}$.

To secure the atomic swap protocol in the presence of such ledgers, Bob's behavior in the protocol needs to be adapted as illustrated in Figure 3 (top): Right before time $t - 2\Delta$, Bob needs to initiate the refund of their assets by publishing $\boxed{rtx_B}$ (denoted by a dotted arrow). Like this, Bob knows right before time $t - \Delta$ whether the refund was successful or whether Alice managed to outrun Bob with $\boxed{ctx_A}$. In the latter case, Bob learns x and publishes $\boxed{ctx_B}$, which is guaranteed to be included before t, the time starting from which Alice could publish $\boxed{rtx_A}$.

Interestingly, even the adapted protocol is insecure when considering another subtlety of realistic ledger workings: Once submitted to the network, a transaction becomes public to the miners, even before being included in the blockchain. Considering that and despite her advantage of exclusively knowing secret x, Alice is subject to an attack (Figure 3, bottom). When Alice claims Bob's assets (by publishing $\boxed{ctx_A}$), a malicious Bob can learn x and still outrun $\boxed{ctx_A}$ with $\boxed{rtx_B}$. Then, Bob could claim Alice's assets (publishing $\boxed{ctx_B}$) before Alice could refund her assets using $\boxed{rtx_A}$ at time t.

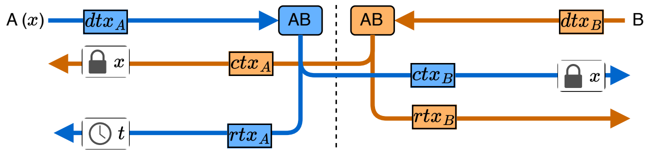
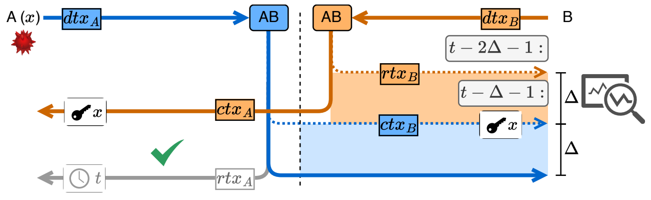


Figure 2: Transactions in an atomic swap protocol.



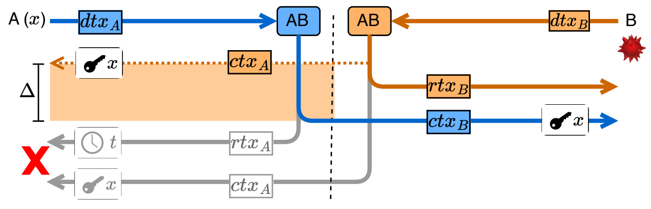


Figure 3: Atomic swap in a ledger with delayed inclusion (top). Attack in an atomic swap in a realistic ledger (bottom).

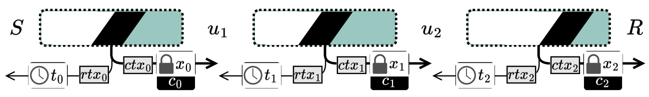


Figure 4: Setup for a multi-hop payment from sender S to receiver R. Dotted boxes denote payment channels between users, with the funds of the left participant in white, funds of the right participant in green, and locked funds in black. The distribution of locked funds is controlled using transactions $\mathtt{ctx}_i$ and $\mathtt{rtx}_i$.

This issue can be mitigated when introducing an additional timelock for Bob's refund transaction $\boxed{rtx_B}$ but it would stay undetected when relying on a ledger model that does not leak transactions to the attacker upon their submission to the network.

Multi-hop payment security. The described issues do not only apply to atomic swaps but extend to a wide range of (AS-based) blockchain protocols. One example is off-chain payments in payment channel networks such as described in [31]. Payment channel networks, like Bitcoin's Lightning Network [33], form a layer for fast peer-to-peer payments on top of cryptocurrencies by relying on two-party payment channels. In a *payment channel*, users lock funds in a shared account and exchange guarantees for the ownership distribution (channel state) of these funds. In this way, collaborative channel users can perform *offchain payments* by updating the channel state without making transactions on the blockchain. The channel parties can always close the channel (e.g. when the other party stops collaborating) by publishing transactions on the blockchain to obtain the funds according to the latest channel state.

Users that do not share a payment channel can still securely exchange offchain payments as long as they are connected via a path in a *payment channel network* as illustrated in Figure 4. For preparing a payment, the users u_i on a payment path between sender S (= u_0) and receiver R (= u_n) lock channel funds for the payment such that the payment later can be enforced atomically. To this end, the users u_i ($0 \leq i < n$) prepare *conditional offchain* *payments* based on some condition c_i to their successors u_{i+1} on the payment path. These conditional offchain payments are realized through AS-locked transactions $\boxed{\mathtt{ctx}_i}$ on the channel funds that can be published once the channel is closed. The conditions c_i are set up such that if user u_{i+1} claims $\boxed{\mathtt{ctx}_i}$, this will reveal a secret x_i to u_i allowing them to satisfy c_{i-1} and claim $\boxed{\mathtt{ctx}_{i-1}}$ in turn. Once all conditional payments are set up, S initiates the payment by revealing a secret s_R to R that allows them to open c_{n-1}. Next, the payment gets propagated through the payment path: Collaborative users u_i and u_{i+1} can update their payment channels offchain after revealing the secret that would enable $\boxed{\mathtt{ctx}_i}$. If u_i is not collaborating, u_{i+1} can close the channel and use $\boxed{\mathtt{ctx}_i}$ to enforce the conditional payment based on the last channel state.

To ensure that a malicious sender cannot indefinitely lock the funds of intermediaries on the path, the users, in addition to $\boxed{\mathtt{ctx}_i}$ prepare a refund transaction $\boxed{\mathtt{rtx}_i}$ that allows u_i to retrieve back their funds after time t_i. As for the atomic swap protocol, such a refund option introduces additional challenges in the design of a secure protocol: If user u_{i+1} is not responding, honest u_i needs to close the channel and invoke the refund using $\boxed{\mathtt{rtx}_i}$. However, even after successful channel closure while waiting for $\boxed{\mathtt{rtx}_i}$ to be included in the blockchain, u_{i+1} may still decide to publish $\boxed{\mathtt{ctx}_i}$ instead. In this case u_i needs to observe $\boxed{\mathtt{ctx}_i}$ on the blockchain, learn x_i and use it to continue the payment (either offchain or onchain). For this, it needs to be ensured that $\boxed{\mathtt{ctx}_{i-1}}$ is still valid at this point and so that $\boxed{\mathtt{rtx}_{i-1}}$ has not been published yet.

This illustrates how the protocol design is closely intertwined with the precise guarantees that the underlying ledger provides: (i) The protocol transactions need to have timelocks that respect the ledger inclusion times. In particular, timelock t_i of $\boxed{\mathtt{rtx}_i}$ needs to be adjusted such that $t_i < t_{i-1} + 2\Delta + \Delta_{close}$ (for t_{i-1} being the timelock of $\boxed{\mathtt{rtx}_{i-1}}$ and Δ_{close} being the channel closing time) to ensure that u_i after closing their outgoing channel and publishing $\boxed{\mathtt{rtx}_i}$ at t_i, when learning (latest) at $t_i + \Delta$ whether $\boxed{\mathtt{rtx}_i}$ or $\boxed{\mathtt{ctx}_i}$ got included in the blockchain there is still enough time to close their ingoing channel and publish $\boxed{\mathtt{ctx}_{i-1}}$ on the blockchain (which may take up to $\Delta_{close} + \Delta$). (ii) The honest participant strategies need to respect the timing constraints. It is e.g., crucial that honest participants frequently poll the blockchain for the inclusion of payment channel closing transactions and to react on the publication of a claim transaction $\boxed{\mathtt{ctx}}$ onchain in well-defined time windows to obtain the desired correctness guarantees.

If the ledger model is not accounting for the exact ledger behavior concerning the attacker's delay and learning capabilities, wrong protocols can easily be proven secure. E.g., consider a version of the multi-hop payment protocol where receiver R accepts S's payment too late: After setting up the payment, if R receives s_R only after $t_n + \Delta_{close}$, R is not guaranteed anymore to receive the payment: If u_{n-1} does not collaborate in updating the channel, R needs to close the channel with u_{n-1} (taking up to Δ_{close}) and then publish $\boxed{\mathtt{ctx}_n}$ at time $t < t_n$. At t_n, however, a malicious u_{n-1} already submitted $\boxed{\mathtt{rtx}_n}$ to outrun $\boxed{\mathtt{ctx}_n}$ resulting in u_{n-1} being refunded while still learning s_R. With the knowledge of s_R, u_{n-1} completes the payment and receives the funds meant for R. Similar to the atomic swap example, such an attack could not be detected in the presence of a ledger model with instant transaction inclusion or without modeling that the attacker may learn transaction details (such as x_i) before the transaction's inclusion in the blockchain.

Ledger Features	$\mathcal{L}_{inst}$	$\mathcal{L}_{\Delta}$	$\mathcal{G}_{\text{Ledger}}$ / $\mathcal{G}_{\text{LedgerLocks}}$
Attacker knowledge	X	X*	✓
Attacker capabilities	X	X*	✓
Inclusion time guarantees	X	✓	✓
Realizability	X	X†	✓

Table 1: Overview of features of ledger models used for the analysis of blockchain protocols. X* denotes that the corresponding ledger feature is underspecified, X† indicates that the realizability of $\mathcal{L}_{\Delta}$ is unknown.

Ledger models in the state of the art. As highlighted by the examples in the last paragraph, there are several realistic ledger features whose modeling comes with immediate security implications: Foremost, this is a realistic attacker model that accounts for both the *attacker knowledge* (e.g., the knowledge of transactions after they got submitted but before they got included in the blockchain) and the *attacker capabilities* (e.g., to influence the order and time of transaction inclusion). Related to the attacker capabilities, the concrete *inclusion time guarantees* for honest users are crucial for secure protocol design (e.g., for the correct adjustment of timeouts).

The importance of a realistic ledger model for the security analysis of blockchain protocols is also emphasized by [26] who propose a formal security analysis of Bitcoin's Lightning Network in the presence of the ledger model $\mathcal{G}_{\text{Ledger}}$ from [8]. The authors showcase that for precisely specifying the Lightning Network protocol, it is inevitable to rely on the exact timing guarantees obtained from the ledger in [8]. Further, they prove that the simplified models that are used in the security analysis of [14–17, 39] do not only not fail to reflect the guarantees of realistic ledgers but that it is impossible to design a ledger that could provide such guarantees.

However, as summarized in Table 1, the state-of-the-art still analyses blockchain protocols in the presence of simplified ledger models, which disregard security-relevant ledger features. These works consider either (i) (provable unrealizable) ledgers $\mathcal{L}_{inst}$ with immediate inclusion guarantees [14–17, 39]; or (ii) ledgers $\mathcal{L}_{\Delta}$ that let the attacker delay the inclusion of a transaction up to delay Δ [1, 2, 4, 5, 22, 34, 36, 38, 40]. Even in protocols from the second category, the exact way that the attacker can exercise their power to delay transactions stays vague. E.g., in [1], it is stated that upon a message being posted by the user, the ledger should "wait until round $\tau_1 \leq \tau_0 + \Delta$ (the exact value of τ_1 is determined by the adversary)". This description leaves open at which point in time and based on which information the adversary determines the inclusion time τ_1. However, as shown for the example of atomic swap and multi-hop payments, leaving these aspects underspecified may result in the security analysis missing relevant attacks.

This work aims to simplify the design of AS-based blockchain protocols in the presence of realistic ledgers. To this end, the proposed LedgerLocks framework extends the realistic ledger model $\mathcal{G}_{\text{Ledger}}$ from [8] to include an abstraction for the cryptographic operations required to synchronize transactions in AS-based protocols. Like this, the security analysis of these protocols can focus on the ledger-specific aspects instead of cryptographic arguments.

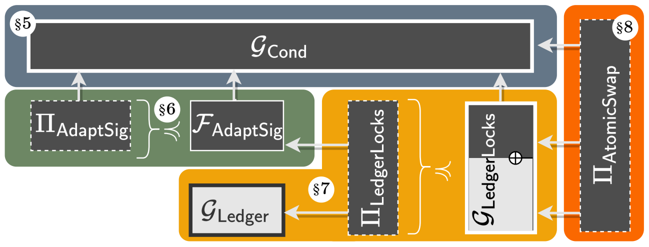

§5
$\mathcal{G}_{\text{Cond}}$
Π_{AdaptSig}
§6
$\mathcal{F}_{\text{AdaptSig}}$
$\mathcal{G}_{\text{Ledger}}$
§7
$\Pi_{\text{LedgerLocks}}$
$\mathcal{G}_{\text{LedgerLocks}}$
§8
$\Pi_{\text{AtomicSwap}}$

Figure 5: Overview of the infrastructure of LedgerLocks. Dark-gray components indicate novel functionalities and protocols introduced in this work. The $\preccurlyeq$ relation denotes that a protocol Π (left) realizes a functionality $\mathcal{F}/\mathcal{G}$ (right) in the UC framework.

3 TECHNICAL OVERVIEW

In this section, we overview the LedgerLocks framework. For enabling modular reasoning about the security of blockchain protocols, LedgerLocks relies on the Universal Composability (UC) framework of Canetti [12]. In the UC framework, the security of protocols is defined in terms of *ideal functionalities*, which describe the idealized secure protocol behavior. Slightly simplified, a protocol is considered secure (w.r.t. an ideal functionality) if an adversarial environment cannot distinguish whether it interacts with the protocol or with the ideal functionality. This security notion is sufficiently strong to enable modular security reasoning. More precisely, once protocol Π is proven secure w.r.t. an ideal functionality $\mathcal{F}$, the security of protocols using Π as a subroutine can be analyzed assuming $\mathcal{F}$ as subroutine instead.

The LedgerLocks framework provides ideal functionalities to characterize the security of cryptographic conditions ($\mathcal{G}_{\text{Cond}}$), adaptor signatures ($\mathcal{F}_{\text{AdaptSig}}$), and lock-enabling ledgers ($\mathcal{G}_{\text{LedgerLocks}}$). Figure 5 depicts how these ideal functionalities connect to finally expose an interface for modularly defining AS-based blockchain protocols (such as an atomic swap protocol $\Pi_{\text{AtomicSwap}}$) based on lock-enabling ledgers and cryptographic conditions.

In the following, we describe in more detail the different components of the LedgerLocks framework (as highlighted by the different background colors in Figure 5). To this end, Figure 6 gives a more granular account of the individual components from Figure 5.

Conditions. We first define the ideal functionality $\mathcal{G}_{\text{Cond}}$ for representing (secure) cryptographic conditions (§5). Intuitively, a cryptographic condition describes the properties as given by a hard relation R. Concretely, a condition is identified by a public statement and we say that the condition is satisfied if the corresponding witness is provided. Due to the hardness of the relation, without prior knowledge, it is hard to come up with a witness satisfying a given condition. A typical example is the discrete logarithm (DLOG) assumption over certain cyclic groups $(\mathbb{G}, g, q)$ (with generator g and order q), where given a group element (the statement) $Y = g^y$, it is hard to compute the exponent y (the witness).

At first sight, it may seem counter-intuitive to define conditions as a standalone ideal functionality. The reason for doing so is that the prerequisites for a party to craft a witness related to a statement often emerge from a cryptographic protocol for the condition creation. As an example, consider the following scenario. Alice plays a simple guessing game with Bob. If Bob can guess a number between 1 and 10 then he gets a prize from Alice. To implement this based on DLOG, Alice prepares ten secret witnesses $(y_i^{mask})_{i\in(1,10)}$ and sends the corresponding statements $\overrightarrow{Y^{mask}} = (g^{y_i^{mask}})_{i\in(1,10)}$ to Bob. Bob himself prepares one witness y^{win} and ten witnesses $(y_i^{blank})_{i\in(1,10)}$. For the guess j, Bob prepares $\overrightarrow{Y^{guess}} = (Y_i^{guess})_{i\in(1,10)}$ such that for $i \neq j$, $Y_i^{guess} = (Y_i^{mask})^{y_i^{blank}}$ and $Y_j^{guess} = g^{y^{win}}$. At this point, Bob knows the witness for Y_j^{guess}, while he cannot know the witness for any statement Y_i^{guess} with $i \neq j$, since for this, Bob would require Alice's secret masking values $(y_i^{mask})_{i\in(1,10)}$. Bob proves in zero-knowledge to Alice that $\overrightarrow{Y^{guess}}$ is well-formed. Now, Alice chooses a number m and prepares a payment to Bob based on Y_m^{guess}. Alice at this point, cannot know which condition Bob can open, only that Bob can open exactly one out of the ten provided conditions. If Alice and Bob chose the same number ($j = m$), Bob can complete the payment, otherwise, the money stays with Alice.

For reasoning about the above-described guessing game, we need to capture that there are ten conditions out of which Bob can open exactly one. However, the creation of conditions with this property involves a protocol itself. This condition-creation protocol is independent of more advanced protocols relying on conditions with the respective property. In summary, by modeling conditions as a separate functionality, we can modularize reasoning about condition creation (e.g., $\overrightarrow{Y^{guess}}$) and protocols using these conditions (e.g., the payment from Alice to Bob based on Y_m^{guess}).

Technically, this means that we can extend the functionality $\mathcal{G}_{\text{Cond}}$ with further types of conditions without reproving any of the results in Sections 6 and 8. For the scope of this work, we present three different forms of conditions: 1) Plain conditions (which we call *individual conditions*), which users can create on their own by creating a fresh witness and the corresponding statement for a given hard relation. 2) Composed conditions, which combine two existing conditions. The concrete composition operation depends on the underlying hard relation and is specified as a parameter f_{merge} to the functionality $\mathcal{G}_{\text{Cond}}$. 3) 1-out-of-n conditions, which enable a party P together with a set of users U to jointly create a vector of statements, such that P can (without the collaboration of all users in U) only know the witness for exactly one out of these statements. The described guessing game included the creation of a 1-out-of-10 condition for DLOG with $P = Bob$ and $U = \{Alice\}$.

Note that we fix the hard relation R (as well as f_{merge}) as a parameter to $\mathcal{G}_{\text{Cond}}$. This is needed to enable composability with the adaptor signature functionality $\mathcal{F}_{\text{AdaptSig}}$, which is also defined w.r.t. to R. In §5, we show how to provably realize $\mathcal{G}_{\text{Cond}}$ for the DLOG relation with the protocol Π_{Cond}.

Within our LedgerLocks framework, we treat $\mathcal{G}_{\text{Cond}}$ as a *global subroutine* [6]. In the UC framework, a global subroutine $\mathcal{G}$ is an ideal functionality that can be securely used by some protocol Π, even if Π relies on another functionality $\mathcal{F}$, which makes use of $\mathcal{G}$ as a subroutine. One can think of $\mathcal{G}$ as constituting some safely shared state between Π and $\mathcal{F}$. An example for this usage of $\mathcal{G}_{\text{Cond}}$ is shown in Figure 6: Here, the protocol $\Pi_{\text{AtomicSwap}}$ uses the functionalities $\mathcal{G}_{\text{Cond}}$ (for creating conditions) and $\mathcal{G}_{\text{LedgerLocks}}$ again uses $\mathcal{G}_{\text{Cond}}$ as a subroutine (for checking conditions). Having $\mathcal{G}_{\text{Cond}}$

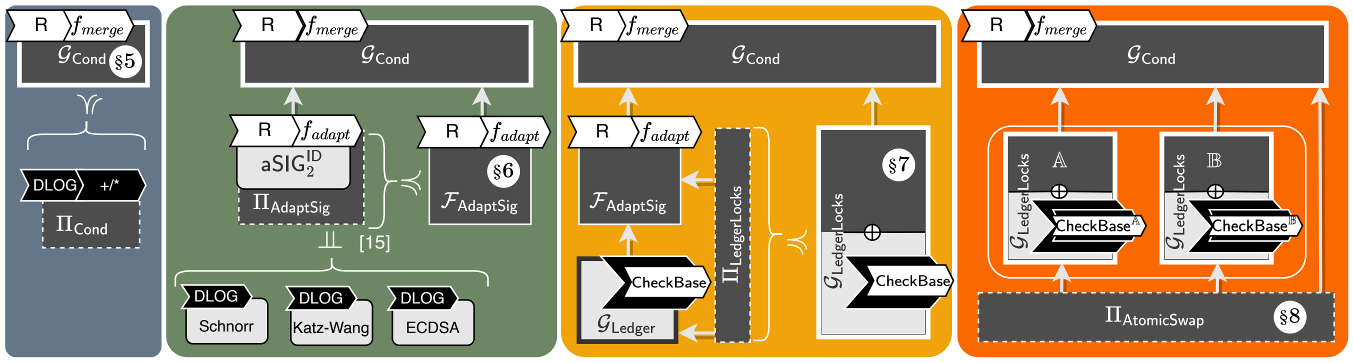


Figure 6: Detailed overview of the different components of the LedgerLocks infrastructure. White arrow-shaped boxes indicate parametrization; black ones instantiation.

as a global subroutine, intuitively allows us to define higher-level functionalities relative to a shared, stateful notion of conditions.

Adaptor signatures. In §6, we define $\mathcal{F}_{\text{AdaptSig}}$, an ideal functionality for adaptor signatures. Following the two-party adaptor signatures scheme defined by Erwig et al. [18], our ideal functionality $\mathcal{F}_{\text{AdaptSig}}$ allows two parties to jointly create a verification key, sign a message and pre-sign a message with respect to a given condition. Moreover, each user on its own can adapt a pre-signature into a valid signature if they can satisfy the corresponding condition. Finally, they can also extract the witness y for a condition Y if, for any given message m, they can provide a valid pre-signature $\hat{\sigma}$ on m with condition Y and the corresponding full signature σ obtained through adaptation with y.

In contrast to the adaptor signature scheme whose security is characterized in terms of game-based security definitions, modeling adaptor signatures as an ideal functionality comes with the benefits of composable reasoning in the UC framework. In particular, it enables modular reasoning with respect to conditions, since our model of $\mathcal{F}_{\text{AdaptSig}}$ relies on $\mathcal{G}_{\text{Cond}}$ to handle conditions. This means that for signature adaption $\mathcal{F}_{\text{AdaptSig}}$ queries $\mathcal{G}_{\text{Cond}}$ for determining whether the correct witness to open a condition was provided. Similar to $\mathcal{G}_{\text{Cond}}$, $\mathcal{F}_{\text{AdaptSig}}$ is parameterized by a hard relation R, and additionally a function f_{adapt}, which transforms a pre-signature $\hat{\sigma}$ and a witness y into a corresponding full signature σ. This parametrization enables $\mathcal{F}_{\text{AdaptSig}}$ to use conditions as provided by $\mathcal{G}_{\text{Cond}}$ in a truly modular fashion. Without fixing f_{adapt}, it would be required to make assumptions on the way that protocols relying on $\mathcal{F}_{\text{AdaptSig}}$ use $\mathcal{G}_{\text{Cond}}$ for condition generation. We include a more detailed discussion of this aspect in §6.

In practice, the parametrization by f_{adapt} does not pose a restriction. Indeed, we can show that $\mathcal{F}_{\text{AdaptSig}}$ is realizable for a big class of adaptor signatures (and corresponding adaptation functions). Erwig et al. [18] showed a generic transformation from signature schemes built from an identification scheme to two-party signature schemes (with aggregatable public keys), and then from there to two-party adaptor signature schemes. This transformation requires an adaptation function f_{adapt} with certain generic properties. We can show that all adaptor signature instances resulting from the transformation realize $\mathcal{F}_{\text{AdaptSig}}$ for the same f_{adapt} function as used in the transformation. More precisely, we give a generic wrapper protocol Π_{AdaptSig} around the algorithmic interface of two-party adaptor signature schemes (as defined in [18]) and prove this protocol to realize $\mathcal{F}_{\text{AdaptSig}}$ by reduction to the game-based security properties for adaptor signatures. Since (virtually all) the concrete adaptor signature constructions proposed so far are identification scheme-based signature schemes, our proof shows all of these schemes to realize $\mathcal{F}_{\text{AdaptSig}}$.

Lock-enabling ledger. In §7, we define $\mathcal{G}_{\text{LedgerLocks}}$, an ideal functionality for a distributed ledger with AS-locked transactions. In the design of $\mathcal{G}_{\text{LedgerLocks}}$, we follow the technique in [8]: The authors in [8] provide $\mathcal{G}_{\text{Ledger}}$, an ideal functionality modeling the subtleties of real-world blockchain consensus, in particular, realistic guarantees about the inclusion of transactions into the ledger. Moreover, they give Π_{Ledger}, a description of the Bitcoin backbone protocol and prove that it UC-realizes $\mathcal{G}_{\text{Ledger}}$.

$\mathcal{G}_{\text{Ledger}}$ and Π_{Ledger} are generic in that they do not fix the concrete transaction format or ledger logic. Instead, both of them are parametrized with a predicate isValidTx, which based on the internal ledger state determines whether a transaction is valid.

In this manner, the UC-realization proof holds for any instantiation of this predicate. Moreover, one can leverage the results in [8] by extending $\mathcal{G}_{\text{Ledger}}$ in two ways: (i) instantiating isValidTx predicate to account for the specific transaction formats and ledger logics; and (ii) extending the API of $\mathcal{G}_{\text{Ledger}}$ to account for further ledger features. Our ideal functionality $\mathcal{G}_{\text{LedgerLocks}}$ follows this blueprint to model multi-party account-based transaction authorization.

In more detail, $\mathcal{G}_{\text{LedgerLocks}}$ allows multiple parties to create a joint account. Transactions are associated with the set of all accounts, which need to provide authorization for transaction publication on the ledger. In addition to full authorization, accounts can lock a transaction on a condition, in which case any account owner can complete the authorization by providing an adequate witness. If such a AS-locked transaction is published on the ledger, honest account owners learn the corresponding witness, while a malicious owner learns the witness already upon transaction submission.

We build $\mathcal{G}_{\text{LedgerLocks}}$ from $\mathcal{G}_{\text{Ledger}}$ by 1) requiring the transaction format to include the list of accounts to authorize the transaction; 2) adding additional state and interfaces for the new operations;

and 3) instantiating the valid predicate to check for correct transaction authorization. The operation for releasing a AS-locked transaction thereby makes use of $\mathcal{G}_{\mathsf{Cond}}$ to determine whether a provided witness satisfies the condition of the corresponding transaction. To stay general, we do not fully fix the transaction format and the isValidTx predicate but introduce another predicate CheckBase, which performs additional transaction validity checks. In this way, we can modularly add further functionality to $\mathcal{G}_{\mathsf{LedgerLocks}}$, e.g., support for timelocks as we will show in §8.

Finally, we show how to realize $\mathcal{G}_{\mathsf{LedgerLocks}}$ with a protocol $\Pi_{\mathsf{LedgerLocks}}$, which uses $\mathcal{G}_{\mathsf{Ledger}}$ and $\mathcal{F}_{\mathsf{AdaptSig}}$. Thanks to our modeling of $\mathcal{G}_{\mathsf{Cond}}$ as global ideal functionality, the whole construction and proof are independent of the concrete realization of conditions.

Using the framework. We show how to use our LedgerLocks framework using an oracle-based atomic swap protocol that relies on AS-locked transactions as a case study. An oracle-based atomic swap works as the swap protocol described in §2 with the only difference that a third party (the oracle) needs to agree to the swap being executed. To this end, the condition locking the claim transactions is composed of a condition Y chosen by Alice and a condition Y_O chosen by the oracle so that the oracle needs to communicate its witness for Y_O to Alice for enabling the swap.

To express this protocol (denoted by $\Pi_{\mathsf{AtomicSwap}}$), we (partially) instantiate the CheckBase predicate of $\mathcal{G}_{\mathsf{LedgerLocks}}$ to encode a timelock check. Since $\mathcal{G}_{\mathsf{LedgerLocks}}$ is an extension of $\mathcal{G}_{\mathsf{Ledger}}$, we inherit its guarantees concerning the transaction inclusion time. Based on these guarantees, we can create timelocks that ensure that (1) Alice can successfully claim Bob's assets before Bob can refund and (2) Bob has always enough time to claim Alice's assets if Alice has claimed Bob's assets before, avoiding the attacks from §2. Note that $\Pi_{\mathsf{AtomicSwap}}$ does not only rely on $\mathcal{G}_{\mathsf{LedgerLocks}}$ but also on $\mathcal{G}_{\mathsf{Cond}}$ for the condition creation. The swap conditions are created by composing two conditions Y and Y_O. Since LedgerLocks is fully modular with respect to $\mathcal{G}_{\mathsf{Cond}}$, it is not even necessary to fix the protocol to create Y_O.

This example shows the flexibility of the LedgerLocks framework to use conditions in a way parametric to the other functionalities. To show this beyond the atomic swaps, we use LedgerLocks to express a multi-hop payment protocol over payment channels in Appendix G, Figures 25 to 28 and 30 to 33. Furthermore, these examples demonstrate how the ledger functionality can be easily extended to account for new ledger features, avoiding repetitive proofs for the ledger core functionality.

Privacy. Adaptor signatures come with privacy advantages that cannot easily be captured within UC-based security definitions. For completeness, we add a discussion on privacy in §6 and give (game-based) definitions for adaptor signature privacy and a resulting privacy notion achieved by lock-enabling ledgers in Appendix C.

4 PRELIMINARIES

We review adaptor signatures and defer the full definition of them and other basic cryptographic primitives to Appendix B. We define a two-party adaptor signature scheme with respect to a standard two-party signature scheme with aggregatable public keys Σ_2 and a hard relation R. We first recall the notion of a hard relation.

Definition 1 (Hard Relation). *Let R be a relation with statement/witness pairs (Y, y). Let L_R be the associated language $L_R := \{Y \mid \exists y \text{ s.t. } (Y, y) \in R\}$. We say that R is a* hard *if:*

- *There exists a* PPT *sampling algorithm* $\mathsf{GenR}(1^\lambda)$ *that on input the security parameter λ outputs a statement/witness pair $(Y, y) \in R$.*
- *The relation is poly-time decidable.*
- *For all* PPT *adversaries $\mathcal{A}$ there exists a negligible function* negl, *such that:*

$$\Pr\left[(Y, y^*) \in R \;\middle|\; \begin{array}{c}(Y, y) \leftarrow \mathsf{GenR}(1^\lambda),\\ y^* \leftarrow \mathcal{A}(Y)\end{array}\right] \leq \mathsf{negl}(\lambda),$$

where probability is over the randomness of GenR *and $\mathcal{A}$.*

In an adaptor signature scheme, for any statement $Y \in L_R$, a signer holding a secret key can produce a *pre-signature* w.r.t. Y on any message m. Such a pre-signature can be *adapted* into a valid full signature on m if and only if the adaptor knows a witness for Y. Moreover, if such a valid signature is produced, it must be possible to extract the witness for Y given the pre-signature and the adapted signature. Next, we formally define the two-party adaptor signature scheme with aggregatable public keys.

Definition 2 (Two-Party Adaptor Signature Scheme with Aggregatable Public Keys [18]). *A two-party adaptor signature scheme with aggregatable public keys is defined w.r.t. a hard relation R and a two-party signature scheme with aggregatable public keys $\Sigma_2 = (\mathsf{Setup}, \mathsf{KGen}, \Pi_{\mathsf{Sig}}, \mathsf{KAgg}, \mathsf{Vf})$. It is run between parties P_0, P_1 and consists of a tuple $\Xi_2^{R,\Sigma} = (\Pi_{\mathsf{PreSig}}, \mathsf{Adapt}, \mathsf{PreVf}, \mathsf{Ext})$ of efficient protocols and algorithms defined as follows:*

$\Pi_{\mathsf{PreSig}\langle \mathsf{sk}_i, \mathsf{sk}_{1-i}\rangle}(\mathsf{pk}_0, \mathsf{pk}_1, m, Y)$**:** *is an interactive protocol with input secret key sk_i from party P_i with $i \in \{0, 1\}$ and common message $m \in \{0, 1\}^*$, public keys $\mathsf{pk}_0, \mathsf{pk}_1$ and statement $Y \in L_R$, outputs a pre-signature $\hat{\sigma}$.*

$\mathsf{PreVf}(\mathsf{apk}, m, Y, \hat{\sigma})$**:** *is a* DPT *algorithm with input an aggregated public key* apk, *a message $m \in \{0, 1\}^*$, a statement $Y \in L_R$ and a pre-signature $\hat{\sigma}$, outputs bit b.*

$\mathsf{Adapt}(\mathsf{apk}, \hat{\sigma}, y)$**:** *is a* DPT *algorithm with input an aggregated public key* apk, *pre-signature $\hat{\sigma}$ and witness y, outputs a signature σ.*

$\mathsf{Ext}(\mathsf{apk}, \sigma, \hat{\sigma}, Y)$**:** *is a* DPT *algorithm with input an aggregated public key* apk, *a signature σ, pre-signature $\hat{\sigma}$ and statement $Y \in L_R$, outputs a witness y s.t. $(Y, y) \in R$, or $\perp$.*

In addition to the standard signature correctness, an adaptor signature scheme has to satisfy *pre-signature correctness.* Informally, it says that an honestly generated pre-signature w.r.t. a statement $Y \in L_R$ is valid and can be adapted into a valid signature from which a witness for Y can be extracted.

We review the security properties of a two-party adaptor signature scheme with aggregatable public keys: *Unforgeability* resembles two-party existential unforgeability under chosen message attacks (2-EUF-CMA) but additionally requires that producing a forgery σ for some message m is hard even given a pre-signature on m w.r.t. a random statement $Y \in L_R$. Allowing the adversary to learn a pre-signature on the forgery message m is crucial as for our applications unforgeability needs to hold even in case the adversary learns a pre-signature for m without knowing a witness for Y.

Pre-signature adaptability requires that any valid pre-signature w.r.t. Y (possibly produced by a malicious signer) can be adapted into a valid signature using the witness y with $(Y, y) \in R$.

Finally, *witness extractability* guarantees that a valid signature/pre-signature pair $(\sigma, \hat{\sigma})$ for a message/statement pair (m, Y) can be used to extract the corresponding witness y of Y.

We formally define these properties in Appendix B. Combining the three properties described above, we can define a secure adaptor signature scheme as follows.

Definition 3 (Secure Two-Party Adaptor Signature Scheme)**.** *A two-party adaptor signature scheme with aggregatable public keys $\Xi_2^{R,\Sigma}$ is secure if it is* 2-aEUF-CMA *secure, two-party pre-signature adaptable and two-party witness extractable.*

Finally, we review the Universal Composability (UC) framework, which we use in our framework LedgerLocks. We briefly overview the notion of secure realization in UC framework [12]. Intuitively, a protocol realizes an ideal functionality if any distinguisher, i.e., the environment, cannot distinguish between a real run of the protocol and a simulated interaction with the ideal functionality.

Let π be a protocol. The output of an environment $\mathcal{E}$ interacting with protocol π and an adversary $\mathcal{A}$, on input the security parameter 1^λ and auxiliary input z, is denoted as $\mathsf{EXEC}_{\pi,\mathcal{A},\mathcal{E}}(1^\lambda, z)$. Let $\phi_\mathcal{F}$ be the ideal protocol for an ideal functionality $\mathcal{F}$, i.e., $\phi_\mathcal{F}$ is a trivial protocol in which the parties simply forward their inputs to the ideal functionality $\mathcal{F}$. The output of an environment $\mathcal{E}$ interacting with protocol $\phi_\mathcal{F}$ and an adversary $\mathcal{S}$ (also called the simulator), on input the security parameter 1^λ and auxiliary input z, is denoted as $\mathsf{EXEC}_{\phi_\mathcal{F},\mathcal{S},\mathcal{E}}(1^\lambda, z)$.

The main security notion of the UC framework informally says that if a protocol π UC-realizes an ideal functionality $\mathcal{F}$, then any attack that can be carried out against the real-world protocol π can also be carried out against the ideal protocol $\phi_\mathcal{F}$.

Definition 4 (UC Security)**.** *We say a protocol π UC-realizes an ideal functionality $\mathcal{F}$, if for every adversary $\mathcal{A}$ there exists an adversary $\mathcal{S}$ such that*

$$\left\{\mathsf{EXEC}_{\pi,\mathcal{A},\mathcal{E}}(1^\lambda, z)\right\}_{\substack{\lambda\in\mathbb{N},\\ z\in\{0,1\}^*}} \approx_c \left\{\mathsf{EXEC}_{\phi_\mathcal{F},\mathcal{S},\mathcal{E}}(1^\lambda, z)\right\}_{\substack{\lambda\in\mathbb{N},\\ z\in\{0,1\}^*}}$$

(where $\approx_c$ denotes computational indistinguishability).

5 GLOBAL CONDITIONS

In existing blockchain protocols, adaptor signatures and their associated conditions are analyzed within monolithic protocol descriptions. Here, we advocate for handling conditions in a modular fashion instead, using a standalone global functionality $\mathcal{G}_{\mathsf{Cond}}$.

Global conditions functionality. We illustrate the (global) ideal functionality $\mathcal{G}_{\mathsf{Cond}}$ for conditions in Figure 7. $\mathcal{G}_{\mathsf{Cond}}$ is parameterized by a hard relation R and merging function f_{merge}. $\mathcal{G}_{\mathsf{Cond}}$ provides three interfaces: The individual conditions interface acts as a bulletin board for conditions created outside the ideal functionality, and just stores the input condition/opening (i.e., statement/witness) pair in the list $\mathcal{L}$. The merged conditions interface models the creation of a condition as the composition of two other conditions, where the concrete composition operations are given by f_{merge}. This function is split into an operation + on witnesses and an operation $\cdot$ on statements, which need to satisfy that $(Y_1 \cdot Y_2, y_1 + y_2) \in R$ if both $(Y_1, y_1), (Y_2, y_2) \in R$. The open condition interface allows checking if a condition/opening pair is valid (i.e., is in $\mathcal{L}$).

Global conditions protocol. We describe the global conditions protocol Π_{Cond} in Figure 8 for DLOG. The protocol is parameterized with a group description $(\mathbb{G}, g, q)$ and the discrete logarithm (DLOG) relation R_{DLOG} over it, i.e., $(Y, y) \in R_{\mathsf{DLOG}} \iff Y = g^y$. We assume that the group $\mathbb{G}$ is a DLOG-hard group here. The function f_{merge} defines the witness operation (+) as addition and the statment operation $(\cdot)$ as the group operation.

In the case of individual conditions, the protocol checks if the input condition/opening pair (i.e., statement/witness pair for R_{DLOG}) is valid, and in such case, returns it. In the case of merged conditions, the protocol multiplies the inputted condition to form the merged condition, which gets returned by this process. Lastly, for opening conditions, the protocol validates the membership of input condition/opening (i.e., statement/witness) pair in the relation R_{DLOG}, and returns the output bit b.

Ideal Functionality $\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}$

The functionality interacts with an adversary $\mathcal{S}$ and set of parties $\mathcal{P} = \{P_1, \ldots, P_n\}$. Additionally, the functionality maintains a list $\mathcal{L}$ that is indexed by conditions and stores their corresponding openings. The functionality is parameterized by a hard relation R and a function f_{merge} for which the following invariant holds: $(Y_1, y_1) \in R \land (Y_2, y_2) \in R \implies (f_{merge}(stmt, R, (Y_1, Y_2)), f_{merge}(wit, R, y_1, y_2)) \in R$

<u>Individual Conditions:</u> Upon receiving $(\mathsf{create\text{-}ind\text{-}cond}, \mathsf{sid}, (Y, y))$ from some party P, check if $(Y, y) \in R$. If not, then ignore this request. Else, set $\mathcal{L}[Y] := y$ and send $(\mathsf{created\text{-}ind\text{-}cond}, \mathsf{sid}, Y)$ to P and $\mathcal{S}$.
<u>Merged Conditions:</u> Upon receiving $(\mathsf{create\text{-}merged\text{-}cond}, \mathsf{sid}, (Y_1, Y_2))$ from some party P check if $\mathcal{L}[Y_1] = \bot$ or $\mathcal{L}[Y_2] = \bot$ and then ignore the request. Otherwise, set $Y^* := f_{merge}(stmt, R, (Y_1, Y_2))$, set $y^* := f_{merge}(wit, R, (\mathcal{L}[Y_1], \mathcal{L}[Y_2]))$, set $\mathcal{L}[Y^*] := y^*$ and send $(\mathsf{created\text{-}merged\text{-}cond}, \mathsf{sid}, Y^*)$ to P and $\mathcal{S}$.
<u>Open Conditions:</u> Upon receiving $(\mathsf{open\text{-}cond}, \mathsf{sid}, (Y^*, y^*))$ from some party P^*, set $b := (\mathcal{L}[Y^*] \overset{?}{=} y^*)$ and send $(\mathsf{opened\text{-}cond}, \mathsf{sid}, b)$ to P^* and $\mathcal{S}$.

Figure 7: Ideal functionality $\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}$.

Protocol $\Pi_{\mathsf{Cond}}^{R_{\mathsf{DLOG}}}$

The protocol is parameterized by group description $(\mathbb{G}, g, q)$, and the corresponding discrete logarithm (DLOG) relation R_{DLOG} over it, i.e., $(Y, y) \in R_{\mathsf{DLOG}} \iff Y = g^y$.
<u>Individual Conditions:</u> Party P upon receiving $(\mathsf{create\text{-}ind\text{-}cond}, \mathsf{sid}, (Y, y))$ from $\mathcal{E}$, checks if $(Y, y) \in R_{\mathsf{DLOG}}$. If not, then ignores the request. Otherwise, returns (Y, y).
<u>Merged Conditions:</u> Party P upon receiving $(\mathsf{create\text{-}and\text{-}cond}, \mathsf{sid}, (Y_1, Y_2))$ from $\mathcal{E}$, compute $Y^* := Y_1 \cdot Y_2$ and return Y^*.
<u>Open Conditions:</u> Party P upon receiving $(\mathsf{open\text{-}cond}, \mathsf{sid}, (Y^*, y^*))$ from $\mathcal{E}$, return $((Y^*, y^*) \overset{?}{\in} R_{\mathsf{DLOG}})$.
<u>Definition of f_{merge}:</u>
$f_{merge}(stmt, R, (Y_1, Y_2)) := Y_1 \cdot Y_2$
$f_{merge}(wit, R, (y_1, y_2)) := y_1 + y_2$

Figure 8: Protocol $\Pi_{\mathsf{Cond}}^{R_{\mathsf{DLOG}}}$.

Ideal Functionality $\mathcal{F}_{\mathsf{AdaptSig}}^{R,f_{adapt}}$

The functionality is parameterized by a hard relation R and an adaptation function f_{adapt}. It maintains the list $\mathcal{K}$ that stores all generated keys; the list Q that stores tuples (m, σ, v, f) representing message, signature, key and a verification flag, and the list $\mathcal{P}$ that stores tuples $(m, \hat{\sigma}, \sigma, v, Y, y, f)$ representing message, pre-signature, signature, key, condition, witness, and pre-verification flag. All lists are indexed by a session identifier and are initially set to $\emptyset$.

Key Generation: Upon receiving $(\mathsf{keygen}, \mathsf{sid})$ from P_0 and P_1, verify that $\mathsf{sid} = (P_0, P_1, \mathsf{sid}')$ for some sid'. If not, ignore the request. Else, send $(\mathsf{keygen}, \mathsf{sid})$ to $\mathcal{S}$. Upon receiving $(\text{verification-key}, \mathsf{sid}, v)$ from $\mathcal{S}$, add v into $\mathcal{K}[\mathsf{sid}]$ and send $(\text{verification-key}, \mathsf{sid}, v)$ to P_0 and P_1.

Adaptation: Upon receiving $(\mathsf{adapt}, \mathsf{sid}, \hat{\sigma}, v, y)$ from some party P, check if there is an entry $\ell := (m, \hat{\sigma}, \bot, v, Y, \bot, 1) \in \mathcal{P}[\mathsf{sid}]$. If not, then ignore this request. Else, send $(\text{open-cond}, \mathsf{sid}, (Y, y))$ to $\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}$. Upon receiving $(\text{opened-cond}, \mathsf{sid}, b)$, if $b = 0$, then abort. Else set $\sigma := f_{adapt}(\hat{\sigma}, y)$ and update ℓ as $(m, \hat{\sigma}, \sigma, v, Y, y, 1)$ in $\mathcal{P}[\mathsf{sid}]$, add $(m, \sigma, v, 1)$ into $Q[\mathsf{sid}]$, and send $(\text{adapted-signature}, \mathsf{sid}, \sigma)$ to P. *(This guarantees pre-signature adaptability: any valid pre-signature $\hat{\sigma}$ can be adapted into a valid full signature σ using the witness y.)*

Extraction: Upon receiving $(\mathsf{extract}, \mathsf{sid}, \sigma, \hat{\sigma}, v)$ from some party P, check if there is an entry $(m, \hat{\sigma}, \sigma, v, Y, y, 1)$ in $\mathcal{P}[\mathsf{sid}]$. If not, then send $(\mathsf{witness}, \mathsf{sid}, \bot)$ to P, otherwise, send $(\mathsf{witness}, \mathsf{sid}, y)$ to P. *(This guarantees witness extractability: any valid signature/pre-signature pair $(\sigma, \hat{\sigma})$ can be used to extract the corresponding witness y.)*

(Pre-)Signature Generation: Upon receiving $(\mathsf{sign}, \mathsf{sid}, m, v, Y, \mathsf{type})$ from P_0 and P_1, verify that $\mathsf{sid} = (P_0, P_1, \mathsf{sid}')$ for some sid' and $v \in \mathcal{K}[\mathsf{sid}]$. If not, ignore the request. Else, send $(\mathsf{sign}, \mathsf{sid}, m, v, Y, \mathsf{type})$ to $\mathcal{S}$. Upon receiving $(\mathsf{signature}, \mathsf{sid}, m, \sigma)$ from $\mathcal{S}$,

- if $\mathsf{type} =$ *signature* and $(m, \sigma, v, 0) \notin Q[\mathsf{sid}]$, then add $(m, \sigma, v, 1)$ into $Q[\mathsf{sid}]$;
- if $\mathsf{type} =$ *pre-signature* and $(m, \sigma, \bot, v, Y, \bot, 0) \notin \mathcal{P}[\mathsf{sid}]$, then add $(m, \hat{\sigma} := \sigma, \bot, v, Y, \bot, 1)$ into $\mathcal{P}[\mathsf{sid}]$.

If any of the above checks fail, then output an error and halt. Otherwise, output $(\mathsf{signature}, \mathsf{sid}, \sigma)$ to P_0 and P_1.

(Pre-)Signature Verification: Upon receiving $(\mathsf{verify}, \mathsf{sid}, m, \sigma, v, Y, \mathsf{type})$ from some party P, send $(\mathsf{verify}, \mathsf{sid}, m, \sigma, v, Y, \mathsf{type})$ to $\mathcal{S}$. Upon receiving $(\mathsf{verified}, \mathsf{sid}, m, \phi)$ from $\mathcal{S}$, do the following:

- If $v \in \mathcal{K}[\mathsf{sid}]$, and $(m, \sigma, v, 1) \in Q[\mathsf{sid}]$ (if $\mathsf{type} =$ *signature*) or $(m, \hat{\sigma} := \sigma, \cdot, v, Y, \cdot, 1) \in \mathcal{P}[\mathsf{sid}]$ (if $\mathsf{type} =$ *pre-signature*), then set $f = 1$. *(This condition guarantees completeness: if the verification key v is registered before and σ is a legitimately generated (pre-)signature for m, then the verification succeeds.)*
- Else, if $v \in \mathcal{K}[\mathsf{sid}]$, the signers are not corrupted, and $(m, \sigma', v, 1) \notin Q[\mathsf{sid}]$ (if $\mathsf{type} =$ *signature*) or $(m, \sigma', \cdot, v, Y, \cdot, 1) \notin \mathcal{P}[\mathsf{sid}]$ (if $\mathsf{type} =$ *pre-signature*) for any σ', then set $f = 0$ and add $(m, \sigma, v, 0)$ into $Q[\mathsf{sid}]$ (if $\mathsf{type} =$ *signature*) or add $(m, \hat{\sigma} := \sigma, \bot, v, Y, \bot, 0)$ into $\mathcal{P}[\mathsf{sid}]$ (if $\mathsf{type} =$ *pre-signature*). *(This condition guarantees unforgeability: if v is one of the registered keys, the signers are not corrupted and never (pre-)signed m, then the verification fails.)*
- Else, if there exists $(m, \sigma, v, f') \in Q[\mathsf{sid}]$ (if $\mathsf{type} =$ *signature*) or $(m, \hat{\sigma} := \sigma, \cdot, v, Y, \cdot, f') \in \mathcal{P}[\mathsf{sid}]$ (if $\mathsf{type} =$ *pre-signature*), then set $f = f'$. *(This guarantees consistency: all verification requests with identical parameters will result in the same answer.)*
- Else, set $f = \phi$, add (m, σ, v, ϕ) into $Q[\mathsf{sid}]$ (if $\mathsf{type} =$ *signature*) or add $(m, \hat{\sigma} := \sigma, \bot, v, Y, \bot, \phi)$ into $\mathcal{P}[\mathsf{sid}]$ (if $\mathsf{type} =$ *pre*−signature).

Output $(\mathsf{verified}, \mathsf{sid}, m, f)$ to P.

Figure 9: Ideal functionality $\mathcal{F}_{\mathsf{AdaptSig}}^{R,f_{adapt}}$.

Security. The security of our construction is established with the following theorem, for which we provide a proof in Appendix E.1.

Theorem 1. *Let $\mathbb{G}$ be a DLOG-hard group, then the protocol $\Pi_{\mathsf{Cond}}^{R_{\mathsf{DLOG}}}$ UC-realizes the ideal functionality $\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}$, for $R = R_{\mathsf{DLOG}}$ and f_{merge} as defined in Figure 8.*

Extensions. Our model can be extended to account for further protocols to create and verify conditions. As an example, in Appendix A we show how to extend our model to account for additional combinations of conditions, e.g., 1-out-of-n. Note that the existing interfaces of $\mathcal{G}_{\mathsf{Cond}}$ would stay unaffected by such extensions and, hence, proofs conducted with respect to the current version of $\mathcal{G}_{\mathsf{Cond}}$ remain valid. Moreover, it might be interesting to analyze constructions for other hard relations such as the RSA assumption (e.g., used in Guillou-Quisquater-based adaptor signature [18]) or the R_{SIS} relation (as used in Dilithium-based adaptor signature [20]).

6 UC ADAPTOR SIGNATURES

We first define an ideal functionality for adaptor signatures, which accounts for multiple keys per session and models two-party key generation (with public key aggregation) and signing. Then, we show that any two-party adaptor signature scheme with aggregatable public keys $\Xi_2^{R,\Sigma}$ securely realizes our ideal functionality.

Two-party adaptor signature functionality. Our signature functionality $\mathcal{F}_{\mathsf{AdaptSig}}$ (shown in Figure 9) extends the digital signature functionality given by Kiayias et al. [27] to adaptor signatures by considering a hard relation R and a deterministic adaptation function f_{adapt}. Furthermore, it accounts for multiple keys per session and models 2-party key generation (with public key aggregation) and the 2-party signing protocol.

The functionality captures the expected correctness and security properties of a two-party adaptor signature as described in §4. More precisely, it captures completeness and consistency, along with (two-party) unforgeability and witness extractability. Pre-signature correctness and adaptability are captured with the help of the function f_{adapt} and global conditions functionality $\mathcal{G}_{\mathsf{Cond}}$.

The parametrization with f_{adapt} is required to stay fully modular with respect to $\mathcal{G}_{\mathsf{Cond}}$. By using $\mathcal{G}_{\mathsf{Cond}}$ in a modular fashion, we do not make any assumption about the creation of conditions, and in particular about which protocol parties know the witness for a condition. However, for parties knowing the witness y, the presignature $\hat{\sigma}$ and the corresponding adapted signature $\sigma = f_{adapt}(\hat{\sigma}, y)$ are distinctly connected. For showing that a protocol Π_{AdaptSig} UC-realizes $\mathcal{F}_{\mathsf{AdaptSig}}$ without assuming any knowledge about which party knows y, we need to ensure that signatures created via $\mathcal{F}_{\mathsf{AdaptSig}}$'s adaptation interface cannot be distinguished from those created through the adaptation algorithm (even for parties knowing y). If we would let the simulator choose σ at this point (as one would usually do in such cases), the simulator could not be guaranteed to know y, nor to complete the signature adequately (without leaking y or making an assumption on condition generation). Consequently, we need to let $\mathcal{F}_{\mathsf{AdaptSig}}$ compute the correct signature based on y, to which end we must fix f_{adapt}.

Two-party adaptor signature protocol and security. We describe how to translate a two-party adaptor signature scheme with aggregatable public keys (from identification scheme) $\Xi_2^{R,\Sigma}$ into a protocol Π_{AdaptSig} in Appendix E.2. Note that the construction is parametric with respect to $\mathcal{G}_{\mathsf{Cond}}$ where $\mathcal{G}_{\mathsf{Cond}}$ needs to support the relation R of $\Xi_2^{R,\Sigma}$. The security is established with the following theorem, which we prove in Appendix E.2.

Theorem 2. *Let $\Xi_2^{R,\Sigma}$ be a secure two-party adaptor signature scheme with aggregatable public keys (from identification scheme) that is composed of a hard relation R and a secure two-party signature scheme Σ_2, then Π_{AdaptSig} UC-realizes the ideal functionality $\mathcal{F}_{\mathsf{AdaptSig}}$.*

Modeling privacy of adaptor signatures in UC. We note that our adaptor signature functionality from Figure 9 does not model any privacy property. We discuss the reason for it here and refer to Appendix C for game-based privacy notions.

Capturing a privacy property (e.g., that the freshly computed signatures and adapted ones are indistinguishable or that the signature does not reveal any information about the corresponding witness used) for two-party adaptor signatures is difficult because it inherently only holds against a third party not involved in (pre-)signature generation and only sees the final signature. However, in the UC security proof we need to consider that the parties actually involved in the two-party (pre-)signing are adversarial, hence, such a third-party privacy notion is not provable.

More technically, in order to model such a privacy notion we need the simulator $\mathcal{S}$ to produce a valid full signature σ without having access to the corresponding witness y. One potential way to achieve this is inside the adaptation interface of $\mathcal{F}_{\mathsf{AdaptSig}}$ to make a call to the simulator $\mathcal{S}$, and let $\mathcal{S}$ return a fresh full signature σ' that is independent of the witness y. Then, we can argue that σ' is indistinguishable from a full signature σ, which is obtained by adapting a pre-signature $\hat{\sigma}$ with the witness y. However, since we are in the two-party adaptor signatures setting, it means that during the simulation we have to assume that one of the two parties involved in (pre-)signature generation is corrupted. Though, if one of the (pre-)signers is adversarial, then it is trivial for the adversary to distinguish different protocol runs, and hence, different signatures, since it is itself involved in the protocol execution.

7 LOCK-ENABLING LEDGER

We model a realistic ledger supporting AS-locked transactions as an ideal functionality $\mathcal{G}_{\mathsf{LedgerLocks}}$. Figure 10 shows how $\mathcal{G}_{\mathsf{LedgerLocks}}$ is constructed from the base ledger $\mathcal{G}_{\mathsf{Ledger}}$ defined in [8].

Lock-enabling ledger functionality. We formally describe functionality $\mathcal{G}_{\mathsf{LedgerLocks}}$ in Figure 11. $\mathcal{G}_{\mathsf{LedgerLocks}}$ builds upon $\mathcal{G}_{\mathsf{Ledger}}$ by adding interfaces, introducing an additional state, and refining the transaction validation check (by instantiating the predicate isValidTx). $\mathcal{G}_{\mathsf{LedgerLocks}}$ keeps three lists to model account management ($\mathcal{L}_{\mathsf{AccId}}$), authorization ($\mathcal{L}_{\mathsf{Auths}}$) and locking of transactions ($\mathcal{L}_{\mathsf{TxsCond}}$). The new validity predicate CheckCond operates on transactions of the form $(\mathbb{A}, \mathtt{tx}')$ where $\mathbb{A}$ denotes the set of accounts controlling the transaction $\mathtt{tx}'$. For checking the validity of a transaction, it checks $\mathcal{L}_{\mathsf{Auths}}$ for authorization of all accounts in $\mathbb{A}$ and invokes CheckBase for further validity checks. In addition to the interfaces for submitting and reading transactions provided by $\mathcal{G}_{\mathsf{Ledger}}$, $\mathcal{G}_{\mathsf{LedgerLocks}}$ provides five interfaces: The account generation interface allows multiple parties to jointly generate an account (added to $\mathcal{L}_{\mathsf{AccId}}$). Although the ideal functionality models multi-party account generation, we note that in our protocol in Figure 12, we only consider two-party account generation as this is sufficient for our envisioned applications. Moreover, a transaction can have several accounts associated to it, contributing to the generality of the ideal functionality definition. In particular, this allows for modeling UTXO-style cryptocurrencies, where a transaction refers to multiple inputs, which may be controlled by different accounts.

The transaction locking interface allows the parties owning an account to jointly create an authorization for a transaction, which is locked under a specified condition and can only be released using the opening information for this condition. This authorization is recorded in $\mathcal{L}_{\mathsf{TxsCond}}$. The transaction release interface allows a party controlling the respective account and knowing the opening information of the condition to submit a locked transaction to the ledger, moving the transaction from $\mathcal{L}_{\mathsf{TxsCond}}$ to $\mathcal{L}_{\mathsf{Auths}}$. The witness signaling interface allows the account parties to extract the condition witness from the published AS-locked transaction.

One subtlety in our model here is that witness signaling is only enabled when the previously released transaction is added to the ledger. Only then we can guarantee that any party (involved in its creation) would have seen both the AS-locked transaction and the released transaction. We model this by checking whether the released transaction is in the ledger's view of the party invoking the witness signaling interface, which can be accessed by the ledger state variable, as defined in $\mathcal{G}_{\mathsf{Ledger}}$. While an honest user is only guaranteed to learn the witness upon inclusion of the transaction in the ledger, a malicious user may learn the witness already upon the transaction's submission to the ledger. As motivated in §2, this situation may occur if the attacker controls both the user participating in the creation of the AS-locked transaction and a miner. We reflect this subtlety in the release interface: If any of the transaction's

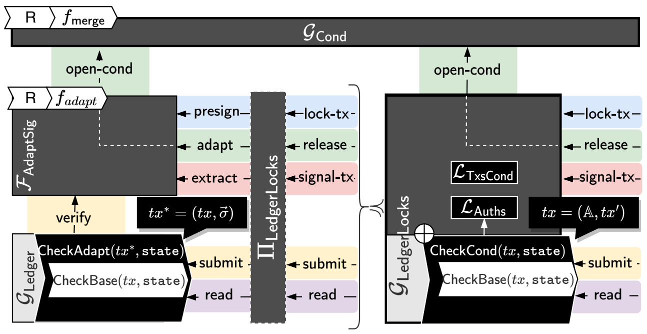

R
f_{merge}
$\mathcal{G}_{\mathsf{Cond}}$
open-cond
R
f_{adapt}
$\mathcal{F}_{\mathsf{AdaptSig}}$
presign
adapt
extract
verify
$tx^* = (tx, \hat{\sigma})$
CheckAdapt(tx^*, state)
CheckBase(tx, state)
$\mathcal{G}_{\mathsf{Ledger}}$
submit
read
$\Pi_{\mathsf{LedgerLocks}}$
lock-tx
release
signal-tx
submit
read
open-cond
$\mathcal{L}_{\mathsf{TxsCond}}$
$\mathcal{L}_{\mathsf{Auths}}$
$tx = (\mathbb{A}, tx')$
$\mathcal{G}_{\mathsf{LedgerLocks}}$
CheckCond(tx, state)
CheckBase(tx, state)
lock-tx
release
signal-tx
submit
read

Figure 10: Realization of $\mathcal{G}_{\mathsf{LedgerLocks}}$ from $\mathcal{G}_{\mathsf{Ledger}}$, $\mathcal{F}_{\mathsf{AdaptSig}}$.

Ideal Functionality $\mathcal{G}_{\text{LedgerLocks}}$

The functionality interacts with an adversary $\mathcal{S}$ and a set of parties $\mathcal{P} = \{P_1, \ldots, P_n\}$. It maintains a set of corrupted parties in C. It uses $\mathcal{L}_{\text{AccId}}$ with entries of the form $(\texttt{AccountId}, (P_1, \ldots, P_m))$, $\mathcal{L}_{\text{Auths}}$ with entries of the form $(\texttt{AccountId}, \texttt{tx})$, and $\mathcal{L}_{\text{TxsCond}}$ with entries of the form $(\text{sid}, \texttt{tx}, Y, \{y, \bot\})$. Moreover, we inherit the **read** and **submit** interfaces from $\mathcal{G}_{\text{Ledger}}$ of [8] (cf. Appendix D).

Account Generation: Upon receiving $(\text{create-account}, \text{sid}, (P_1, \ldots, P_m))$ from P do the following:
- For each P_i in $(P_1, \ldots, P_m)$: Send $(\text{acc-req}, \text{sid}, (P_1, \ldots, P_m, P))$ to P_i and receive $(\text{acc-rep}, \text{sid}, b_i)$. If any $b_i = 0$, then ignore the request.
- Send $(\text{account-req}, \text{sid}, (P_1, \ldots, P_m, P))$ to $\mathcal{S}$, and upon receiving a reply $(\text{account-rep}, \text{sid}, \texttt{AccountId})$, add $(\texttt{AccountId}, (P_1, \ldots, P_m, P))$ in $\mathcal{L}_{\text{AccId}}$ and return $(\text{create-account}, \text{sid}, \texttt{AccountId})$ to all $P_1, \ldots, P_m$ and P.

Authorize TX: Upon receiving $(\text{auth-tx}, \text{sid}, \texttt{tx}, \texttt{AccountId})$ from P, do the following:
- Extract the pair $\alpha := (\texttt{AccountId}, \{P^*\})$ from $\mathcal{L}_{\text{AccId}}$. If it does not exist, then ignore the request.
- Set $\texttt{auth-flag} := 1$. For $P_i \in \{P^*\} \setminus \{P\}$: Send $(\text{auth-req}, \text{sid}, \texttt{tx}, \alpha)$ to P_i and $\mathcal{S}$, and receive $(\text{auth-rep}, \text{sid}, b_i)$ from P_i. If $b_i = 0$, set $\texttt{auth-flag} := 0$.
- If $\texttt{auth-flag} = 1$, store $(\texttt{AccountId}, \texttt{tx})$ in $\mathcal{L}_{\text{Auths}}$.
- Return $(\text{auth-tx}, \text{sid}, \texttt{auth-flag})$ to P.

Lock TX: Upon receiving $(\text{lock-tx}, \text{sid}, \texttt{tx}, \texttt{AccountId}, Y)$ from P, do the following:
- Extract the pair $\alpha := (\texttt{AccountId}, \{P^*\})$ from $\mathcal{L}_{\text{AccId}}$. If it does not exist, then ignore the request.
- Set $\texttt{lock-flag} := 1$. For $P_i \in \{P^*\} \setminus \{P\}$: Send $(\text{lock-req}, \text{sid}, \texttt{tx}, \alpha, Y)$ to P_i and $\mathcal{S}$, and receive $(\text{lock-rep}, \text{sid}, b_i)$ from P_i. If $b_i = 0$, set $\texttt{lock-flag} := 0$.
- If $\texttt{lock-flag} = 1$, store $(\texttt{AccountId}, \texttt{tx}, Y, \bot)$ in $\mathcal{L}_{\text{TxsCond}}$.
- Return $(\text{lock-tx}, \text{sid}, \texttt{lock-flag})$ to P.

Release TX: Upon receiving $(\text{release-tx}, \text{sid}, \texttt{tx}, \texttt{AccountId}, Y, y)$ from some party P, do the following:
- Extract the pair $(\texttt{AccountId}, \{P^*\})$ from $\mathcal{L}_{\text{AccId}}$. If it does not exist, then ignore the request.
- If $P \notin \{P^*\}$, then ignore the request.
- Extract the entry $(\texttt{AccountId}, \texttt{tx}, Y, \bot)$ from $\mathcal{L}_{\text{TxsCond}}$. If it does not exist, then ignore the request.
- Invoke $\mathcal{G}^R_{\text{Cond}}$ on input $(\text{open-cond}, \text{sid}, (Y, y))$ and receive $(\text{opened-cond}, \text{sid}, b)$.
- If $b = 1$, then replace $(\texttt{AccountId}, \texttt{tx}, Y, \bot)$ with $(\texttt{AccountId}, \texttt{tx}, Y, y)$ in $\mathcal{L}_{\text{TxsCond}}$, and store $(\texttt{AccountId}, \texttt{tx})$ in $\mathcal{L}_{\text{Auths}}$.
- Invoke $(\text{submit}, \text{sid}, \texttt{tx})$. Moreover, if $\exists P \in \{P^*\}$, such that $P \in C$, then send $(\text{release-tx}, \text{sid}, y)$ to $\mathcal{S}$.
- Return $(\text{release-tx}, \text{sid}, b)$ to P.

Signal Witness: Upon receiving $(\text{signal-tx}, \text{sid}, \texttt{AccountId}, \texttt{tx}, Y)$ from party P, do the following:
- Extract the pair $(\texttt{AccountId}, \{P^*\})$ from $\mathcal{L}_{\text{AccId}}$. If it does not exist, then ignore the request.
- If $P \notin \{P^*\}$, then ignore the request.
- Set $\text{state}_i := \text{state}|_{min\{\text{pt}_P, |\text{state}|\}}$. Check if $\text{inState}(\texttt{tx}, \text{state}_i)$. Otherwise, ignore the request.
- Extract the entry $(\texttt{AccountId}, \texttt{tx}, Y, w)$ from $\mathcal{L}_{\text{TxsCond}}$, where $w := y$ or $w := \bot$. Otherwise, ignore the request.
- Return $(\text{signal-tx}, \text{sid}, w)$ to P.

Validate Predicate: Our predicate $\text{CheckCond}(\texttt{tx}, \text{state})$ instantiates $\text{isValidTx}(\texttt{tx}, \text{state})$ from [8] as follows:
- Parse $\texttt{tx} := (\mathbb{A}, \texttt{tx}')$. Then, for $\texttt{AccountId}_i \in \mathbb{A}$: Set $b_{1,i} := ((\texttt{AccountId}_i, \texttt{tx}) \in \mathcal{L}_{\text{Auths}})$.
- Set $b_2 := \text{CheckBase}(\texttt{tx}, \text{state})$.
- Return $b_{1,1} \wedge \ldots \wedge b_{1,|\mathbb{A}|} \wedge b_2$.

Figure 11: Ideal functionality $\mathcal{G}_{\text{LedgerLocks}}$. Here, pt_P is P's pointer into the state, as defined for $\mathcal{G}_{\text{Ledger}}$ [8]. Moreover, $\text{inState}(\texttt{tx}, \text{state}) := \exists B \in \text{state},\ \texttt{tx} \in \text{Blockify}^{-1}(B)$, where Blockify is a predicate to parse transactions into a block [8].

account owners is corrupted, the witness is immediately sent to the adversary. As shown in §2, modeling this behavior is crucial for an accurate security analysis of blockchain protocols.

Lock-enabling ledger protocol. Our lock-enabling ledger protocol $\Pi_{\text{LedgerLocks}}$ is defined in the $(\mathcal{F}_{\text{AdaptSig}}, \mathcal{G}_{\text{Ledger}})$-hybrid model and given in Figure 12. During account generation, parties obtain verification keys by making calls to the key generation interface of $\mathcal{F}_{\text{AdaptSig}}$. Analogously, authorization of transactions and locking of transactions happen with a call to the signing interface of $\mathcal{F}_{\text{AdaptSig}}$, where in the latter case, only a pre-signature $\hat{\sigma}$ that is conditioned on Y is computed, whereas in the former a full signature σ over the transaction is computed. Releasing of transactions happens by calling the adaptation interface of $\mathcal{F}_{\text{AdaptSig}}$ with the witness (i.e., opening) y of the corresponding condition (i.e., statement) Y used during the locking procedure. Lastly, witness signaling calls the extraction interface of $\mathcal{F}_{\text{AdaptSig}}$, which returns witness y.

Finally, the validation predicate CheckAdapt verifies the signatures attached to the transactions. Note that the instantation of $\mathcal{G}_{\text{Ledger}}$ used for $\Pi_{\text{LedgerLocks}}$ differs from the one that $\mathcal{G}_{\text{LedgerLocks}}$ extends. More specifically, $\mathcal{G}_{\text{Ledger}}$ used in $\Pi_{\text{LedgerLocks}}$ operates on transactions of the form $\texttt{tx}^* = (\texttt{tx}, \vec{\sigma})$ that (in addition to the account identities of $\texttt{tx}$) hold the signatures $\vec{\text{Sig}}$ that CheckAdapt verifies via $\mathcal{F}_{\text{AdaptSig}}$. To hide this difference in format from a distinguishing environment, $\Pi_{\text{LedgerLocks}}$ wraps the corresponding interfaces of $\mathcal{G}_{\text{Ledger}}$ for reading and submitting.

Security. The security of conditional ledger is captured with the theorem below, which we prove in Appendix E.4.

Theorem 3. *The protocol $\Pi_{\text{LedgerLocks}}$ UC-realizes $\mathcal{G}_{\text{LedgerLocks}}$, in the $(\mathcal{F}_{\text{AdaptSig}}, \mathcal{G}_{\text{Ledger}})$-hybrid model.*

8 LEDGERLOCKS APPLICATIONS

We demonstrate how to use LedgerLocks for modeling AS-based blockchain protocols. To this end, we first give a concrete example

Protocol $\Pi^R_{\mathsf{LedgerLocks}}$

Each party has a list $\mathcal{K}$ with entries (P, vk), a list $\mathcal{P}$ with entries $(\mathtt{tx}, \mathsf{vk}, Y, \hat{\sigma})$, and a list Q with entries $(\mathtt{tx}, \mathsf{vk}, \sigma)$.

Account Generation: Party P upon receiving $(\mathsf{create\text{-}account}, \mathsf{sid}, P')$ from $\mathcal{E}$:

- Party P: Compute $\mathsf{sid}' := (\mathsf{sid}, P, P')$, send (sid') to P', and invoke $\mathcal{F}^{R,f_{adapt}}_{\mathsf{AdaptSig}}$ on input $(\mathsf{keygen}, \mathsf{sid}')$. Receive $(\mathsf{verification\text{-}key}, \mathsf{sid}, \mathsf{vk})$ from $\mathcal{F}^{R,f_{adapt}}_{\mathsf{AdaptSig}}$, and store (P', vk) in $\mathcal{K}$.
- Party P': Receive sid' from P. Invoke $\mathcal{F}^{R,f_{adapt}}_{\mathsf{AdaptSig}}$ on input $(\mathsf{keygen}, \mathsf{sid}')$ and receive $(\mathsf{verification\text{-}key}, \mathsf{sid}, \mathsf{vk})$. Store (P, vk) in $\mathcal{K}$.

Authorize TX: Party P upon receiving $(\mathsf{auth\text{-}tx}, \mathsf{sid}, \mathtt{tx}, P_0, P_1, \mathsf{vk})$ from $\mathcal{E}$:

- Party P: Send $(\mathsf{auth\text{-}req}, \mathsf{sid}, \mathtt{tx}, \{P_0, P_1\})$ to P_0 and P_1 and receive $(\mathsf{auth\text{-}rep}, \mathsf{sid}, f_b)$ from each P_b. If $f_b = 0$, abort.
- Party P: Compute $\mathsf{sid}' := (\mathsf{sid}, P_0, P_1)$ and send $(\mathsf{sid}', \mathsf{vk})$ to P_0 and P_1.
- Party P_b (symmetrically party P_{1-b}): Receive $(\mathsf{sid}', \mathsf{vk})$ from P. Parse $\mathsf{sid}' := (\mathsf{sid}, P_b, P_{1-b})$. Extract (P_{1-b}, vk) from $\mathcal{K}$, and otherwise abort. Invoke $\mathcal{F}^{R,f_{adapt}}_{\mathsf{AdaptSig}}$ on input $(\mathsf{sign}, \mathsf{sid}', \mathtt{tx}, \mathsf{vk}, \bot, \mathit{signature})$ and receive $(\mathsf{signature}, \mathsf{sid}', \sigma)$. Store $(\mathtt{tx}, \mathsf{vk}, \sigma)$ in Q, and send σ to P.
- Party P: Receive σ from P_b and P_{1-b}, store $(\mathtt{tx}, \mathsf{vk}, \sigma)$ in Q.

Lock TX: Party P upon receiving $(\mathsf{lock\text{-}tx}, \mathsf{sid}, \mathtt{tx}, Y, P_0, P_1, \mathsf{vk})$ from $\mathcal{E}$:

- Party P: Send $(\mathsf{pre\text{-}auth\text{-}req}, \mathsf{sid}, \mathtt{tx}, \{P_0, P_1\})$ to P_0 and P_1 and receive $(\mathsf{auth\text{-}rep}, \mathsf{sid}, f_b)$ from each P_b. If $f_b = 0$, abort.
- Party P: Compute $\mathsf{sid}' := (\mathsf{sid}, P_0, P_1)$ and send $(\mathsf{sid}', \mathsf{vk})$ to P_0 and P_1.
- Party P_b (symmetrically party P_{1-b}): Receive $(\mathsf{sid}', \mathsf{vk})$ from P. Parse $\mathsf{sid}' := (\mathsf{sid}, P_b, P_{1-b})$. Extract (P_{1-b}, vk) from $\mathcal{K}$, otherwise, abort. Invoke $\mathcal{F}^{R,f_{adapt}}_{\mathsf{AdaptSig}}$ on input $(\mathsf{sign}, \mathsf{sid}', \mathtt{tx}, \mathsf{vk}, Y, \mathit{pre\text{-}signature})$ and receive $(\mathsf{signature}, \mathsf{sid}', \hat{\sigma})$. Store $(\mathtt{tx}, \mathsf{vk}, Y, \hat{\sigma})$ in $\mathcal{P}$, and send $\hat{\sigma}$ to P.
- Party P: Receive $\hat{\sigma}$ from P_b and P_{1-b}, and store $(\mathtt{tx}, \mathsf{vk}, Y, \hat{\sigma})$ in $\mathcal{P}$.

Release TX: Party P upon receiving $(\mathsf{release\text{-}tx}, \mathsf{sid}, \mathtt{tx}, Y, y, P, P', \mathsf{vk})$ from $\mathcal{E}$:

- Party P: Compute $\mathsf{sid}' := (\mathsf{sid}, P, P')$, extract entry $(\mathtt{tx}, \mathsf{vk}, Y, \hat{\sigma})$ from $\mathcal{P}$, invoke $\mathcal{F}^{R,f_{adapt}}_{\mathsf{AdaptSig}}$ on input $(\mathsf{adapt}, \mathsf{sid}', \hat{\sigma}, \mathsf{vk}, y)$, receive $(\mathsf{adapted\text{-}signature}, \mathsf{sid}', \sigma)$, and store $(\mathtt{tx}, \mathsf{vk}, \sigma)$ in Q.
- Invoke $\mathcal{G}_{\mathsf{Ledger}}$ on input $(\mathsf{submit}, \mathsf{sid}, (\mathtt{tx}, \sigma))$.

Signal Witness: Party P upon receiving $(\mathsf{signal\text{-}tx}, \mathsf{sid}, \mathtt{tx}, Y, \mathsf{vk})$ from $\mathcal{E}$:

- Extract the entry $(\mathtt{tx}, \mathsf{vk}, Y, \hat{\sigma})$ from $\mathcal{P}$, otherwise abort.
- Invoke $\mathcal{G}_{\mathsf{Ledger}}$ on input $(\mathsf{read}, \mathsf{sid})$ and receive the current state.
- Check if $\mathsf{inState}(((\mathsf{vk}_1, \dots \mathsf{vk}_n), \mathtt{tx}'), (\sigma_1, \dots, \sigma_n), \mathsf{state})$ and $\mathsf{vk}_i = \mathsf{vk}$ for some vk_i, otherwise abort.
- Invoke $\mathcal{F}^{R,f_{adapt}}_{\mathsf{AdaptSig}}$ on input $(\mathsf{extract}, \sigma_i, \hat{\sigma}, \mathsf{vk})$, receive $(\mathsf{witness}, \mathsf{sid}, y)$ and return y.

Ledger Read: Party P upon receiving $(\mathsf{read}, \mathsf{sid})$ from $\mathcal{E}$, do the following:

- Invoke $\mathcal{G}_{\mathsf{Ledger}}$ on input $(\mathsf{read}, \mathsf{sid})$ and receive the current state $\mathsf{state} := \mathsf{st}_1 || \dots || \mathsf{st}_n$.
- $\mathsf{state}' := \mathsf{st}_1$. Then, for $\mathsf{st}_2, \dots, \mathsf{st}_n$: Extract $(\mathtt{tx}_1, (\sigma_{1,1}, \dots, \sigma_{1,n})) || \dots || (\mathtt{tx}_m, (\sigma_{m,1}, \dots, \sigma_{m,n'}))$.
- Define new block content $\vec{x}' := \mathtt{tx}_1 || \dots || \mathtt{tx}_m$. Set $\mathsf{state}' := \mathsf{state}' || \mathsf{Blockify}(\vec{x}')$ and return $(\mathsf{read}, \mathsf{sid}, \mathsf{state}')$.

Submit TX: Party P upon receiving $(\mathsf{submit}, \mathsf{sid}, \mathtt{tx})$ from $\mathcal{E}$:

- Parse $\mathtt{tx} := ((\mathsf{vk}_1, \dots, \mathsf{vk}_n), \mathtt{tx}')$ and check that each vk_i is in $\mathcal{K}$. Otherwise, ignore the request.
- Read the state from $\mathcal{G}_{\mathsf{Ledger}}$ as above. For each vk_i, extract the entry $(\mathtt{tx}, \mathsf{vk}_i, \sigma_i)$ from Q. If any of them is missing, then abort.
- Invoke $\mathcal{G}_{\mathsf{Ledger}}$ on input $(\mathsf{submit}, \mathsf{sid}, (\mathtt{tx}, (\sigma_1, \dots, \sigma_n)))$.

Validate Predicate: Our predicate $\mathsf{CheckAdapt}(\mathtt{tx}, \mathsf{state})$ instantiates $\mathsf{isValidTx}(\mathtt{tx}, \mathsf{state})$ in [8] as follows:

- Parse $\mathtt{tx}^* := (((\mathsf{vk}_1, \dots, \mathsf{vk}_n), \mathtt{tx}'), (\sigma_1, \dots, \sigma_n))$. For each pair $(\mathsf{vk}_i, \sigma_i)$, invoke $\mathcal{F}^{R,f_{adapt}}_{\mathsf{AdaptSig}}$ on input $(\mathsf{verify}, \mathsf{sid}, \mathtt{tx}, \sigma_i, \mathsf{vk}_i, \bot, \mathsf{signature})$, receive $(\mathsf{verified}, \mathsf{sid}, \mathtt{tx}, f_i)$, and set $b_{1,i} := f_i$.
- Set $b_2 := \mathsf{CheckBase}(((\mathsf{vk}_1, \dots, \mathsf{vk}_n), \mathtt{tx}'), \mathsf{state})$ and return $b_{1,1} \wedge \dots \wedge b_{1,n} \wedge b_2$.

Figure 12: Protocol $\Pi^R_{\mathsf{LedgerLocks}}$ in the $(\mathcal{F}^{R,f_{adapt}}_{\mathsf{AdaptSig}}, \mathcal{G}_{\mathsf{Ledger}})$-hybrid world.

of an oracle-enabled atomic swap protocol that we express with the help of LedgerLocks. Finally, we provide a general recipe for working with LedgerLocks to model and analyze blockchain protocols that are based on adaptor signatures.

8.1 Case Study: Oracle-enabled Atomic Swaps

In the following, we consider the oracle-enabled atomic swap protocol, as described in §3. We provide the full protocol description in Appendix F (Figure 21 and Figure 22). Here, for illustrative purposes, Figures 14 and 15 show different phases of the protocol.

The key feature of LedgerLocks is that for describing an AS-based blockchain protocol such as the atomic swap protocol considered here, we only need to rely on the functionality $\mathcal{G}_{\mathsf{Cond}}$ and instances of $\mathcal{G}_{\mathsf{LedgerLocks}}$ representing the blockchains involved in the protocol. In particular, we do not need to make use of (adaptor) signatures because their utility is abstracted by the corresponding interfaces of $\mathcal{G}_{\mathsf{LedgerLocks}}$. For the given protocol, we use two instances of $\mathcal{G}_{\mathsf{LedgerLocks}}$ (called $\mathbb{A}$ and $\mathbb{B}$ here in the following) representing the two blockchains between which assets shall be swapped. Thanks to the generality of LedgerLocks, $\mathcal{G}_{\mathsf{LedgerLocks}}$ can easily be instantiated to support additional blockchain features such as scripting capabilities by instantiating the CheckBase predicate. For modeling the atomic swap protocol from §3, the underlying blockchains need to support timelocks. A timelock of a transaction ensures that this particular transaction can only be included starting from a certain *blockheight*. The blockheight denotes the number of blocks in the blockchain and can be accessed through the variable state of $\mathcal{G}_{\text{LedgerLocks}}$ (which is inherited from $\mathcal{G}_{\text{Ledger}}$). To obtain $\mathbb{A}$ and $\mathbb{B}$ with timelock support from $\mathcal{G}_{\text{LedgerLocks}}$, we fix their transaction format to pairs of the form $(\texttt{tx}'', tl)$, where tl denotes the timelock and by defining their CheckBase predicate as follows to support the timelock check:

$$\text{CheckBase}((\mathbb{A}, (\texttt{tx}'', tl)), \text{state}) := |\text{state}| \geq tl$$
$$\wedge\ \text{CheckBase}^{\mathbb{C}}((\mathbb{A}, \texttt{tx}''), \text{state})$$

For generality, we again only specify checks for the desired feature and leave further validity checks to another ledger-specific predicate $\text{CheckBase}^{\mathbb{C}}$.

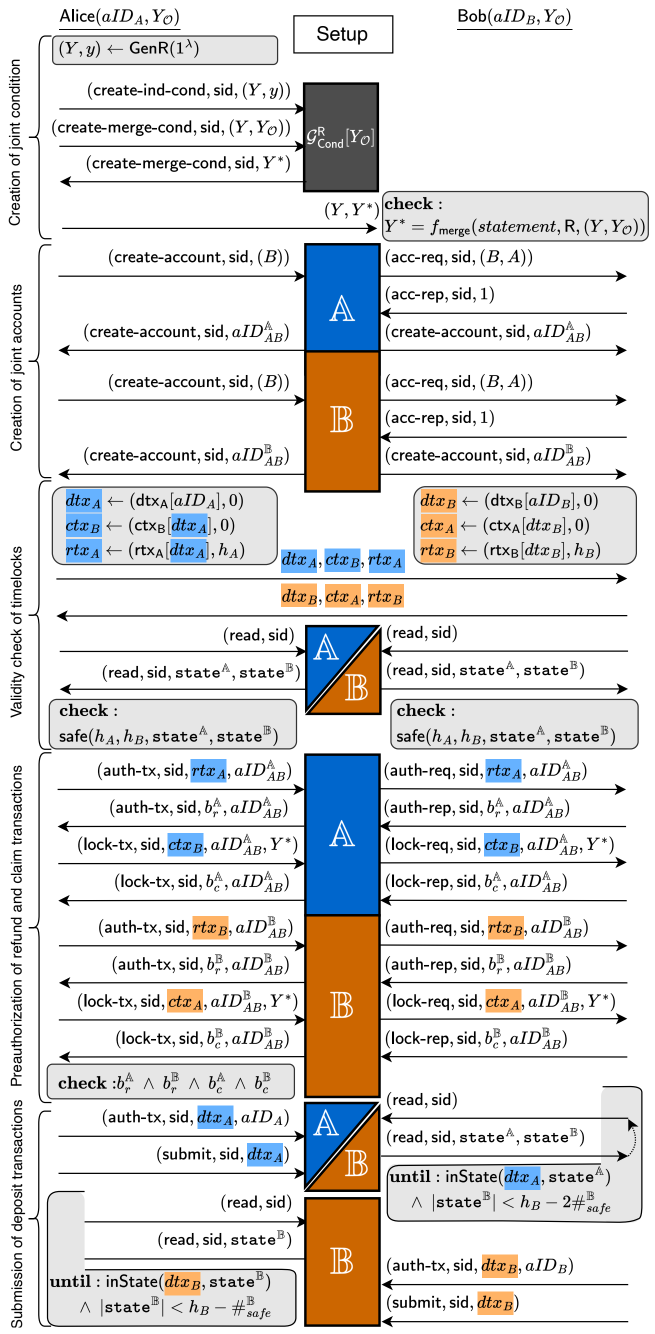
Figure 14 shows the setup phase of the atomic swap protocol. Both parties initially know the oracle condition Y_O, which is assumed to be registered in $\mathcal{G}_{\text{Cond}}$. In the first step, Alice constructs the locking condition Y^* by merging a newly created individual condition Y and the oracle condition Y_O using $\mathcal{G}_{\text{Cond}}$. Next, Alice and Bob create joint accounts on the two ledgers $\mathbb{A}$ and $\mathbb{B}$. Based on these accounts, both of them prepare three kinds of transactions: (1) a deposit transaction *dtx* (which spends funds of the corresponding party to the joint account), (2) a claim transaction *ctx* (which transfers the deposit from the joint account to the other user), and (3) a refund transaction *rtx* (which transfers the deposit back to the original owner). Importantly, ctx_A and rtx_B (along with ctx_B and rtx_A, respectively) are conflicting transactions, which can only be spent with authorization from the joint account. Conflicting means that only one out of them can be published in the ledger. To establish an order on the transaction execution, the refund transactions are equipped with a timelock h. These timelocks will ensure that Alice will always have enough time to safely claim ctx_A and that Bob will always have enough time to claim ctx_B once ctx_A has been claimed. The corresponding property is checked by the predicate safe, which is formally defined in Figure 13. The safe predicate accounts for the fact that $\mathbb{A}$ and $\mathbb{B}$ are independent blockchains that can provide different inclusion guarantees and proceed at different block creation rates. Inherited from $\mathcal{G}_{\text{Ledger}}$, the relevant behavior of a $\mathcal{G}_{\text{LedgerLocks}}$ instance $\mathbb{C}$ is determined by the parameters $\texttt{windowsize}^{\mathbb{C}}$ (the maximum amount of blocks that an honest user can be lacking behind the current state of the blockchain), $\texttt{minTime}^{\mathbb{C}}_{\texttt{window}}$ (the minimal amount of time for including $\texttt{windowsize}^{\mathbb{C}}$ blocks) and $\texttt{maxTime}^{\mathbb{C}}_{\texttt{window}}$ (the maximal amount of time for including $\texttt{windowsize}^{\mathbb{C}}$ blocks). Based on these parameters, we can define $\#^{\mathbb{C}}_{safe} = 5 \cdot \texttt{windowsize}^{\mathbb{C}}$ to be the maximal amount of time that it takes an honest user on ledger $\mathbb{C}$ to include a transaction in reaction to a change in the blockchain state[1].

[1]Up to windowsize blocks may be added to the ledger per round, so when user U makes an observation at blockheight h, the ledger may be at height $h+\texttt{windowsize}-1$. After submission, $\mathcal{G}_{\text{Ledger}}$ (and hence $\mathcal{G}_{\text{LedgerLocks}}$) guarantees a valid transaction to appear in U's view within $4 \cdot \texttt{windowsize}$ blocks.

$$ub_{\mathbb{C}_1 \to \mathbb{C}_2}(n) = \left\lceil \left\lceil \frac{n}{\texttt{windowsize}^{\mathbb{C}_1}} \right\rceil \cdot \frac{\texttt{maxTime}^{\mathbb{C}_1}_{\texttt{window}}}{\texttt{minTime}^{\mathbb{C}_2}_{\texttt{window}}} \right\rceil \cdot \texttt{windowsize}^{\mathbb{C}_2}$$

$$\text{safe}(h_A, h_B, n_A, n_B) = h_B < n_B + ub_{\mathbb{A}\to\mathbb{B}}(\#^{\mathbb{A}}_{safe}) + 2 \cdot \#^{\mathbb{B}}_{safe}$$
$$\wedge\ h_A < n_A + ub_{\mathbb{B}\to\mathbb{A}}(h_B + \#^{\mathbb{B}}_{safe} - n_B) + \#^{\mathbb{A}}_{safe}$$

Figure 13: Notions for safe timelocks in atomic swaps.



Alice(aID_A, Y_O)
Setup
Bob(aID_B, Y_O)
Creation of joint condition
$(Y, y) \leftarrow \text{GenR}(1^\lambda)$
(create-ind-cond, sid, (Y, y))
(create-merge-cond, sid, (Y, Y_O))
(create-merge-cond, sid, Y^*)
$\mathcal{G}^{\mathrm{R}}_{\text{Cond}}[Y_O]$
(Y, Y^*)
check : $Y^* = f_{\text{merge}}(statement, \mathrm{R}, (Y, Y_O))$
Creation of joint accounts
(create-account, sid, (B))
(acc-req, sid, (B, A))
(acc-rep, sid, 1)
(create-account, sid, $aID^{\mathbb{A}}_{AB}$)
$\mathbb{A}$
$\mathbb{B}$
(create-account, sid, $aID^{\mathbb{B}}_{AB}$)
Validity check of timelocks
$dtx_A \leftarrow (\text{dtx}_{\mathsf{A}}[aID_A], 0)$
$ctx_B \leftarrow (\text{ctx}_{\mathsf{B}}[dtx_A], 0)$
$rtx_A \leftarrow (\text{rtx}_{\mathsf{A}}[dtx_A], h_A)$
$dtx_B \leftarrow (\text{dtx}_{\mathsf{B}}[aID_B], 0)$
$ctx_A \leftarrow (\text{ctx}_{\mathsf{A}}[dtx_B], 0)$
$rtx_B \leftarrow (\text{rtx}_{\mathsf{B}}[dtx_B], h_B)$
dtx_A, ctx_B, rtx_A
dtx_B, ctx_A, rtx_B
(read, sid)
(read, sid, $\text{state}^{\mathbb{A}}, \text{state}^{\mathbb{B}}$)
check : safe$(h_A, h_B, \text{state}^{\mathbb{A}}, \text{state}^{\mathbb{B}})$
Preauthorization of refund and claim transactions
(auth-tx, sid, rtx_A, $aID^{\mathbb{A}}_{AB}$)
(auth-req, sid, rtx_A, $aID^{\mathbb{A}}_{AB}$)
(auth-tx, sid, $b^{\mathbb{A}}_r$, $aID^{\mathbb{A}}_{AB}$)
(auth-rep, sid, $b^{\mathbb{A}}_r$, $aID^{\mathbb{A}}_{AB}$)
(lock-tx, sid, ctx_B, $aID^{\mathbb{A}}_{AB}$, Y^*)
(lock-req, sid, ctx_B, $aID^{\mathbb{A}}_{AB}$, Y^*)
(lock-tx, sid, $b^{\mathbb{A}}_c$, $aID^{\mathbb{A}}_{AB}$)
(lock-rep, sid, $b^{\mathbb{A}}_c$, $aID^{\mathbb{A}}_{AB}$)
(auth-tx, sid, rtx_B, $aID^{\mathbb{B}}_{AB}$)
(auth-req, sid, rtx_B, $aID^{\mathbb{B}}_{AB}$)
(auth-tx, sid, $b^{\mathbb{B}}_r$, $aID^{\mathbb{B}}_{AB}$)
(auth-rep, sid, $b^{\mathbb{B}}_r$, $aID^{\mathbb{B}}_{AB}$)
(lock-tx, sid, ctx_A, $aID^{\mathbb{B}}_{AB}$, Y^*)
(lock-req, sid, ctx_A, $aID^{\mathbb{B}}_{AB}$, Y^*)
(lock-tx, sid, $b^{\mathbb{B}}_c$, $aID^{\mathbb{B}}_{AB}$)
(lock-rep, sid, $b^{\mathbb{B}}_c$, $aID^{\mathbb{B}}_{AB}$)
check : $b^{\mathbb{A}}_r \wedge b^{\mathbb{B}}_r \wedge b^{\mathbb{A}}_c \wedge b^{\mathbb{B}}_c$
Submission of deposit transactions
(auth-tx, sid, dtx_A, aID_A)
(submit, sid, dtx_A)
(read, sid)
(read, sid, $\text{state}^{\mathbb{A}}, \text{state}^{\mathbb{B}}$)
until : inState$(dtx_A, \text{state}^{\mathbb{A}})$ $\wedge\ |\text{state}^{\mathbb{B}}| < h_B - 2\#^{\mathbb{B}}_{safe}$
(read, sid, $\text{state}^{\mathbb{B}}$)
until : inState$(dtx_B, \text{state}^{\mathbb{B}})$ $\wedge\ |\text{state}^{\mathbb{B}}| < h_B - \#^{\mathbb{B}}_{safe}$
(auth-tx, sid, dtx_B, aID_B)
(submit, sid, dtx_B)

Figure 14: Setup of the Atomic Swap. inState is a predicate checking for the inclusion of a transaction in the ledger state. dtx_U, ctx_U, and rtx_U denote constructors for the deposit, claim, and refund transactions of user U.

The function $ub_{\mathbb{C}_1 \to \mathbb{C}_2}$ computes an upper bound on the number of blocks that can be added in ledger $\mathbb{C}_2$ while n blocks are added to $\mathbb{C}_1$. Using this, safe can check that given the current blockheight n_B

on ledger $\mathbb{B}$ there is still enough time to include transactions dtx_A on $\mathbb{A}$ (taking up to $ub_{\mathbb{A}\to\mathbb{B}}(\#^{\mathbb{A}}_{safe})$ blocks) and to include transactions dtx^B and ctx^A on $\mathbb{B}$ (taking up to $2 \cdot \#^{\mathbb{B}}_{safe}$ blocks) before reaching timelock h_B (first conjunct). The second conjunct ensures that given current blockheight n_A on ledger $\mathbb{A}$, at the point that an honest Bob learns whether rtx^B that they submitted after h_B has been included on $\mathbb{B}$ (latest after $h_B + \#^{\mathbb{B}}_{safe}$ blocks), the blockheight on $\mathbb{A}$ (computed by $n_A + ub_{\mathbb{B}\to\mathbb{A}}(h_B + \#^{\mathbb{B}}_{safe} - n_B)$) is still at least $\#^{\mathbb{A}}_{safe}$ before h_A so that ctx^B can be safely included.

After checking the timelocks, the users authorize their refund transactions and lock their claim transactions on the condition Y^*. Once this is done, Alice submits dtx_A to $\mathbb{A}$, and once Bob sees it published on $\mathbb{A}$ they submit dtx_B to $\mathbb{B}$ and finish the setup. Note that the setup should be completed in a timely fashion ($|\text{state}| < |h_B| - \#^{\mathbb{B}}_{safe}$) so that there is still time for Alice to claim ctx_A before Bob needs to initiate the refund.

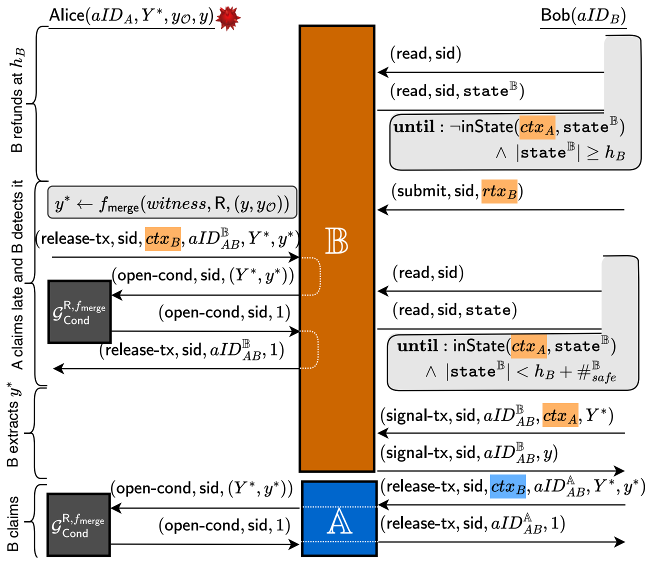
Figure 15 shows the case when a malicious Alice does not submit ctx_A in time. Here, Bob needs to submit rtx_B at the earliest possible point (once h_B is reached). Thanks to $\mathbb{B}$'s transaction inclusion guarantees, Bob can enforce that by $h_B + \#^{\mathbb{B}}_{safe}$, either the refund was successful, or ctx_A must have been published. In the latter case (thanks to the original validity check), there is still enough time ($\#^{\mathbb{A}}_{safe}$) for Bob to submit ctx_B to $\mathbb{A}$ before (at h_A) rtx_A can be submitted and, hence, Bob can complete the protocol as in an honest execution.

The example illustrates how delicate the correct modeling of the blockchain fairness guarantees is to reason about the security of blockchain-based protocols: If the timelocks are wrongly set up, Bob could lose their money, because a malicious Alice could claim ctx_A and still prevent Bob from claiming ctx_B by sneaking the refund rtx_A in before ctx_B. Similarly, in an attack as shown in §2, if h_B is not set properly, Bob could outrun Alice's ctx_A transaction with rtx_B and receive the assets on both chains. Setting up timeouts correctly and reasoning about their security become particularly hard when considering protocols across independent ledgers as shown in the example. Even for stating the checks to be done by the protocol participants (here given through the safe predicate), we need to resort to the safety and liveness guarantees provided by the underlying ledgers – an aspect disregarded in most prior work.

8.2 Template for using LedgerLocks

The oracle-enabled atomic swap case illustrates how to model AS-based blockchain protocols using LedgerLocks. We generalize this approach here and outline further steps towards using LedgerLocks for the verification of AS-based blockchain protocols.

Modeling the protocol. For describing a protocol Π^* (such as $\Pi_{\text{AtomicSwap}}$) with the help of LedgerLocks, Π^* can be defined in a $\mathcal{G}_{\text{Cond}}$,$\mathcal{G}_{\text{LedgerLocks}}$-hybrid world, meaning that it may interact with $\mathcal{G}_{\text{Cond}}$ and $\mathcal{G}_{\text{LedgerLocks}}$. Intuitively, all (adaptor-)signature-related operations of the protocols can be replaced with calls to the corresponding interfaces of $\mathcal{G}_{\text{LedgerLocks}}$. $\mathcal{G}_{\text{Cond}}$ allows for a logical separation between the creation of conditions and their usage to restrict transaction publication on $\mathcal{G}_{\text{LedgerLocks}}$.



Alice($aID_A, Y^*, y_{\mathcal{O}}, y$)
Bob(aID_B)
B refunds at h_B
A claims late and B detects it
B extracts y^*
B claims
(read, sid)
(read, sid, state$^{\mathbb{B}}$)
until : ¬inState(ctx_A, state$^{\mathbb{B}}$) ∧ |state$^{\mathbb{B}}$| ≥ h_B
(submit, sid, rtx_B)
$y^* \leftarrow f_{\text{merge}}(witness, \text{R}, (y, y_{\mathcal{O}}))$
(release-tx, sid, ctx_B, $aID^{\mathbb{B}}_{AB}$, Y^*, y^*)
(open-cond, sid, (Y^*, y^*))
$\mathcal{G}^{\text{R}, f_{\text{merge}}}_{\text{Cond}}$
(open-cond, sid, 1)
(release-tx, sid, $aID^{\mathbb{B}}_{AB}$, 1)
$\mathbb{B}$
(read, sid)
(read, sid, state)
until : inState(ctx_A, state$^{\mathbb{B}}$) ∧ |state$^{\mathbb{B}}$| < $h_B + \#^{\mathbb{B}}_{safe}$
(signal-tx, sid, $aID^{\mathbb{B}}_{AB}$, ctx_A, Y^*)
(signal-tx, sid, $aID^{\mathbb{B}}_{AB}$, y)
(open-cond, sid, (Y^*, y^*))
$\mathcal{G}^{\text{R}, f_{\text{merge}}}_{\text{Cond}}$
(open-cond, sid, 1)
$\mathbb{A}$
(release-tx, sid, ctx_B, $aID^{\mathbb{A}}_{AB}$, Y^*, y^*)
(release-tx, sid, $aID^{\mathbb{A}}_{AB}$, 1)

Figure 15: Atomic Swap: Malicious A claims late.

In cases where Π^* involves several blockchains, the description of Π^* uses multiple instances of $\mathcal{G}_{\text{LedgerLocks}}$ (such as $\mathbb{A}$ and $\mathbb{B}$ in the previous example). The characteristics of these blockchain instances can be further refined by specifying the CheckBase predicate (and correspondingly the transaction format) to describe the logic of transaction execution. For the atomic swap example, we showed how to extend the blockchain logic with timelocks in this way. However, the formalism is expressive enough to encode more involved smart contract logic.

If Π^* requires the creation of conditions with additional properties (that go beyond simple conditions, merged conditions, or 1-out-of-n conditions), then $\mathcal{G}_{\text{Cond}}$'s condition creation interface can be extended to $\mathcal{G}^*_{\text{Cond}}$ to support the new condition types. In this case, for some relation R (known to realize $\mathcal{G}_{\text{Cond}}$) one needs to give a protocol Π_{R} and prove that it realizes the newly added interfaces in $\mathcal{G}^*_{\text{Cond}}$. Alternatively, one can immediately give a protocol Π_{R^*} for some new relation R* (that is known to be supported by an adaptor signature scheme) and show it to realize all of $\mathcal{G}^*_{\text{Cond}}$.

Defining protocol security. LedgerLocks can also serve as a starting point for defining the security of AS-based protocols. An ideal functionality $\mathcal{F}^*$ capturing the desired security of Π^* can be described by extending $\mathcal{G}_{\text{LedgerLocks}}$, e.g., by instantiating the CheckBase predicate that determines which transactions will be considered valid in a faithful protocol execution. Similar to how we extended $\mathcal{G}_{\text{Ledger}}$ to $\mathcal{G}_{\text{LedgerLocks}}$, $\mathcal{F}^*$ may make use of additional state to capture the desired correctness and security properties of the protocol. For cross-chain blockchain protocols, $\mathcal{F}^*$ may hold several internal copies of extended $\mathcal{G}_{\text{LedgerLocks}}$ functionalities.

Proving UC-realization. Finally, one needs to prove Π^* to UC-realize $\mathcal{F}^*$. This proof should not involve cryptographic reductions but focus on how the interactions of Π^* with $\mathcal{G}_{\text{LedgerLocks}}$ and $\mathcal{G}^*_{\text{Cond}}$ are translated into interactions with $\mathcal{F}^*$. As $\mathcal{F}^*$ uses

$\mathcal{G}_{\text{LedgerLocks}}$ as a component, the proof's essence should lie in showing that the way that transactions are created, locked and released within Π^* enforces the transaction inclusion logic encoded in $\mathcal{F}^*$.

Limitations. LedgerLocks, in its current form, is not suitable for modeling blockchain protocols operating on ledgers that do not support transaction authorization through adaptor signature schemes. However, most cryptocurrencies base their transaction authorization on signature schemes shown to support adaptor signatures. One notable exception is the cryptocurrency Zerocash [9].

Further, since $\mathcal{G}_{\text{Ledger}}$ is currently only shown to be realized by the Bitcoin (PoW) backbone protocol [8] and the Ouroboros Genesis (PoS) protocol [7], LedgerLocks (relying on the security of $\mathcal{G}_{\text{Ledger}}$) only provides full end-to-end guarantees for ledgers implementing one of these consensus protocols.

9 CONCLUSION

In this work, we provide foundations for the security of adaptor signatures as well as the applications using them as building blocks for blockchain protocols. We give novel ideal functionalities in the UC framework to model standalone cryptographic conditions, as well as adaptor signatures and lock-enabling ledgers operating upon such conditions. We define concrete protocols and show them to securely realize the different functionalities and finally, we showcase the utility of our model by using it to describe an atomic swap protocol in a clear and modular fashion. In the future, the same blueprint can be used to define other blockchain protocols based on AS-locked transactions.

Acknowledgements. We thank Christian Badertscher for answering our questions about the UC framework. This work was partially supported by the Austrian Science Fund (FWF) through the project PROFET (grant agreement P31621) and the Christian Doppler Research Association through the Christian Doppler Laboratory Blockchain Technologies for the Internet of Things (CDL-BOT); by grant IJC2020-043391-I/MCIN/AEI/10.13039/501100011033, by PRODIGY Project (TED2021-132464B-I00) funded by MCIN/AEI/10.13039/501100011033, and the European Union NextGenerationEU/PRTR. Further, this work has been supported by the Heinz Nixdorf Foundation through a Heinz Nixdorf Research Group (HN-RG) and funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy—EXC 2092 CASA—390781972.

REFERENCES

[1] Lukas Aumayr, Oguzhan Ersoy, Andreas Erwig, Sebastian Faust, Kristina Hostáková, Matteo Maffei, Pedro Moreno-Sanchez, and Siavash Riahi. 2021. Generalized Channels from Limited Blockchain Scripts and Adaptor Signatures. In *ASIACRYPT 2021*.

[2] Lukas Aumayr, Matteo Maffei, Oguzhan Ersoy, Andreas Erwig, Sebastian Faust, Siavash Riahi, Kristina Hostáková, and Pedro Moreno-Sanchez. 2021. Bitcoin-Compatible Virtual Channels. In *2021 IEEE Symposium on Security and Privacy*.

[3] Lukas Aumayr, Pedro Moreno-Sanchez, Aniket Kate, and Matteo Maffei. 2021. Blitz: Secure Multi-Hop Payments Without Two-Phase Commits. In *USENIX Security 2021*.

[4] Lukas Aumayr, Pedro Moreno-Sanchez, Aniket Kate, and Matteo Maffei. 2021. Donner: UTXO-Based Virtual Channels Across Multiple Hops. Cryptology ePrint Archive, Report 2021/855.

[5] Lukas Aumayr, Sri Aravinda Krishnan Thyagarajan, Giulio Malavolta, Pedro Moreno-Sanchez, and Matteo Maffei. 2022. Sleepy Channels: Bi-directional Payment Channels without Watchtowers. In *ACM CCS 2022*.

[6] Christian Badertscher, Ran Canetti, Julia Hesse, Björn Tackmann, and Vassilis Zikas. 2020. Universal Composition with Global Subroutines: Capturing Global Setup Within Plain UC. In *TCC 2020*.

[7] Christian Badertscher, Peter Gazi, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. 2018. Ouroboros Genesis: Composable Proof-of-Stake Blockchains with Dynamic Availability. In *ACM CCS 2018*.

[8] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. 2017. Bitcoin as a Transaction Ledger: A Composable Treatment. In *CRYPTO 2017*.

[9] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. 2014. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *2014 IEEE Symposium on Security and Privacy*.

[10] Manuel Blum, Paul Feldman, and Silvio Micali. 1988. Non-Interactive Zero-Knowledge and Its Applications. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*.

[11] Sergiu Bursuc and Sjouke Mauw. 2022. Contingent payments from two-party signing and verification for abelian groups. Cryptology ePrint Archive, Report 2022/719.

[12] Ran Canetti. 2020. Universally Composable Security. *J. ACM* 67, 5 (sep 2020).

[13] Wei Dai, Tatsuaki Okamoto, and Go Yamamoto. 2022. Stronger Security and Generic Constructions for Adaptor Signatures. Cryptology ePrint Archive, Report 2022/1687.

[14] Stefan Dziembowski, Lisa Eckey, and Sebastian Faust. 2018. FairSwap: How To Fairly Exchange Digital Goods. In *ACM CCS 2018*.

[15] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, and Daniel Malinowski. 2019. Perun: Virtual Payment Hubs over Cryptocurrencies. In *2019 IEEE Symposium on Security and Privacy*.

[16] Stefan Dziembowski, Sebastian Faust, and Kristina Hostáková. 2018. General State Channel Networks. In *ACM CCS 2018*.

[17] Christoph Egger, Pedro Moreno-Sanchez, and Matteo Maffei. 2019. Atomic Multi-Channel Updates with Constant Collateral in Bitcoin-Compatible Payment-Channel Networks. In *ACM CCS 2019*.

[18] Andreas Erwig, Sebastian Faust, Kristina Hostáková, Monosij Maitra, and Siavash Riahi. 2021. Two-Party Adaptor Signatures from Identification Schemes. In *PKC 2021*.

[19] Andreas Erwig and Siavash Riahi. 2022. Deterministic Wallets for Adaptor Signatures. In *European Symposium on Research in Computer Security*.

[20] Muhammed F. Esgin, Oguzhan Ersoy, and Zekeriya Erkin. 2020. Post-Quantum Adaptor Signatures and Payment Channel Networks. In *ESORICS 2020*.

[21] Amos Fiat and Adi Shamir. 1987. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In *CRYPTO'86*.

[22] Noemi Glaeser, Matteo Maffei, Giulio Malavolta, Pedro Moreno-Sanchez, Erkan Tairi, and Sri Aravinda Krishnan Thyagarajan. 2022. Foundations of Coin Mixing Services. In *ACM CCS 2022*.

[23] Louis C. Guillou and Jean-Jacques Quisquater. 1988. Efficient Digital Public-Key Signature with Shadow (Abstract). In *CRYPTO'87*.

[24] Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. 2013. Universally Composable Synchronous Computation. In *TCC 2013*.

[25] Jonathan Katz and Nan Wang. 2003. Efficiency Improvements for Signature Schemes with Tight Security Reductions. In *ACM CCS 2003*.

[26] Aggelos Kiayias and Orfeas Stefanos Thyfronitis Litos. 2020. A Composable Security Treatment of the Lightning Network. In *CSF 2020 Computer Security Foundations Symposium*.

[27] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. 2017. Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol. In *CRYPTO 2017*.

[28] Eike Kiltz, Daniel Masny, and Jiaxin Pan. 2016. Optimal Security Proofs for Signatures from Identification Schemes. In *CRYPTO 2016*.

[29] Thibaut Le Guilly, Nadav Kohen, and Ichiro Kuwahara. 2022. Bitcoin Oracle Contracts: Discreet Log Contracts in Practice. In *2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*.

[30] Varun Madathil, Sri AravindaKrishnan Thyagarajan, Dimitrios Vasilopoulos, Lloyd Fournier, Giulio Malavolta, and Pedro Moreno-Sanchez. 2022. Practical Decentralized Oracle Contracts for Cryptocurrencies. Cryptology ePrint Archive, Report 2022/499.

[31] Giulio Malavolta, Pedro Moreno-Sanchez, Clara Schneidewind, Aniket Kate, and Matteo Maffei. 2019. Anonymous Multi-Hop Locks for Blockchain Scalability and Interoperability. In *NDSS 2019*.

[32] Arash Mirzaei. 2022. Daric: A Storage Efficient Payment Channel With Penalization Mechanism. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume (DSN-S)*.

[33] Joseph Poon and Thaddeus Dryja. 2016. The bitcoin lightning network: Scalable off-chain instant payments. (2016).

[34] Xianrui Qin, Shimin Pan, Arash Mirzaei, Zhimei Sui, Oğuzhan Ersoy, Amin Sakzad, Muhammed F. Esgin, Joseph K. Liu, Jiangshan Yu, and Tsz Hon Yuen. 2022. BlindHub: Bitcoin-Compatible Privacy-Preserving Payment Channel Hubs Supporting Variable Amounts. Cryptology ePrint Archive, Report 2022/1735.

[35] Claus-Peter Schnorr. 1991. Efficient Signature Generation by Smart Cards. *Journal of Cryptology* 4, 3 (Jan. 1991), 161–174.

[36] Erkan Tairi, Pedro Moreno-Sanchez, and Matteo Maffei. 2021. A^2L: Anonymous Atomic Locks for Scalability in Payment Channel Hubs. In *2021 IEEE Symposium*

on Security and Privacy.
[37] Erkan Tairi, Pedro Moreno-Sanchez, and Matteo Maffei. 2021. Post-Quantum Adaptor Signature for Privacy-Preserving Off-Chain Payments. In *FC 2021.*
[38] Sri Aravinda Krishnan Thyagarajan and Giulio Malavolta. 2021. Lockable Signatures for Blockchains: Scriptless Scripts for All Signatures. In *2021 IEEE Symposium on Security and Privacy.*
[39] Sri Aravinda Krishnan Thyagarajan, Giulio Malavolta, and Pedro Moreno-Sanchez. 2022. Universal Atomic Swaps: Secure Exchange of Coins Across All Blockchains. In *2022 IEEE Symposium on Security and Privacy.*
[40] Sri Aravinda Krishnan Thyagarajan, Giulio Malavolta, Fritz Schmidt, and Dominique Schröder. 2020. PayMo: Payment Channels For Monero. Cryptology ePrint Archive, Report 2020/1441.

A EXTENDED GLOBAL CONDITIONS

In this section, we present an extension of the global conditions functionality $\mathcal{G}_{\mathsf{Cond}}$ to model additional protocols to create conditions, in this case, 1-out-of-n, where a set of users can create a set of conditions where the requesting user can only open one of them (see 1-out-of-n illustrative example in §3). The details are shown in Figure 16.

Accordingly, we have extended the protocol from §5 to include the realization of the 1-out-of-n condition (as shown in Figure 17). Concretely, we propose a multi-party protocol where each party provides a random share s_i for each of the conditions Y_j. Finally, the invoking party combines the shares from other users to compute each Y_j except for the position that she commits to, denoted by `index`, where she just uses the condition for which she knows the opening. Finally, she proves in zero-knowledge that the final (set of) conditions are computed correctly.

We analyze the security of the extended global conditions in Appendix E.

B BASIC CRYPTOGRAPHIC PRIMITIVES

Identification scheme. We recall the notion of canonical identification scheme, as in [28]. It can be transformed to a digital signature using Fiat-Shamir heuristic [21].

Definition 5 (Canonical Identification Scheme [28]). *A canonical identification scheme consists of four algorithms* $\mathsf{ID} = (\mathsf{IGen}, \mathsf{P}, \mathsf{ChSet}, \mathsf{V})$*, where*

$\mathsf{IGen}(1^\lambda)$**:** *is a* PPT *algorithm that on input a security parameter* λ *outputs a key pair* $(\mathsf{sk}, \mathsf{pk})$*. We assume that* pk *defines the challenge set* ChSet*.*

P**:** *is a* PPT *algorithm composed of* P_1 *and* P_2*:*
- $\mathsf{P}_1(\mathsf{sk})$*: on input a secret key* sk*, outputs a commitment* $R \in \mathcal{D}_{\mathsf{rand}}$ *and a state* st*.*
- $\mathsf{P}_2(\mathsf{sk}, R, h, \mathsf{st})$*: on input a secret key* sk*, commitment* $R \in \mathcal{D}_{\mathsf{rand}}$*, challenge* $h \in \mathsf{ChSet}$ *and state* st*, outputs a response* $s \in \mathcal{D}_{\mathsf{resp}}$*.*

$\mathsf{V}(\mathsf{pk}, R, h, s)$**:** *is a* DPT *algorithm that on input a public key* pk*, and conversation transcript composed of* (R, h, s)*, outputs a bit* b*.*

We require that for all $(\mathsf{sk}, \mathsf{pk}) \in \mathsf{IGen}(1^\lambda)$*, all* $(R, \mathsf{st}) \in \mathsf{P}_1(\mathsf{sk})$*, all* $h \in \mathsf{ChSet}$ *and all* $s \in \mathsf{P}_2(\mathsf{sk}, R, h, \mathsf{st})$*, we have that* $\mathsf{V}(\mathsf{pk}, R, h, s) = 1$*.*

Digital signature. We recall the definition and security notions of a digital signature.

Definition 6 (Digital Signature). *A signature scheme is a tuple of three algorithms* $\Sigma = (\mathsf{KGen}, \mathsf{Sig}, \mathsf{Vf})$ *defined as:*

$\mathsf{KGen}(1^\lambda)$**:** *is a* PPT *algorithm that on input a security parameter* λ*, outputs a key pair* $(\mathsf{sk}, \mathsf{pk})$*.*

$\mathsf{Sig}(\mathsf{sk}, m)$**:** *is a* PPT *algorithm that on input a secret key* sk *and message* $m \in \{0,1\}^*$*, outputs a signature* σ*.*

$\mathsf{Vf}(\mathsf{pk}, m, \sigma)$**:** *is a* DPT *algorithm that on input a public key* pk*, message* $m \in \{0,1\}^*$ *and signature* σ*, outputs a bit* b*.*

Every signature scheme must satisfy *correctness* meaning that for every $\lambda \in \mathbb{N}$ and every message $m \in \{0,1\}^*$:

$$\Pr\left[\mathsf{Vf}(\mathsf{pk}, m, \mathsf{Sig}(\mathsf{sk}, m)) = 1 \mid (\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KGen}(1^\lambda)\right] = 1.$$

The most common security requirement of a signature scheme is existential unforgeability under chosen message attack (EUF-CMA). On a high level, it guarantees a malicious party, that does not know the private key, cannot produce a valid signature on a message m even if he knows polynomially many valid signatures on messages of his choice (but different from m). Next, we recall this notion.

Definition 7 (EUF-CMA Security). *A signature scheme* Σ *is* EUF-CMA *secure if for every* PPT *adversary* $\mathcal{A}$ *there exists a negligible function* negl *such that*

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

where the experiment $\mathsf{SigForge}_{\mathcal{A},\Sigma}$ *is defined as follows:*

Ideal Functionality $\mathcal{G}_{\mathsf{Cond}}^{R, f_{merge}}$

The functionality interacts with an adversary $\mathcal{S}$ and set of parties $\mathcal{P} = \{P_1, \ldots, P_n\}$. Additionally, the functionality maintains a list $\mathcal{L}$ that is indexed by conditions and stores their corresponding openings. The functionality is parameterized by a hard relation R and a function f_{merge} for which the following invariant holds: $(Y_1, y_1) \in R \wedge (Y_2, y_2) \in R \implies (f_{merge}(stmt, R, (Y_1, Y_2)), f_{merge}(wit, R, y_1, y_2)) \in R$

<u>**Individual Conditions:**</u> Upon receiving $(\text{create-ind-cond}, \text{sid}, (Y, y))$ from some party P, check if $(Y, y) \in R$. If not, then ignore this request. Else, set $\mathcal{L}[Y] := y$ and send $(\text{created-ind-cond}, \text{sid}, Y)$ to P and $\mathcal{S}$.

<u>**1-out-of-n Conditions:**</u> Upon receiving $(\text{create-1-of-n-cond}, \text{sid}, (Y, y), \texttt{index}, n, \{P_i\})$ from some party P, do the following:
- If $(Y, y) \notin R$, then ignore the request. Otherwise, continue.
- For all $i \in [n] \wedge i \neq \texttt{index}$, sample random $(Y_i, y_i) \in R$.
- Set $Y^* := (Y_1, \ldots, Y_{\texttt{index}} := Y, \ldots, Y_n)$.
- For all $P^* \in \{P_i\}$, send $(\text{join-1-of-n-cond}, \text{sid}, P, Y^*)$, and receive back $(\text{joined-1-of-n-cond}, \text{sid}, b_i)$.
- If any $b_i = 0$, then abort. Otherwise, continue.
- Set $\mathcal{L}[Y^*] = (\bot, \ldots, y_{\texttt{index}} := y, \ldots, \bot)$.
- Send $(\text{created-1-of-n-cond}, \text{sid}, Y^*)$ to P and $\mathcal{S}$.

<u>**Merged Conditions:**</u> Upon receiving $(\text{create-merged-cond}, \text{sid}, (Y_1, Y_2))$ from some party P check if $\mathcal{L}[Y_1] = \bot$ or $\mathcal{L}[Y_2] = \bot$ and then ignore the request. Otherwise, set $Y^* := f_{merge}(stmt, R, (Y_1, Y_2))$, set $y^* := f_{merge}(wit, R, (\mathcal{L}[Y_1], \mathcal{L}[Y_2]))$, set $\mathcal{L}[Y^*] := y^*$ and send $(\text{created-merged-cond}, \text{sid}, Y^*)$ to P and $\mathcal{S}$.

<u>**Open Conditions:**</u> Upon receiving $(\text{open-cond}, \text{sid}, (Y^*, y^*))$ from some party P^*, set $b := (\mathcal{L}[Y^*] \overset{?}{=} y^*)$ and send $(\text{opened-cond}, \text{sid}, b)$ to P^* and $\mathcal{S}$.

Figure 16: Ideal functionality $\mathcal{G}_{\mathsf{Cond}}^{R, f_{merge}}$ (extension from Figure 7 with 1-out-of-n conditions).

Protocol $\Pi_{\mathsf{Cond}}^{R_{\mathsf{DLOG}}}$

The protocol is parameterized by group description $(\mathbb{G}, g, q)$, and the corresponding discrete logarithm (DLOG) relation R_{DLOG} over it, i.e., $(Y, y) \in R_{\mathsf{DLOG}} \iff Y = g^y$.

Individual Conditions: Party P upon receiving $(\text{create-ind-cond}, \mathsf{sid}, (Y, y))$ from $\mathcal{E}$, checks if $(Y, y) \in R_{\mathsf{DLOG}}$. If not, then ignores the request. Otherwise, returns (Y, y).

1-out-of-n Conditions:

Party P upon receiving $(\text{create-1-of-n-cond}, (Y, y), \mathsf{index}, n, \{P_i\})$ from $\mathcal{E}$:

- For all $P^* \in \{P_i\}$, send $(n, \{P_i\})$ to P^*.

Party $P^* \in \{P_i\}$ upon receiving $(n, \{P_i\})$ from P:

- For all $j \in [n]$, sample $s_i[j] \leftarrow\$ \mathbb{Z}_q$.
- For all $j \in [n]$, compute $h_i[j] := g^{s[j]}$.
- Send vector $\vec{h}_i := \{h_i[j]\}_{j \in [n]}$ to P and $\{P_i\} \setminus \{P^*\}$.

Party P upon receiving $\vec{h}_i$ from $P^* \in \{P_i\}$:

- For all $j \in [n]$, sample $s[j] \leftarrow\$ \mathbb{Z}_q$.
- For all $j \in [n]$ and $j \neq \mathsf{index}$, compute $f_j := \prod_{i \in |\{P_i\}|} h_i[j]$ and $c[j] := f_j \cdot g^{s[j]}$.
- Set $c[\mathsf{index}] := Y$ and $s[\mathsf{index}] := y$.
- Compute $\pi_{\mathsf{index}} \leftarrow \mathsf{NIZK.P}(\{\exists(\vec{s}, \mathsf{index}) \mid c[\mathsf{index}] = g^{s[\mathsf{index}]} \wedge (\forall j \in [n] \wedge j \neq \mathsf{index}, c[j] = f_j \cdot g^{s[j]})\}, (\vec{s}, \mathsf{index}))$.
- Return $((\vec{c}, \{f_j\}_{j \in [n]}, \pi_{\mathsf{index}}), y)$.

Merged Conditions: Party P upon receiving $(\text{create-and-cond}, \mathsf{sid}, (Y_1, Y_2))$ from $\mathcal{E}$, compute $Y^* := Y_1 \cdot Y_2$ and return Y^*.

Open Conditions: Party P upon receiving $(\text{open-cond}, \mathsf{sid}, (Y^*, y^*))$ from $\mathcal{E}$, return $((Y^*, y^*) \overset{?}{\in} R_{\mathsf{DLOG}})$.

Definition of f_{merge}:

$f_{merge}(stmt, R, (Y_1, Y_2)) := Y_1 \cdot Y_2$

$f_{merge}(wit, R, (y_1, y_2)) := y_1 + y_2$

Figure 17: Protocol $\Pi_{\mathsf{Cond}}^{R_{\mathsf{DLOG}}}$ (extension from protocol in Figure 8 with the functionality for 1-out-of-n conditions).

$\mathsf{SigForge}_{\mathcal{A},\Sigma}(\lambda)$	$\mathcal{O}_{\mathsf{S}}(m)$
$Q \leftarrow \emptyset$	$\sigma \leftarrow \mathsf{Sig}(\mathsf{sk}, m)$
$(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KGen}(1^\lambda)$	$Q := Q \cup \{m\}$
$(m, \sigma) \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{S}}(\cdot)}(\mathsf{pk})$	**return** σ
return $(m \notin Q \wedge \mathsf{Vf}(\mathsf{pk}, m, \sigma))$	

Existential unforgeability does not say anything about the difficulty of transforming a valid signature on m into another valid signature on m. Hardness of such transformation is captured by a stronger notion, called strong existential unforgeability under chosen message attack (or SUF-CMA for short), which we recall next.

Definition 8 (SUF-CMA Security). *A signature scheme Σ is* SUF-CMA *secure if for every* PPT *adversary $\mathcal{A}$ there exists a negligible function* negl *such that*

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

where the experiment $\mathsf{StrongSigForge}_{\mathcal{A},\Sigma}$ *is defined as follows:*

$\mathsf{StrongSigForge}_{\mathcal{A},\Sigma}(\lambda)$	$\mathcal{O}_{\mathsf{S}}(m)$
$Q \leftarrow \emptyset$	$\sigma \leftarrow \mathsf{Sig}(\mathsf{sk}, m)$
$(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KGen}(1^\lambda)$	$Q := Q \cup \{m, \sigma\}$
$(m, \sigma) \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{S}}(\cdot)}(\mathsf{pk})$	**return** σ
return $((m, \sigma) \notin Q \wedge \mathsf{Vf}(\mathsf{pk}, m, \sigma))$	

The advantage of the adversary $\mathcal{A}$ playing the game $\mathsf{StrongSigForge}$ *is defined as follows:*

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{StrongSigForge}} = \Pr[\mathsf{StrongSigForge}_{\mathcal{A},\Sigma}(\lambda) = 1].$$

Two-party signature with aggregatable public keys. We primarily make use of two-party signatures with aggregatable public keys as defined by Erwig et al. [18].

Definition 9 (Two-Party Signature with Aggregatable Public Keys [18]). *A two-party signature scheme with aggregatable public keys is a tuple of protocols and algorithms $\Sigma_2 = (\mathsf{Setup}, \mathsf{KGen}, \Pi_{\mathsf{Sig}}, \mathsf{KAgg}, \mathsf{Vf})$ defined as follows:*

$\mathsf{Setup}(1^\lambda)$: *is a* PPT *algorithm that on input a security parameter λ, outputs public parameters* pp.

$\mathsf{KGen}(\mathsf{pp})$: *is a* PPT *algorithm that on input public parameters* pp, *outputs a key pair* $(\mathsf{sk}, \mathsf{pk})$.

$\Pi_{\mathsf{Sig}\langle \mathsf{sk}_i, \mathsf{sk}_{1-i}\rangle}(\mathsf{pk}_0, \mathsf{pk}_1, m)$: *is an interactive* PPT *protocol that on input secret keys sk_i from party P_i with $i \in \{0, 1\}$ and common values messages $m \in \{0, 1\}^*$ and public keys* $\mathsf{pk}_0, \mathsf{pk}_1$, *outputs a signature σ.*

$\mathsf{KAgg}(\mathsf{pk}_0, \mathsf{pk}_1)$: *is a* DPT *algorithm that on input two public keys* $\mathsf{pk}_0, \mathsf{pk}_1$, *outputs an aggregated public key* apk.

$\mathsf{Vf}(\mathsf{apk}, m, \sigma)$: *is a* DPT *algorithm that on input a public key* pk, *message $m \in \{0, 1\}^*$ and signature σ, outputs a bit b.*

We can define *completeness* for Σ_2 in a natural way: a two-party signature scheme with aggregatable public keys Σ_2 satisfies completeness, if for all public parameters $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$, key pair $(\mathsf{sk}, \mathsf{pk})\mathsf{KGen}(\mathsf{pp})$ and messages $m \in \{0, 1\}^*$, the protocol $\Pi_{\mathsf{Sig}\langle \mathsf{sk}_i, \mathsf{sk}_{1-i}\rangle}(\mathsf{pk}_0, \mathsf{pk}_1, m)$ outputs a signature σ to both parties P_0, P_1, such that $\mathsf{Vf}(\mathsf{apk}, m, \sigma) = 1$, where $\mathsf{apk} := \mathsf{KAgg}(\mathsf{pk}_0, \mathsf{pk}_1)$.

A two-party signature scheme with aggregatable public keys should satisfy *unforgeability.* At a high level, this property guarantees that if one of the two parties is malicious, this party is not able to produce a valid signature under the aggregated public key without the cooperation of the other party. We formalize the property through an experiment $\mathsf{SigForge}^b_{\mathcal{A},\Sigma_2}$, where $b \in \{0, 1\}$ defines which of the two parties is corrupted.

Definition 10 (2-EUF-CMA Security). *A two-party signature scheme with aggregatable public keys Σ_2 is* 2-EUF-CMA *secure if for every* PPT *adversary $\mathcal{A}$ there exists a negligible function* negl *such that, for $b \in \{0, 1\}$,*

$$\Pr[\mathsf{SigForge}^b_{\mathcal{A},\Sigma_2}(\lambda) = 1] \leq \mathsf{negl}(\lambda),$$

where the experiment $\mathsf{SigForge}^b_{\mathcal{A},\Sigma_2}$ *is defined as follows:*

```
$\mathsf{SigForge}^b_{\mathcal{A},\Sigma_2}(\lambda)$
    $Q \leftarrow \emptyset$
    $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$
    $(\mathsf{sk}_{1-b}, \mathsf{pk}_{1-b}) \leftarrow \mathsf{KGen}(\mathsf{pp})$
    $(\mathsf{sk}_b, \mathsf{pk}_b) \leftarrow \mathcal{A}(\mathsf{pp}, \mathsf{pk}_{1-b})$
    $(m, \sigma) \leftarrow \mathcal{A}^{\mathcal{O}^b_{\Pi_\mathsf{S}}(\cdot)}(\mathsf{pk}_{1-b}, \mathsf{sk}_b, \mathsf{pk}_b)$
    $\mathsf{apk} := \mathsf{KAgg}(\mathsf{pk}_0, \mathsf{pk}_1)$
    return $(m \notin Q \wedge \mathsf{Vf}(\mathsf{apk}, m, \sigma))$

$\mathcal{O}^b_{\Pi_\mathsf{S}}(m)$
    $Q := Q \cup \{m\}$
    $\sigma \leftarrow \Pi^{\mathcal{A}}_{\mathsf{Sig}\langle \mathsf{sk}_{1-b}, \cdot \rangle}(\mathsf{pk}_0, \mathsf{pk}_1, m)$
    return $\sigma$
```

Generic transformation to adaptor signature scheme. Erwig et al. [18] showed how to generically transform a canonical identification scheme (as defined in Appendix B) into an adaptor signature scheme (as defined in §4). Here we describe the generic transformation from two-party signature with aggregatable public keys (obtained from an identification scheme) to an adaptor signature scheme, given in [18, Section 5.1]. This transformation is given in Figure 18, and it makes use of the following functions and protocols:

- The randomness shift function $f_{shift}\colon \mathcal{D}_\mathsf{rand} \times L_R \to \mathcal{D}_\mathsf{rand}$, takes as input a commitment value $R \in \mathcal{D}_\mathsf{rand}$ of the identification scheme and a statement $Y \in L_R$ of the hard relation, and outputs a new commitment value $R' \in \mathcal{D}_\mathsf{rand}$. We assume that the inverse of this function is well-defined.
- The adaptation function $f_{adapt}\colon \mathcal{D}_\mathsf{resp} \times \mathcal{D}_\mathsf{w} \to \mathcal{D}_\mathsf{resp}$, takes as input a pre-signature value $\hat{s} \in \mathcal{D}_\mathsf{resp}$ (which corresponds to the response value of the identification scheme) and a witness $y \in \mathcal{D}_\mathsf{w}$ of the hard relation R, and outputs a new value $s \in \mathcal{D}_\mathsf{resp}$.
- The witness extraction function $f_{ext}\colon \mathcal{D}_\mathsf{resp} \times \mathcal{D}_\mathsf{resp} \to \mathcal{D}_\mathsf{w}$, takes as input two response values $\hat{s}, s \in \mathcal{D}_\mathsf{resp}$ and outputs a witness $y \in \mathcal{D}_\mathsf{w}$.
- The randomness combining function $f_{com\text{-}rand}\colon \mathcal{D}_\mathsf{rand} \times \mathcal{D}_\mathsf{rand} \to \mathcal{D}_\mathsf{rand}$, that takes as input two randomness $R_0, R_1 \in \mathcal{D}_\mathsf{rand}$ and outputs a new combined randomness $R \in \mathcal{D}_\mathsf{rand}$.
- The signature combining function $f_{com\text{-}sig}$, that takes as input two partial signatures and returns a new combined signature.
- The randomness exchange protocol $\Pi_\mathsf{Rand\text{-}Exc}$.
- The partial signature exchange protocol Π_Exchange.

In [18] it was shown how to instantiate the functions and protocols specified above for different type of signatures, such as Schnorr [35], Katz-Wang [25] and Guillou-Quisquater [23]. We note here that the recently proposed post-quantum adaptor signatures, such as lattice-based LAS [20] and isogeny-based IAS [37] are also adaptor signature scheme obtained from an identification scheme, and hence, fit this framework.

Adaptor signatures. In this section, we describe the full correctness and security properties of adaptor signatures that were summarized in §4.

Definition 11 (Two-Party Pre-signature Correctness). *A two-party adaptor signature scheme with aggregatable public keys* $\Xi_2^{R,\Sigma}$ *satisfies two-party pre-signature correctness if for every* $\lambda \in \mathbb{N}$, *every message* $m \in \{0,1\}^*$ *and every statement/witness pair* $(Y, y) \in R$, *the following holds:*

$$\Pr\left[\begin{array}{c|l} & \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda) \\ \mathsf{PreVf}(\mathsf{apk}, m, Y, \hat{\sigma}) = 1 & (\mathsf{sk}_0, \mathsf{pk}_0) \leftarrow \mathsf{KGen}(\mathsf{pp}) \\ \wedge & (\mathsf{sk}_1, \mathsf{pk}_1) \leftarrow \mathsf{KGen}(\mathsf{pp}) \\ \mathsf{Vf}(\mathsf{apk}, m, \sigma) = 1 & (Y, y) \leftarrow \mathsf{GenR}(1^\lambda) \\ \wedge & \hat{\sigma} \leftarrow \Pi_{\mathsf{PreSig}\langle \mathsf{sk}_0, \mathsf{sk}_1 \rangle} \\ (Y, y') \in R & \qquad (\mathsf{pk}_0, \mathsf{pk}_1, m, Y) \\ & \mathsf{apk} := \mathsf{KAgg}(\mathsf{pk}_0, \mathsf{pk}_1) \\ & \sigma := \mathsf{Adapt}(\mathsf{apk}, \hat{\sigma}, y) \\ & y' := \mathsf{Ext}(\mathsf{apk}, \sigma, \hat{\sigma}, Y) \end{array}\right] = 1.$$

We now formally define the existential unforgeability under chosen message attack for two-party adaptor signature scheme with aggregatable public keys (2-aEUF-CMA).

Definition 12 (2-aEUF-CMA Security). *A two-party adaptor signature scheme with aggregatable public keys* $\Xi_2^{R,\Sigma}$ *is* 2-aEUF-CMA *secure if for every* PPT *adversary* $\mathcal{A}$ *there exists a negligible function* negl *such that:* $\Pr[\mathsf{aSigForge}^b_{\mathcal{A},\Xi_2^{R,\Sigma}}(\lambda) = 1] \leq \mathsf{negl}(\lambda)$, *where the experiment* $\mathsf{aSigForge}^b_{\mathcal{A},\Xi_2^{R,\Sigma}}$ *is defined as follows:*

```
$\mathsf{aSigForge}^b_{\mathcal{A},\Xi_2^{R,\Sigma}}(\lambda)$
    $Q := \emptyset$; $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$
    $(\mathsf{sk}_{1-b}, \mathsf{pk}_{1-b}) \leftarrow \mathsf{KGen}(\mathsf{pp})$
    $(\mathsf{sk}_b, \mathsf{pk}_b) \leftarrow \mathcal{A}(\mathsf{pp}, \mathsf{pk}_{1-b})$
    $m \leftarrow \mathcal{A}^{\mathcal{O}^b_{\Pi_\mathsf{S}}(\cdot), \mathcal{O}^b_{\Pi_\mathsf{pS}}(\cdot,\cdot)}(\mathsf{pk}_{1-b}, \mathsf{sk}_b, \mathsf{pk}_b)$
    $(Y, y) \leftarrow \mathsf{GenR}(1^\lambda)$
    $\hat{\sigma} \leftarrow \Pi^{\mathcal{A}}_{\mathsf{PreSig}\langle \mathsf{sk}_{1-b}, \cdot \rangle}(m, Y)$
    $\sigma \leftarrow \mathcal{A}^{\mathcal{O}^b_{\Pi_\mathsf{S}}(\cdot), \mathcal{O}^b_{\Pi_\mathsf{pS}}(\cdot,\cdot)}(\hat{\sigma}, Y)$
    $\mathsf{apk} := \mathsf{KAgg}(\mathsf{pk}_0, \mathsf{pk}_1)$
    return $(m \notin Q \wedge \mathsf{Vf}(\mathsf{apk}, m, \sigma))$

$\mathcal{O}^b_{\Pi_\mathsf{S}}(m)$
    $Q := Q \cup \{m\}$
    $\sigma \leftarrow \Pi^{\mathcal{A}}_{\mathsf{Sig}\langle \mathsf{sk}_{1-b}, \cdot \rangle}(\mathsf{pk}_0, \mathsf{pk}_1, m)$
    return $\sigma$

$\mathcal{O}^b_{\Pi_\mathsf{pS}}(m, Y)$
    $Q := Q \cup \{m\}$
    $\hat{\sigma} \leftarrow \Pi^{\mathcal{A}}_{\mathsf{PreSig}\langle \mathsf{sk}_{1-b}, \cdot \rangle}(\mathsf{pk}_0, \mathsf{pk}_1, m, Y)$
    return $\hat{\sigma}$
```

The following definition formalizes the property of *pre-signature adaptability*.

Definition 13 (Two-Party Pre-signature Adaptability). *A two-party adaptor signature scheme with aggregatable public keys* $\Xi_2^{R,\Sigma}$ *satisfies two-party pre-signature adaptability if for any* $\lambda \in \mathbb{N}$, *any message* $m \in \{0,1\}^*$, *any statement/witness pair* $(Y, y) \in R$, *any public keys* pk_0 *and* pk_1, *and any pre-signature* $\hat{\sigma} \leftarrow \{0,1\}^*$ *satisfying* $\mathsf{PreVf}(\mathsf{apk}, m, Y, \hat{\sigma}) = 1$, *where* $\mathsf{apk} := \mathsf{KAgg}(\mathsf{pk}_0, \mathsf{pk}_1)$, *we have:* $\Pr[\mathsf{Vf}(\mathsf{apk}, m, \mathsf{Adapt}(\mathsf{apk}, \hat{\sigma}, y)) = 1] = 1$.

We note that this property is stronger than the pre-signature correctness property from Definition 11, since we require that even maliciously produced pre-signatures, can always be completed into valid signatures.

The last property that we are interested in is *witness extractability*.

Definition 14 (Two-Party Witness Extractability). *A two-party adaptor signature scheme with aggregatable public keys* $\Xi_2^{R,\Sigma}$ *two-party witness extractable if for every* PPT *adversary* $\mathcal{A}$, *there exists a negligible function* negl *such that the following holds:*

```
$\Pi_{\mathsf{PreSig}\langle \mathsf{sk}_i,\mathsf{sk}_{1-i}\rangle}(\mathsf{pk}_0, \mathsf{pk}_1, m, Y)$
Parse $\mathsf{pk}_i = ((1^\lambda, \mathsf{pp}_C, \mathsf{crs}), \mathsf{pk}'_i), i \in \{0,1\}$
$(R_i, \mathsf{st}_i, R_{1-i}) \leftarrow \Pi_{\mathsf{Rand\text{-}Exc}\langle sk_i, sk_{1-i}\rangle}(\mathsf{pp}_C, \mathsf{crs})$
$R_{\mathsf{pre}} := f_{com\text{-}rand}(R_0, R_1)$
$R_{\mathsf{sign}} := f_{shift}(R_{\mathsf{pre}}, Y),\ h := H(R_{\mathsf{sign}}, m)$
$\hat{s}_i \leftarrow \mathsf{P}_2(\mathsf{sk}_i, R_i, h, \mathsf{st}_i)$
$\hat{s}_{i-1} \leftarrow \Pi_{\mathsf{Exchange}}\langle \hat{s}_i, \hat{s}_{i-1}\rangle$
$(h, \hat{s}) := f_{com\text{-}sig}(h, (\hat{s}_i, \hat{s}_{i-1}))$
return $\hat{\sigma} := (h, \hat{s})$

$\mathsf{PreVf}(\mathsf{apk}, m, Y, \hat{\sigma} := (h, \hat{s}))$
$\widehat{R}_{\mathsf{pre}} := \mathsf{V}_0(\mathsf{apk}, h, \hat{s})$
return $h = H(f_{shift}(\widehat{R}_{\mathsf{pre}}, Y), m)$

$\mathsf{Adapt}(\mathsf{apk}, \hat{\sigma} := (h, \hat{s}), y)$
return $\sigma := (h, f_{adapt}(\hat{s}, y))$

$\mathsf{Ext}(\mathsf{apk}, \sigma := (h, s), \hat{\sigma} := (h, \hat{s}), Y)$
return $f_{ext}(s, \hat{s})$
```

Figure 18: Two-party adaptor signature scheme with aggregatable public keys $\Xi_2^{R,\Sigma}$ with respect to Σ_2 and hard relation R.

$$\Pr[\mathsf{aWitExt}^b_{\mathcal{A},\Xi_2^{R,\Sigma}}(\lambda) = 1] \leq \mathsf{negl}(\lambda)$$

where the experiment $\mathsf{aWitExt}^b_{\mathcal{A},\Xi_2^{R,\Sigma}}$ *is defined as follows*

```
$\mathsf{aWitExt}^b_{\mathcal{A},\Xi_{R,\Sigma}}(\lambda)$
$Q := \emptyset;\ \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$
$(\mathsf{sk}_{1-b}, \mathsf{pk}_{1-b}) \leftarrow \mathsf{KGen}(\mathsf{pp})$
$(\mathsf{sk}_b, \mathsf{pk}_b) \leftarrow \mathcal{A}(\mathsf{pp}, \mathsf{pk}_{1-b})$
$(m, Y) \leftarrow \mathcal{A}^{\mathcal{O}^b_{\Pi_\mathsf{S}}(\cdot), \mathcal{O}^b_{\Pi_\mathsf{pS}}(\cdot,\cdot)}(\mathsf{pk}_{1-b}, \mathsf{sk}_b, \mathsf{pk}_b)$
$\hat{\sigma} \leftarrow \Pi^{\mathcal{A}}_{\mathsf{PreSig}\langle \mathsf{sk}_{1-b}, \cdot\rangle}(m, Y)$
$\sigma \leftarrow \mathcal{A}^{\mathcal{O}^b_{\Pi_\mathsf{S}}(\cdot), \mathcal{O}^b_{\Pi_\mathsf{pS}}(\cdot,\cdot)}(\hat{\sigma})$
$\mathsf{apk} := \mathsf{KAgg}(\mathsf{pk}_0, \mathsf{pk}_1)$
$y' := \mathsf{Ext}(\mathsf{apk}, \sigma, \hat{\sigma}, Y)$
return $(m \notin Q \wedge (Y, y') \notin R \wedge \mathsf{Vf}(\mathsf{apk}, m, \sigma))$

$\mathcal{O}^b_{\Pi_\mathsf{S}}(m)$
$Q := Q \cup \{m\}$
$\sigma \leftarrow \Pi^{\mathcal{A}}_{\mathsf{Sig}\langle \mathsf{sk}_{1-b}, \cdot\rangle}(\mathsf{pk}_0, \mathsf{pk}_1, m)$
return $\sigma$

$\mathcal{O}^b_{\Pi_\mathsf{pS}}(m, Y)$
$Q := Q \cup \{m\}$
$\hat{\sigma} \leftarrow \Pi^{\mathcal{A}}_{\mathsf{PreSig}\langle \mathsf{sk}_{1-b}, \cdot\rangle}(\mathsf{pk}_0, \mathsf{pk}_1, m, Y)$
return $\hat{\sigma}$
```

Although the witness extractability experiment aWitExt looks similar to the experiment aSigForge, there is one important difference, namely, the adversary is allowed to choose the forgery statement Y. Hence, we can assume that the adversary knows a witness for Y, and therefore, can generate a valid signature on the forgery message m. However, this is not sufficient to win the experiment. The adversary wins *only* if the valid signature does not reveal a witness for Y.

Non-interactive zero-knowledge proof system. Let R be an efficiently computable binary relation, where for pairs $(x, w) \in R$ we call x the statement and w the witness. Let L be the language consisting of statements in R. A non-interactive zero-knowledge (NIZK) proof system [10] for a language L allows to prove in a non-interactive manner that some statements are in L without leaking information about the corresponding witnesses. We formally define it as follows.

Definition 15 (Non-Interactive Zero-Knowledge Proof System). *A non-interactive zero-knowledge (NIZK) proof system* NIZK *for a language* $L \in \mathsf{NP}$ *(with witness relation* R*) is a tuple of* PPT *algorithms* $\mathsf{NIZK} = (\mathsf{PGen}, \mathsf{P}, \mathsf{V})$*, such that:*

$\mathsf{PGen}(1^\lambda)$**:** *on input a security parameter* λ*, outputs a common reference string* crs*.*

$\mathsf{P}(\mathsf{crs}, x, w)$**:** *on input a common reference string* crs*, a statement* x *and a witness* w*, outputs a proof* π*.*

$\mathsf{V}(\mathsf{crs}, x, \pi)$**:** *on input a common reference string* crs*, a statement* x *and a proof* π*, outputs a bit* b*.*

We require NIZK *to meet the following properties:*

Completeness. *For every* $(x, w) \in R$ *we have that*

$$\Pr\Big[\mathsf{crs} \leftarrow \mathsf{PGen}(1^\lambda), \pi \leftarrow \mathsf{P}(\mathsf{crs}, x, w) \colon \mathsf{V}(\mathsf{crs}, x, \pi) = 1\Big] = 1.$$

Soundness. *For every* $x \notin L$*, and every adversary* $\mathcal{A}$*, we have that*

$$\Pr\Big[\mathsf{crs} \leftarrow \mathsf{PGen}(1^\lambda), \pi \leftarrow \mathcal{A}(\mathsf{crs}, x) \colon \mathsf{V}(\mathsf{crs}, x, \pi) = 1\Big] \leq \mathsf{negl}(\lambda).$$

Zero-Knowledge. *There exists a* PPT *algorithm* $\mathcal{S} = (\mathcal{S}_1, \mathcal{S}_2)$ *such that for every* PPT *adversary* $\mathcal{A}$*,*

$$\mathsf{Adv}^{\mathsf{ZK}}_{\mathcal{A}}(\lambda) := \Big|\Pr\Big[\mathsf{crs} \leftarrow \mathsf{PGen}(1^\lambda) \colon \mathcal{A}^{\mathsf{P}(\mathsf{crs},\cdot,\cdot)}(\mathsf{crs}) = 1\Big] - \Pr\Big[(\mathsf{crs}, \tau) \leftarrow \mathcal{S}_1(1^\lambda) \colon \mathcal{A}^{\mathcal{O}(\mathsf{crs},\tau,\cdot,\cdot)}(\mathsf{crs}) = 1\Big]\Big|$$

is negligible in λ*, where* $\mathcal{O}(\mathsf{crs}, \tau, \cdot, \cdot)$ *is an oracle that outputs* $\perp$ *on input* (x, w) *when* $(x, w) \notin R$ *and outputs* $\pi \leftarrow \mathcal{S}_2(\mathsf{crs}, \tau, x)$ *when* $(x, w) \in R$*.*

C PRIVATE ADAPTOR SIGNATURE AND LEDGER

In this section we extend adaptor signature and lock-enabling ledger with privacy notions.

C.1 Private Adaptor Signatures

We extend the adaptor signature definition from §4 with privacy properties, namely, we define *perfect unlinkability.*

Perfect unlinkability. Informally, unlinkability guarantees that an adversary cannot distinguish freshly computed signatures from adapted ones. Such a property thereby raises the bar in practice for the adversary (e.g., the miner) to behave differently for transactions authorized through adaptor signatures. Note that current miners in blockchains such as Bitcoin and Ethereum do tag transactions in order to e.g., censor them and not include them in a block.[2] [3]

[2]https://www.coindesk.com/tech/2021/05/07/marathon-miners-have-started-censoring-bitcoin-transactions-heres-what-that-means/

[3]https://cointelegraph.com/news/slippery-slope-as-new-bitcoin-mining-pool-censors-transactions

Here we consider a stronger notion of indistinguishability of fresh signatures and adapted ones, dubbed *perfect unlinkability*. It is a strengthening of computational unlinkability definition given by Dai et al. [13]. More precisely, we consider a statistical unlinkability, where the distributions of fresh and adapted signatures are identical. We define it as follows.

Definition 16 (Perfect Unlinkability). *A two-party adaptor signature scheme with aggregatable public keys $\Xi_2^{R,\Sigma}$ is perfectly unlinkable, if for every $\lambda \in \mathbb{N}$, every message $m \in \{0,1\}^*$ and every pair $(Y, y) \in R$, it holds that*

$$[\Pi_{\mathsf{Sig}\langle \mathsf{sk}_0, \mathsf{sk}_1 \rangle}(m)] \text{ and } [\mathsf{Adapt}(\mathsf{apk}, \Pi_{\mathsf{PreSig}\langle \mathsf{sk}_0, \mathsf{sk}_1 \rangle}(m, Y), y)]$$

are identically distributed. Here, $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$, $(\mathsf{sk}_0, \mathsf{pk}_0) \leftarrow \mathsf{KGen}(\mathsf{pp})$, $(\mathsf{sk}_1, \mathsf{pk}_1) \leftarrow \mathsf{KGen}(\mathsf{pp})$, *and* $\mathsf{apk} := \mathsf{KAgg}(\mathsf{pk}_0, \mathsf{pk}_1)$.

Clearly, all schemes that satisfy this stronger notion (stated above) also satisfy the weaker (computational) notion given in [13]. Next, we state that perfect unlinkability is achieved by any two-party adaptor signature with aggregatable public keys constructed from an identification scheme, as described in Appendix B, for which we provide proof in Appendix E.3.

Lemma 1. *A two-party adaptor signature scheme with aggregatable public keys (from identification scheme) $\Xi_2^{R,\Sigma}$ is perfectly unlinkable.*

Private adaptor signatures. Using the above definition we can define a private adaptor signature scheme as follows.

Definition 17 (Private Adaptor Signature Scheme). *A two-party adaptor signature scheme with aggregatable public keys $\Xi_2^{R,\Sigma}$ is private if it is perfectly unlinkable.*

C.2 The LedgerLockTx Primitive

Definition. We define here a primitive, dubbed LedgerLockTx, in order to capture the privacy notions relevant for conditional ledgers §7 in game-based setting.

Definition 18 (LedgerLockTx). *A LedgerLockTx scheme is defined w.r.t a hard relation R and consists of a tuple $\Lambda^R = (\mathsf{Setup}, \Pi_{\mathsf{AccGen}}, \Pi_{\mathsf{AuthTx}}, \Pi_{\mathsf{LockTx}}, \mathsf{RelTx}, \mathsf{SigTx}, \mathsf{VerTx})$ of efficient protocols, run between parties P_b for $b \in \{0,1\}$, and algorithms defined as follows:*

$\mathsf{Setup}(1^\lambda)$: *is a* PPT *algorithm that on input a security parameter λ, outputs public parameters* pp.

$\Pi_{\mathsf{AccGen}}(\mathsf{pp})$: *is an interactive protocol that on input public parameters, outputs secret keys $(\mathsf{sk}_b, \mathsf{sk}_{1-b})$ and a verification key* vk.

$\Pi_{\mathsf{AuthTx}\langle \mathsf{sk}_b, \mathsf{sk}_{1-b} \rangle}(\mathsf{vk}, \mathbf{tx})$: *is an interactive protocol that on input a verification key* vk *and transaction* tx, *outputs a signature σ.*

$\Pi_{\mathsf{LockTx}\langle \mathsf{sk}_b, \mathsf{sk}_{1-b} \rangle}(\mathsf{vk}, \mathbf{tx}, Y)$: *is an interactive protocol that on input a verification key* vk, *transaction* tx *and condition Y, outputs a pre-signature $\hat{\sigma}$.*

$\mathsf{RelTx}(\mathsf{vk}, \mathbf{tx}, \hat{\sigma}, y)$: *is a* DPT *algorithm that on input a verification key* vk, *transaction* tx, *pre-signature $\hat{\sigma}$ and witness y, outputs a signature σ.*

$\mathsf{SigTx}(\mathsf{vk}, \sigma, \hat{\sigma}, Y)$: *is a* DPT *algorithm that on input a verification key* vk, *signature σ, pre-signature $\hat{\sigma}$ and statement $Y \in L_R$, outputs a witness y s.t. $(Y, y) \in R$, or $\bot$.*

$\mathsf{VerTx}(\mathsf{vk}, \mathbf{tx}, \sigma)$: *is a* DPT *algorithm that on input a verification key* vk, *transaction* tx *and signature σ, outputs a bit b.*

Construction. We give an instantiation of LedgerLockTx using two-party adaptor signature scheme with aggregatable public keys (as defined in §4). In our instantiation, account generation Π_{AccGen} corresponds to key generation, transaction authorization Π_{AuthTx} corresponds to full signature generation, and transaction locking Π_{LockTx} corresponds to pre-signature generation using the adaptor signature scheme. Similarly, releasing transaction RelTx and signaling transaction SigTx correspond to adaptation and extraction operations, respectively. Our instantiation is given in Figure 19.

```
$\mathsf{Setup}(1^\lambda)$
    $\mathsf{pp} \leftarrow \Sigma_2.\mathsf{Setup}(1^\lambda)$
    return pp

$\Pi_{\mathsf{AccGen}}(\mathsf{pp})$
    $(\mathsf{sk}_b, \mathsf{pk}_b) \leftarrow \Sigma_2.\mathsf{KGen}(\mathsf{pp})$
    $(\mathsf{sk}_{1-b}, \mathsf{pk}_{1-b}) \leftarrow \Sigma_2.\mathsf{KGen}(\mathsf{pp})$
    $\mathsf{vk} := (\mathsf{pk}_b, \mathsf{pk}_{1-b})$
    return $(\mathsf{sk}_b, \mathsf{vk})$ to $P_b$ and $(\mathsf{sk}_{1-b}, \mathsf{vk})$ to $P_{1-b}$

$\Pi_{\mathsf{AuthTx}\langle \mathsf{sk}_b, \mathsf{sk}_{1-b} \rangle}(\mathsf{vk}, \mathsf{tx})$
    Parse vk as $(\mathsf{pk}_b, \mathsf{pk}_{1-b})$
    $\sigma \leftarrow \Sigma_2.\Pi_{\mathsf{Sig}\langle \mathsf{sk}_b, \mathsf{sk}_{1-b} \rangle}(\mathsf{pk}_b, \mathsf{pk}_{1-b}, \mathsf{tx})$
    return $\sigma$

$\Pi_{\mathsf{LockTx}\langle \mathsf{sk}_b, \mathsf{sk}_{1-b} \rangle}(\mathsf{vk}, \mathsf{tx}, Y)$
    Parse vk as $(\mathsf{pk}_b, \mathsf{pk}_{1-b})$
    $\hat{\sigma} \leftarrow \Xi_2^{R,\Sigma}.\Pi_{\mathsf{PreSig}\langle \mathsf{sk}_b, \mathsf{sk}_{1-b} \rangle}(\mathsf{pk}_b, \mathsf{pk}_{1-b}, \mathsf{tx}, Y)$
    return $\hat{\sigma}$

$\mathsf{VerTx}(\mathsf{vk}, \mathsf{tx}, \sigma)$
    Parse vk as $(\mathsf{pk}_b, \mathsf{pk}_{1-b})$
    $\mathsf{apk} := \Sigma_2.\mathsf{KAgg}(\mathsf{pk}_b, \mathsf{pk}_{1-b})$
    $b := \Sigma_2.\mathsf{Vf}(\mathsf{apk}, \mathsf{tx}, \sigma)$
    return $b$

$\mathsf{RelTx}(\mathsf{vk}, \mathsf{tx}, \hat{\sigma}, y)$                      $\mathsf{SigTx}(\mathsf{vk}, \sigma, \hat{\sigma}, Y)$
    Parse vk as $(\mathsf{pk}_b, \mathsf{pk}_{1-b})$                 Parse vk as $(\mathsf{pk}_b, \mathsf{pk}_{1-b})$
    $\mathsf{apk} := \Sigma_2.\mathsf{KAgg}(\mathsf{pk}_b, \mathsf{pk}_{1-b})$   $\mathsf{apk} := \Sigma_2.\mathsf{KAgg}(\mathsf{pk}_b, \mathsf{pk}_{1-b})$
    $\sigma := \Xi_2^{R,\Sigma}.\mathsf{Adapt}(\mathsf{apk}, \hat{\sigma}, y)$     $y := \Xi_2^{R,\Sigma}.\mathsf{Ext}(\mathsf{apk}, \sigma, \hat{\sigma}, Y)$
    return $\sigma$                                  return $y$
```

Figure 19: Instantiation of LedgerLockTx Λ^R using two-party adaptor signature scheme $\Xi_2^{R,\Sigma}$.

Perfect unlinkability. Analogous to the perfect unlinkability definition for adaptor signatures (see Appendix C.1), we can also define perfect unlinkability for LedgerLockTx, in order to argue that signatures from Π_{AuthTx} and RelTx are identically distributed.

Definition 19 (Perfect Unlinkability for LedgerLockTx). *We say that a LedgerLockTx scheme Λ^R is perfectly unlinkable, if for every $\lambda \in \mathbb{N}$, every transaction $\mathsf{tx} \in \{0,1\}^*$ and every pair $(Y, y) \in R$, it holds that*

$$[\Pi_{\mathsf{AuthTx}\langle \mathsf{sk}_0, \mathsf{sk}_1 \rangle}(\mathsf{vk}, \mathsf{tx})] \text{ and } [\mathsf{RelTx}(\mathsf{vk}, \Pi_{\mathsf{LockTx}\langle \mathsf{sk}_0, \mathsf{sk}_1 \rangle}(\mathsf{vk}, \mathsf{tx}, Y), y)]$$

are identically distributed. Here $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$ *and* $(\mathsf{sk}_0, \mathsf{sk}_1, \mathsf{vk}) \leftarrow \Pi_{\mathsf{AccGen}}(\mathsf{pp})$.

It naturally follows that our LedgerLockTx instantiation from two-party adaptor signatures, as shown in Figure 19, achieves perfect unlinkability.

Lemma 2. *Let* $\Xi_2^{R,\Sigma}$ *be a perfectly unlinkable two-party adaptor signature scheme, then our LedgerLockTx construction from Figure 19 is perfectly unlinkable according to Definition 19.*

D AUXILIARY IDEAL FUNCTIONALITIES

In this section we describe the auxiliary ideal functionalities that we make use of throughout the paper.

Clock functionality. Next, we describe the global clock functionality $\mathcal{G}_{\mathsf{Clock}}$ [6, 24], which allows the parties to proceed in synchronized rounds. More specifically, the functionality keeps track of a round variable whose value the parties can request by sending it $(\mathsf{clock\text{-}read}, \mathsf{sid}_C)$. This value is updated only once all honest parties send $(\mathsf{clock\text{-}update}, \mathsf{sid}_C)$ request to the functionality. The functionality is given in Figure 20.

Ledger functionality. Lastly, we describe the ledger ideal functionality $\mathcal{G}_{\mathsf{Ledger}}$ of Badertscher et al. [8], which is depicted in Figure 34. The functionality makes use of the clock functionality $\mathcal{G}_{\mathsf{Clock}}$ define in Appendix D, and is parameterized by four algorithms Validate, ExtendPolicy, Blockify, and predict-time, along with two parameters windowSize, Delay $\in \mathbb{N}$. We refer the reader to [8] for all the details of this ledger functionality.

E FULL SECURITY ANALYSIS

In this section we provide the full security analysis of our constructions.

E.1 Security Analysis of Global Conditions

We recall the theorem stated in §5, for which we provide a proof here. We note that we prove the extended version of global conditions as defined in Appendix A.

Theorem 4. *Let* NIZK *be a non-interactive zero-knowledge proof system and* $\mathbb{G}$ *be a DLOG-hard group, then the protocol* $\Pi_{\mathsf{Cond}}^{R_{\mathsf{DLOG}}}$ *UC-realizes the ideal functionality* $\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}$*, for* $R = R_{\mathsf{DLOG}}$ *and* f_{merge} *as defined in Figure 8.*

Proof. Throughout the following proof, we implicitly assume that all messages of the adversary are well-formed and we treat the malformed messages as aborts. Since we consider static corruption model, we denote the set of users corrupted by the adversary with C. The proof is composed of a series of hybrids.

Hybrid $\mathcal{H}_0$: This corresponds to the original Π_{Cond}.

Hybrid $\mathcal{H}_1$: All calls to non-interactive zero-knowledge proof system NIZK are simulated using the simulator $\mathcal{S}_{\mathsf{NIZK}}$ for the corresponding language $\mathcal{L}$.

Hybrid $\mathcal{H}_2$: For the set of corrupted parties C, if the adversary outputs $(\mathsf{open\text{-}cond}, (Y^*, y^*))$, such that $(Y^*, y^*) \in R_{\mathsf{DLOG}}$, for the condition Y^* created by party P via $(\mathsf{create\text{-}ind\text{-}cond}, (Y^*, y^*))$ and $P \notin C$, then the experiment aborts by outputting fail_1.

Hybrid $\mathcal{H}_3$: For the set of corrupted parties C, if the adversary outputs $(\mathsf{open\text{-}cond}, (\vec{c}^*[\mathtt{index}^*], y^*))$, such that $(\vec{c}^*[\mathtt{index}^*], y^*) \in R_{\mathsf{DLOG}}$, for the condition $\vec{c}^*$ created by party P via $(\mathsf{create\text{-}1\text{-}of\text{-}n\text{-}cond}, (Y^*, y^*), \mathtt{index}^*, n, \{P_i\})$ and $P \notin C$, then the experiment aborts by outputting fail_2.

Hybrid $\mathcal{H}_4$: For the set of corrupted parties C, if the adversary outputs $(\mathsf{open\text{-}cond}, (Y^*, y^*))$, such that $(Y^*, y^*) \in R_{\mathsf{DLOG}}$, for the condition Y^* created by party P via $(\mathsf{create\text{-}merged\text{-}cond}, (Y_1^*, Y_2^*))$ and $P \notin C$, then the experiment aborts by outputting fail_3.

Simulator $\mathcal{S}$: The simulator $\mathcal{S}$ simulates the honest parties as in the previous hybrid, except that its actions are dictated by the interaction with the ideal functionality $\mathcal{G}_{\mathsf{Cond}}^{R}$. More precisely, the simulator proceeds as in the execution of $\mathcal{H}_4$ by simulating the view of the adversary appropriately as it receives messages from the ideal functionality $\mathcal{G}_{\mathsf{Cond}}$. If the simulated view deviates from the execution of the ideal functionality, then the simulation must have already aborted (as given in cases of abort in the above hybrids).

Next, we proceed to proving the indistinguishability of the neighboring hybrids for the environment $\mathcal{E}$.

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

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

Proof. The proof follows directly from the zero-knowledge property of the non-interactive zero-knowledge proof system NIZK, for which the simulator $\mathcal{S}_{\mathsf{NIZK}}$ is guaranteed to exist. □

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

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

Proof. Let fail_1 be the event that triggers an abort in $\mathcal{H}_2$ but not in $\mathcal{H}_1$. In the following we are going to show that the probability that such an event happens can be bounded by a negligible function in the security parameter. Assume towards contradiction that $\Pr[\mathsf{fail}_1 \mid \mathcal{H}_1] \geq \frac{1}{\mathsf{poly}(\lambda)}$. To show that the probability of fail_1 happening in $\mathcal{H}_2$ cannot be inverse polynomial we reduce it to the hardness of DLOG. The reduction receives as input a group element h, and samples an index $j \in [1, s]$, where $s \in \mathsf{poly}(\lambda)$ is a bound on the total number of sessions. The reduction sets the condition as $Y^* = h$ in the j-th session. If the event fail_1 happens, then the reductions returns the corresponding y^*, otherwise it aborts.

The reduction is clearly efficient, and whenever j is guessed correctly it does not abort. Since fail_1 happens it means that $(Y^*, y^*) \in R_{\mathsf{DLOG}}$, for the condition Y^*, and $P \notin C$. This implies that the reduction succeeded in breaking the DLOG. By assumption this happens with probability at least $\frac{1}{s \cdot \mathsf{poly}(\lambda)}$, which is a contradiction and proves that $\Pr[\mathsf{fail}_1 \mid \mathcal{H}_1] \leq \mathsf{negl}(\lambda)$.

□

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

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

Proof. The proof of this lemma is analogous to that of Lemma 4, but we need to additionally account for $\mathtt{index}^*$. More precisely, since $(\vec{c}^*[\mathtt{index}^*], y^*) \in R_{\mathsf{DLOG}}$, we know that $\vec{c}^*[\mathtt{index}^*] = g^{y^*}$.

Ideal Functionality $\mathcal{G}_{\mathsf{Clock}}$

The functionality manages the set $\mathcal{P}$ of registered identities, i.e., parties $P := (\mathtt{pid}, \mathtt{sid})$. It also manages the set F of functionalities (together with their session identifier). Initially, $\mathcal{P} = \emptyset$ and $F = \emptyset$.
For each session sid the clock maintains a variable τ_{sid}. For each identity $P := (\mathtt{pid}, \mathtt{sid}) \in \mathcal{P}$ it manages variable d_P. For each pair $(\mathcal{F}, \mathsf{sid}) \in F$ it manages variable $d_{\mathcal{F},\mathsf{sid}}$ (all integer variables are initially 0).

Synchronization:

- Upon receiving $(\text{clock-update}, \mathsf{sid}_C)$ from some party $P \in \mathcal{P}$ set $d_P = 1$, execute **Round-Update** and forward $(\text{clock-update}, \mathsf{sid}_C, P)$ to $\mathcal{S}$.
- Upon receiving $(\text{clock-update}, \mathsf{sid}_C)$ from some functionality $\mathcal{F}$ in a session sid such that $(\mathcal{F}, \mathsf{sid}) \in F$ set $d_{(\mathcal{F},\mathsf{sid})} = 1$, execute **Round-Update** and return $(\text{clock-update}, \mathsf{sid}_C, \mathcal{F})$ to this instance of $\mathcal{F}$.
- Upon receiving $(\text{clock-read}, \mathsf{sid}_C)$ from any participant (including the environment on behalf of a party, the adversary, or any ideal—shared or local—functionality) return $(\text{clock-read}, \mathsf{sid}_C, \tau_{\mathsf{sid}})$ to the requestor (where sid is the session identifier of the calling instance).

Round-Update: For each session sid do: If $d_{(\mathcal{F},\mathsf{sid})} = 1$ for all $\mathcal{F} \in F$ and $d_P = 1$ for all honest parties $P := (\cdot, \mathsf{sid}) \in \mathcal{P}$, then set $\tau_{\mathsf{sid}} = \tau_{\mathsf{sid}} + 1$ and reset $d_{(\mathcal{F},\mathsf{sid})} = 0$ and $d_P = 0$ for all parties $P := (\cdot, \mathsf{sid}) \in \mathcal{P}$.

Figure 20: Ideal functionality $\mathcal{G}_{\mathsf{Clock}}$ [6, 24].

Hence, the reduction needs to guess this $\mathtt{index}^* \in [n]$ before embedding the DLOG challenge. Therefore, the reduction incurs an additional $\frac{1}{n}$ loss, where $n \in \mathsf{poly}(\lambda)$.

□

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

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

Proof. The proof of this lemma is analogous to that of Lemma 4. □

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

$$\mathsf{EXEC}_{\mathcal{H}_4,\mathcal{A},\mathcal{E}} \approx \mathsf{EXEC}_{\mathcal{G}^{R}_{\mathsf{Cond}},\mathcal{S},\mathcal{E}}.$$

Proof. The two experiments are identical and the change here is only syntactical. Hence, indistinguishability follows. □

This concludes the proof of Theorem 4. □

E.2 Security Analysis of Two-Party Adaptor Signature

First, we describe how to straightforwardly translate a two-party adaptor signature scheme with aggregatable public keys (from identification scheme) $\Xi_2^{R,\Sigma}$ into a protocol Π_{AdaptSig}[4]. We consider parties P_b for $b \in \{0, 1\}$ running Π_{AdaptSig}, and upon each request we verify that $\mathsf{sid} = (P_b, P_{1-b}, \mathsf{sid}')$ for some sid', and if not, then ignore the request.

- Upon receiving (keygen, sid) from $\mathcal{E}$, generate keys $(\mathsf{sk}_b, \mathsf{pk}_b) \leftarrow \Sigma_2.\mathsf{KGen}(\mathsf{pp})$, for some public parameters pp, store sk_b, and output $(\text{verification-key}, \mathsf{sid}, v := (\mathsf{pk}_b, \mathsf{pk}_{1-b}))$.
- Upon receiving $(\mathsf{sign}, \mathsf{sid}, m, v, Y, \textit{signature})$ from $\mathcal{E}$, parse v as $(\mathsf{pk}_b, \mathsf{pk}_{1-b})$, execute the protocol $\sigma \leftarrow \Sigma_2.\Pi_{\mathsf{Sig}\langle \mathsf{sk}_b, \mathsf{sk}_{1-b}\rangle}(\mathsf{pk}_b, \mathsf{pk}_{1-b}, m)$ and output $(\text{signature}, \mathsf{sid}, \sigma)$.
- Upon receiving $(\mathsf{sign}, \mathsf{sid}, m, v, Y, \textit{pre-signature})$ from $\mathcal{E}$, parse v as $(\mathsf{pk}_b, \mathsf{pk}_{1-b})$ and Y as Y, execute the protocol $\sigma \leftarrow \Xi_2^{R,\Sigma}.\Pi_{\mathsf{PreSig}\langle \mathsf{sk}_b, \mathsf{sk}_{1-b}\rangle}(\mathsf{pk}_b, \mathsf{pk}_{1-b}, m, Y)$, and output $(\text{signature}, \mathsf{sid}, \hat{\sigma})$.
- Upon receiving $(\mathsf{verify}, \mathsf{sid}, m, \sigma, v, Y, \textit{signature})$ from $\mathcal{E}$, parse v as $(\mathsf{pk}_b, \mathsf{pk}_{1-b})$, compute $\mathsf{apk} := \Sigma_2.\mathsf{KAgg}(\mathsf{pk}_b, \mathsf{pk}_{1-b})$ and $f \leftarrow \Sigma_2.\mathsf{Vf}(\mathsf{apk}, m, \sigma)$, and output $(\text{verified}, \mathsf{sid}, m, f)$.
- Upon receiving $(\mathsf{verify}, \mathsf{sid}, m, \sigma, v, Y, \textit{pre-signature})$ from $\mathcal{E}$, parse v as $(\mathsf{pk}_b, \mathsf{pk}_{1-b})$ and Y as Y, compute $\mathsf{apk} := \Sigma_2.\mathsf{KAgg}(\mathsf{pk}_b, \mathsf{pk}_{1-b})$ and $f \leftarrow \Xi_2^{R,\Sigma}.\mathsf{PreVf}(\mathsf{apk}, m, Y, \sigma)$, and output $(\text{verified}, \mathsf{sid}, m, f)$.
- Upon receiving $(\text{adapt}, \mathsf{sid}, \hat{\sigma}, v, y, Y)$ from $\mathcal{E}$, send $(\text{open-cond}, \mathsf{sid}, Y, y)$ to $\mathcal{G}^{R,f_{merge}}_{\mathsf{Cond}}$, and obtain the response $(\text{opened-cond}, \mathsf{sid}, b)$. If $b = 0$, then abort. Else, compute $\sigma := f_{adapt}(\hat{\sigma}, y)$, and output $(\text{adapted-signature}, \mathsf{sid}, \sigma)$.
- Upon receiving $(\text{extract}, \mathsf{sid}, \sigma, \hat{\sigma}, v)$ from $\mathcal{E}$, parse v as $(\mathsf{pk}_b, \mathsf{pk}_{1-b})$, compute $\mathsf{apk} := \Sigma_2.\mathsf{KAgg}(\mathsf{pk}_b, \mathsf{pk}_{1-b})$ and $y \leftarrow \Xi_2^{R,\Sigma}.\mathsf{Ext}(\mathsf{apk}, \sigma, \hat{\sigma}, Y)$, and output $(\text{witness}, \mathsf{sid}, y)$.

Next, we prove that the ideal functionality $\mathcal{F}_{\mathsf{AdaptSig}}$, described in §6, for a hard relation R, is realized by Π_{AdaptSig}.

Theorem 2. *Let $\Xi_2^{R,\Sigma}$ be a secure two-party adaptor signature scheme with aggregatable public keys (from identification scheme) that is composed of a hard relation R and a secure two-party signature scheme Σ_2, then Π_{AdaptSig} UC-realizes the ideal functionality $\mathcal{F}_{\mathsf{AdaptSig}}$.*

Proof. We give a proof by contradiction. Assume that $\Xi_2^{R,\Sigma}$ does not realize $\mathcal{F}_{\mathsf{AdaptSig}}$, i.e., there exists an environment $\mathcal{E}$ that can differentiate whether it is interacting with $\mathcal{F}_{\mathsf{AdaptSig}}$ and $\mathcal{S}$ in the ideal world, or with $\Xi_2^{R,\Sigma}$ and $\mathcal{A}$ in the real world. We show that $\Xi_2^{R,\Sigma}$ violates the definition of secure two-party adaptor signature scheme from Appendix B. Since the environment $\mathcal{E}$ succeeds for any $\mathcal{S}$, it also succeeds for the following *generic* $\mathcal{S}$, which runs a simulated copy of $\mathcal{A}$ and does the following (where $b \in \{0, 1\}$ defines which of the two parties is corrupted):

(1) Any input from $\mathcal{E}$ is forwarded to $\mathcal{A}$, and any output from $\mathcal{A}$ is copied to $\mathcal{S}$'s output (to be read by $\mathcal{E}$).
(2) Upon receiving a message (keygen, sid) from $\mathcal{F}^{R,f_{adapt}}_{\mathsf{AdaptSig}}$:

[4]Parameterizing the protocol with a deterministic adaptation function f_{adapt} is without loss of generality, since the generic transformation of Erwig et al. [18, Figure 7] considers that the adaptation algorithm of two-party adaptor signature coincides with the function f_{adapt}.

- If sid is not of the form $(P_{1-b}, P_b, \mathsf{sid}')$, then $\mathcal{S}$ ignores the request.
- Else, $\mathcal{S}$ runs $(\mathsf{sk}_{1-b}, \mathsf{pk}_{1-b}) \leftarrow \Sigma_2.\mathsf{KGen}(\mathsf{pp})$, sets $v := (\mathsf{pk}_{1-b}, \mathsf{pk}_b)$, records the tuple $(\mathsf{sid}, \mathsf{sk}_{1-b}, v)$, and returns (verification−key, sid, v) to $\mathcal{F}^{R,f_{adapt}}_{\mathsf{AdaptSig}}$.

(3) Upon receiving a message (sign, sid, m, v, Y, type) from $\mathcal{F}^{R,f_{adapt}}_{\mathsf{AdaptSig}}$:
- If sid is not of the form $(P_{1-b}, P_b, \mathsf{sid}')$ and no tuple of the form $(\mathsf{sid}, \mathsf{sk}_{1-b}, v)$, has been previously recorded, then $\mathcal{S}$ ignores the request.
- If type = signature, then $\mathcal{S}$ parses $v := (\mathsf{pk}_{1-b}, \mathsf{pk}_b)$, simulates a run of the protocol $\sigma \leftarrow \Sigma_2.\Pi^{\mathcal{A}}_{\mathsf{Sig}\langle \mathsf{sk}_{1-b}, \cdot \rangle}(\mathsf{pk}_{1-b}, \mathsf{pk}_b, m)$ (by simulating the honest party P_{1-b}), and sends (signature, sid, m, σ) to $\mathcal{F}^{R,f_{adapt}}_{\mathsf{AdaptSig}}$.
- If type = pre−signature, then $\mathcal{S}$ parses $Y := Y$ and $v := (\mathsf{pk}_{1-b}, \mathsf{pk}_b)$, simulates a run of the protocol $\hat{\sigma} \leftarrow \Xi_2^{R,\Sigma}.\Pi^{\mathcal{A}}_{\mathsf{PreSig}\langle \mathsf{sk}_{1-b}, \cdot \rangle}(\mathsf{pk}_{1-b}, \mathsf{pk}_b, m, Y)$ (by simulating the honest party P_{1-b}), and sends (signature, sid, m, $\hat{\sigma}$) to $\mathcal{F}^{R,f_{adapt}}_{\mathsf{AdaptSig}}$.

(4) Upon receiving (verify, sid, m, σ, v, Y, type) from $\mathcal{F}^{R,f_{adapt}}_{\mathsf{AdaptSig}}$:
- If type = signature, then $\mathcal{S}$ parses $v := (\mathsf{pk}_{1-b}, \mathsf{pk}_b)$, computes $\mathsf{apk} := \Sigma_2.\mathsf{KAgg}(\mathsf{pk}_{1-b}, \mathsf{pk}_b)$, sets $\phi := \Sigma_2.\mathsf{Vf}(\mathsf{apk}, m, \sigma)$, and returns (verified, sid, m, ϕ) to $\mathcal{F}^{R,f_{adapt}}_{\mathsf{AdaptSig}}$.
- If type = pre−signature, then $\mathcal{S}$ parses $Y := Y$ and $v := (\mathsf{pk}_{1-b}, \mathsf{pk}_b)$, computes $\mathsf{apk} := \Sigma_2.\mathsf{KAgg}(\mathsf{pk}_{1-b}, \mathsf{pk}_b)$, sets $\phi := \Xi_2^{R,\Sigma}.\mathsf{PreVf}(\mathsf{apk}, m, Y, \sigma)$, and returns (verified, sid, m, ϕ) to $\mathcal{F}^{R,f_{adapt}}_{\mathsf{AdaptSig}}$.

(5) When $\mathcal{A}$ corrupts some party P, then $\mathcal{S}$ corrupts P in the ideal world. If P is the signer, then $\mathcal{S}$ reveals the signing key sk (and the internal state of the signing algorithm, if such a state exists) as P's internal state.

Next, we assume that $\Xi_2^{R,\Sigma}$ is both complete and consistent, and construct an attacker $\mathcal{B}$ that breaks the unforgeability. $\mathcal{B}$ runs a simulated copy of $\mathcal{E}$ and simulates the interactions with $\mathcal{S}$ and $\mathcal{F}_{\mathsf{AdaptSig}}$. Analogous to $\mathcal{S}$, $\mathcal{B}$ also runs a simulated copy of $\mathcal{A}$. However, in the first activation, instead of running $\Sigma_2.\mathsf{KGen}$ to generate the key, $\mathcal{B}$ gives to $\mathcal{A}$ the verification key pk_{1-b} that it has received as an input from its own challenger (where $b \in \{0, 1\}$ defines which of the two parties is corrupted). Whenever $\mathcal{B}$ needs to provide (pre-)signature on a message m, $\mathcal{B}$ asks its oracles to (pre-)sign m and obtains (pre-)signature σ. Moreover, $\mathcal{B}$ and $\mathcal{A}$ jointly generate pre-signature $\hat{\sigma}$ on the challenge message m^* (provided by $\mathcal{A}$), where $\mathcal{B}$ just relays the protocol messages of its own challenger when computing the joint pre-signature on the same challenge message m^*. Finally, whenever the simulated $\mathcal{E}$ activates some party with input (verify, sid, m^*, σ^*, v, Y, type), where type = signature, $\mathcal{B}$ checks whether (m^*, σ^*) constitutes a valid forgery (i.e., m^* is the challenge message and it has never been signed before and $\Sigma_2.\mathsf{Vf}(\mathsf{apk}, m^*, \sigma^*) = 1$, for $\mathsf{apk} := \Sigma_2.\mathsf{KAgg}(\mathsf{pk}_{1-b}, \mathsf{pk}_b)$, where $v := (\mathsf{pk}_{1-b}, \mathsf{pk}_b)$ and pk_b is the verification key of $\mathcal{A}$). If (m^*, σ^*) is a valid forgery, then $\mathcal{B}$ outputs σ^* as its own forgery and halts. Otherwise, it continues the simulation. If $\mathcal{A}$ asks to corrupt the honest signer P_{1-b}, then $\mathcal{B}$ halts with a failure.

We analyze the success probability of $\mathcal{B}$. Let win denote the event that in a run of $\Xi_2^{R,\Sigma}$ with $\mathcal{A}$ and $\mathcal{E}$ with $\mathsf{sid} = (P_{1-b}, P_b, \mathsf{sid}')$, the signers generate the verification keys pk_{1-b} and pk_b, such that $v := (\mathsf{pk}_{1-b}, \mathsf{pk}_b)$, some party is activated with verification request (verify, sid, m^*, σ^*, v, Y, type), where type = signature and $\Sigma_2.\mathsf{Vf}(v, m^*, \sigma^*) = 1$, party P_{1-b} is uncorrupted and signers never signed m^*. Since $\Xi_2^{R,\Sigma}$ is complete and consistent, we have that as long as the event win does not occur $\mathcal{E}$'s view of an interaction with $\mathcal{A}$ and $\Xi_2^{R,\Sigma}$ in the real world is statistically close to its view of an interaction with $\mathcal{S}$ and $\mathcal{F}^{R,f_{adapt}}_{\mathsf{AdaptSig}}$ in the ideal world. However, we are given that $\mathcal{E}$ distinguishes with non-negligible probability between the ideal and real world, thus, we are guaranteed that when $\mathcal{E}$ interacts with $\mathcal{A}$ and $\Xi_2^{R,\Sigma}$, event win occurs with non-negligible probability. Finally, observe that from the point of view of $\mathcal{A}$ and $\mathcal{E}$, the interaction with the forger $\mathcal{B}$ looks the same as an interaction in the real world with $\Xi_2^{R,\Sigma}$. Hence, we are guaranteed that event win occurs with non-negligible probability. Furthermore, event win can only occur when P_{1-b} is not corrupted. This means whenever event win occurs, $\mathcal{B}$ outputs a successful forgery, which contradicts the unforgeability definition of $\Xi_2^{R,\Sigma}$.

We can construct a similar adversary $\mathcal{B}'$ against the witness extractability property of $\Xi_2^{R,\Sigma}$. $\mathcal{B}'$ works exactly as $\mathcal{B}$ above, with the caveat that the joint pre-signature $\hat{\sigma}$ is computed over both the challenge message m^* and challenge statement Y^* provided by the adversary $\mathcal{A}$. Moreover, the winning condition is adjusted such that m^* is the challenge message and it has never been signed before and $\Sigma_2.\mathsf{Vf}(\mathsf{apk}, m^*, \sigma^*) = 1$, for $\mathsf{apk} := \Sigma_2.\mathsf{KAgg}(\mathsf{pk}_{1-b}, \mathsf{pk}_b)$, where $v := (\mathsf{pk}_{1-b}, \mathsf{pk}_b)$ and pk_b is the verification key of $\mathcal{A}$, and $(Y^*, y') \notin R$, for $y' := \Xi_2^{R,\Sigma}.\mathsf{Ext}(v, \sigma^*, \hat{\sigma}, Y^*)$.

Lastly, we observe that pre-signature adaptability is captured within the adaptation interface of the ideal functionality $\mathcal{F}_{\mathsf{AdaptSig}}$, which makes use of the global ideal functionality $\mathcal{G}_{\mathsf{Cond}}$ and parameterized deterministic adaptation function f_{adapt}. More precisely, a valid pre-signature $\hat{\sigma}$ w.r.t. some condition $Y := Y$ can be adapted into a valid signature, i.e., $\sigma := f_{adapt}(\hat{\sigma}, y)$, using the witness y that satisfies $(Y, y) \in R$.

This concludes the proof of Theorem 2. □

E.3 Security Analysis of Private Adaptor Signature

We recall and prove the lemma stated in Appendix C.1.

Lemma 1. *A two-party adaptor signature scheme with aggregatable public keys (from identification scheme) $\Xi_2^{R,\Sigma}$ is perfectly unlinkable.*

Proof. We prove this lemma by considering the deterministic adaptation function f_{adapt} given in Appendix B for the generic transformation from an identification scheme-based two-party signature to an adaptor signature. $f_{adapt}\colon \mathcal{D}_{\mathsf{resp}} \times \mathcal{D}_{\mathsf{w}} \to \mathcal{D}_{\mathsf{resp}}$, takes as input a pre-signature value $\hat{s} \in \mathcal{D}_{\mathsf{resp}}$ (which corresponds to the response value of the identification scheme) and a witness $y \in \mathcal{D}_{\mathsf{w}}$ of the hard relation R, and outputs a new value $s \in \mathcal{D}_{\mathsf{resp}}$. Then, the adaptation function of $\Xi_2^{R,\Sigma}$ is defined as $\mathsf{Adapt}(\mathsf{apk}, \hat{s}, y) := f_{adapt}(\hat{s}, y)$. Since a freshly computed signature from an identification scheme is some response value $s' \in \mathcal{D}_{\text{resp}}$, it is immediate that the adapted signature s and the fresh signature s' come from the same distribution $\mathcal{D}_{\text{resp}}$. □

E.4 Security Analysis of Lock-enabling Ledger

We prove the following theorem about the security of lock-enabling ledger, which was previously stated in §7.

Theorem 3. *The protocol* $\Pi_{\text{LedgerLocks}}$ *UC-realizes* $\mathcal{G}_{\text{LedgerLocks}}$, *in the* $(\mathcal{F}_{\text{AdaptSig}}, \mathcal{G}_{\text{Ledger}})$*-hybrid model.*

Proof. The only non-trivial properties that $\mathcal{G}_{\text{LedgerLocks}}$ enforces (in addition to what the base ledger functionality $\mathcal{G}_{\text{Ledger}}$ provides) are that only the account holders can submit transactions and transactions can be tied to conditions. Since both of these properties are achieved through the usage of adaptor signature functionality $\mathcal{F}_{\text{AdaptSig}}$, we have that the real world indeed implements the stronger validation predicate. More precisely, due to the security of $\mathcal{F}_{\text{AdaptSig}}$, we are guaranteed existence of the simulator $\mathcal{S}_{\text{AdaptSig}}$, which can handle our calls in ideal world execution to perfectly simulate the protocol. Our simulator $\mathcal{S}_{\text{LedgerLocks}}$ is given below. We note that indistinguishability follows because the simulator $\mathcal{S}_{\text{LedgerCond}}$ makes exactly the same calls to $\mathcal{F}_{\text{AdaptSig}}$ that an honest party makes in $\Pi_{\text{LedgerLocks}}$. Furthermore, in the case of releasing transactions, we have that the simulator $\mathcal{S}_{\text{LedgerCond}}$ learns the witness as long as it is involved in the transaction, which coincides with the real world protocol. This concludes the proof.

Simulator $\mathcal{S}_{\text{LedgerLocks}}$

Initialization: The simulator internally runs $\mathcal{A}$ in a black-box way and simulates the interaction between $\mathcal{A}$ and (emulated) real-world hybrid functionalities. The inputs from $\mathcal{A}$ to the base ledger $\mathcal{G}_{\text{Ledger}}$ are simply relayed (and replies given back to $\mathcal{A}$). The simulator maintains locally a list of keys $\mathcal{K}_P$, list of pre-signed transactions $\mathcal{P}_P$ and list of signed transactions Q_P, for an honest party P. Moreover, the simulator manages internally a simulated adaptor signature functionality $\mathcal{F}_{\text{AdaptSig}}$.

Messages from Lock-enabling Ledger:

- Upon receiving (account-req, sid, (P', P)) from $\mathcal{G}_{\text{LedgerLocks}}$, set sid′ := (sid, P, P'), forward (keygen, sid′) to the simulated adaptor signature functionality $\mathcal{F}_{\text{AdaptSig}}$ in the name of P. Upon receiving (verification-key, sid′, vk) from $\mathcal{F}_{\text{AdaptSig}}$, output this to $\mathcal{A}$ and store (P', vk) in $\mathcal{K}_P$.
- Upon receiving (auth-req, sid, tx, α) from $\mathcal{G}_{\text{LedgerLocks}}$, parse α as (AccountId, $\{P^*\}$) and $\{P^*\}$ as (P, P'), set sid′ := (sid, P, P'), and forward (sign, sid′, tx, vk, $\perp$, *signature*) to the simulated adaptor signature functionality $\mathcal{F}_{\text{AdaptSig}}$ in the name of P. Upon receiving (signature, sid′, σ) from $\mathcal{F}_{\text{AdaptSig}}$, output this answer to $\mathcal{A}$ and store (tx, vk, σ) in list Q_P.
- Upon receiving (lock-req, sid, tx, α, Y) from the $\mathcal{G}_{\text{LedgerLocks}}$, parse α as (AccountId, $\{P^*\}$) and $\{P^*\}$ as (P, P'), set sid′ := (sid, P, P'), forward (sign, sid′, tx, vk, Y, *pre-signature*) to the simulated adaptor signature functionality $\mathcal{F}_{\text{AdaptSig}}$ in the name of P. Upon receiving (signature, sid′, $\hat{\sigma}$) from $\mathcal{F}_{\text{AdaptSig}}$, output this answer to $\mathcal{A}$ and store (tx, vk, $Y, \hat{\sigma}$) in list $\mathcal{P}_P$.
- Upon receiving (release-tx, sid, y) from $\mathcal{G}_{\text{LedgerLocks}}$, store y.

□

F LEDGERLOCKS APPLICATION: ATOMIC SWAPS PROTOCOL DESCRIPTION

In this section, we use the LedgerLocks framework to describe the atomic swaps protocol. The protocol description is included in Figures 21 and 22.

G LEDGERLOCKS APPLICATION: MULTI-HOP PAYMENT PROTOCOL

In this section, we first use the LedgerLocks framework to describe the payment channel protocol (as given in [1]). The protocol description is included in Figures 25 to 29. We then describe a multi-hop payment over payment channels (as given in [31]). The protocol description is included in Figures 30 to 33.

For modeling these protocols we will instantiate $\mathcal{G}_{\text{LedgerLocks}}$ to support simple UTXO style transactions and will instantiate the CheckBase predicate accordingly. To this end, we first fix the transaction format. Recall that $\mathcal{G}_{\text{LedgerLocks}}$ transactions are of the form $(\mathbb{A}, \texttt{tx}')$ where $\mathbb{A}$ denotes a set of account identifiers. In §8, we already showed how to refine CheckBase to account for absolute (transaction-level) timelocks by fixing the transaction format of $\texttt{tx}'$ to $(\texttt{tx}'', tl)$.

We will further refine the format of $\texttt{tx}''$ to be of the form $(id, \vec{in}, \vec{out})$ with id being a transaction identifier, $\vec{in}$ being a vector of inputs and $\vec{out}$ being a vector of outputs. Inputs $in_i \in \vec{in}$ are of the form (id_{out}, j, rtl) where (id_{out}, j) refers to the output that the input is spending (with id_{out} being the transaction id and j the offset in the transactions output vector), and rtl denotes a *relative timelock* indicating the number of blocks that need to have been included in the blockchain since the publication of the transaction with the referenced out before the transaction can be published. Outputs are $out_i \in \vec{out}$ and they are of the form (aID, v) with aID denoting the id of the account controlling the output and v denoting the output's value.

The CheckBase predicate was already refined in §8 to account for the (absolute) timelock check. We will now refine the $\text{CheckBase}^{\mathbb{C}}$ predicate to account for the additional UTXO checks.

Intuitively, the following checks need to be conducted:

(1) The transaction id is fresh
(2) The transaction inputs are unique
(3) The transactions should be required to be authorized by all accounts that control consumed inputs
(4) The values of the outputs created by a transaction should not exceed the values of the inputs consumed
(5) All consumed inputs should exist and respect the relative input timelocks
(6) All consumed inputs should not yet have been consumed by another transaction on the blockchain (no double-spending)

To simplify these checks, we define a helper function getOutput that accesses information of a transaction input in the blockchain state. Given an input in and the blockchain state, getOutput returns a set containing tuples with additional information for that output, namely the aID of the account controlling the spent output, the value v of the output and the height h at which the transaction with the output was added to the state.

Protocol $\Pi^R_{\text{AtomicSwap}}$

Alice(aID_A, Y_O)		Bob(aID_B, Y_O)
Sample $(Y, y) \leftarrow \mathsf{GenR}(1^\lambda)$		
Set $Y^* := (Y, Y_O)$		
Invoke $\mathcal{G}^{R,f_{merge}}_{\text{Cond}}[Y_O]$ on input (create-ind-cond, sid, (Y, y))		
Invoke $\mathcal{G}^{R,f_{merge}}_{\text{Cond}}[Y_O]$ on input (create-merged-cond, sid, (Y, Y_O))		
Receive (create-merged-cond, sid, Y^*) from $\mathcal{G}^{R,f_{merge}}_{\text{Cond}}[Y_O]$		
	$\xrightarrow{(Y, Y^*)}$	
		If $Y^* \neq f_{merge}(stmt, R, (Y, Y_O))$ then abort
Invoke $\mathcal{G}^A_{\text{LedgerLocks}}$ on input (create-account, sid, B)		Invoke $\mathcal{G}^B_{\text{LedgerLocks}}$ on input (create-account, sid, A)
Receive (acc-req, sid, (A, B)) from $\mathcal{G}^B_{\text{LedgerLocks}}$		Receive (acc-req, sid, (B, A)) from $\mathcal{G}^A_{\text{LedgerLocks}}$
Send (acc-rep, sid, $b := 1$) to $\mathcal{G}^B_{\text{LedgerLocks}}$		Send (acc-rep, sid, $b := 1$) to $\mathcal{G}^A_{\text{LedgerLocks}}$
Receive (create-account, sid, aID^A_{AB}) from $\mathcal{G}^A_{\text{LedgerLocks}}$		Receive (create-account, sid, aID^A_{AB}) from $\mathcal{G}^A_{\text{LedgerLocks}}$
Receive (create-account, sid, aID^B_{AB}) from $\mathcal{G}^B_{\text{LedgerLocks}}$		Receive (create-account, sid, aID^B_{AB}) from $\mathcal{G}^B_{\text{LedgerLocks}}$
Set $\mathtt{dtx}_A := (\mathtt{dtx}_A[aID_A], 0)$		Set $\mathtt{dtx}_B := (\mathtt{dtx}_B[aID_B], 0)$
Set $\mathtt{ctx}_B := (\mathtt{ctx}_B[\mathtt{dtx}_A], 0)$		Set $\mathtt{ctx}_A := (\mathtt{ctx}_A[\mathtt{dtx}_B], 0)$
Set $\mathtt{rtx}_A := (\mathtt{rtx}_A[\mathtt{dtx}_A], h_A)$		Set $\mathtt{rtx}_B := (\mathtt{rtx}_B[\mathtt{dtx}_B], h_B)$
	$\xrightarrow{\mathtt{dtx}_A, \mathtt{ctx}_B, \mathtt{rtx}_A}$	
	$\xleftarrow{\mathtt{dtx}_B, \mathtt{ctx}_A, \mathtt{rtx}_B}$	
Invoke $\mathcal{G}^A_{\text{LedgerLocks}}$ and $\mathcal{G}^B_{\text{LedgerLocks}}$ with (read, sid)		Invoke $\mathcal{G}^A_{\text{LedgerLocks}}$ and $\mathcal{G}^B_{\text{LedgerLocks}}$ with (read, sid)
Receive (read, sid, state^A) and (read, sid, state^B)		Receive (read, sid, state^A) and (read, sid, state^B)
If safe$(h_A, h_B, \text{state}^A, \text{state}^B) \neq 1$ then abort		If safe$(h_A, h_B, \text{state}^A, \text{state}^B) \neq 1$ then abort
Invoke $\mathcal{G}^A_{\text{LedgerLocks}}$ with (auth-tx, sid, $\mathtt{rtx}_A$, aID^A_{AB})		Invoke $\mathcal{G}^B_{\text{LedgerLocks}}$ with (auth-tx, sid, $\mathtt{rtx}_B$, aID^B_{AB})
Receive (auth-req, sid, $\mathtt{rtx}_B$, aID^B_{AB}) from $\mathcal{G}^B_{\text{LedgerLocks}}$		Receive (auth-req, sid, $\mathtt{rtx}_A$, aID^A_{AB}) from $\mathcal{G}^A_{\text{LedgerLocks}}$
Send (auth-rep, sid, $b^B_r := 1$, aID^B_{AB}) to $\mathcal{G}^B_{\text{LedgerLocks}}$		Send (auth-rep, sid, $b^A_r := 1$, aID^A_{AB}) to $\mathcal{G}^A_{\text{LedgerLocks}}$
Invoke $\mathcal{G}^A_{\text{LedgerLocks}}$ with (lock-tx, sid, $\mathtt{ctx}_B$, aID^A_{AB}, Y^*)		Invoke $\mathcal{G}^B_{\text{LedgerLocks}}$ with (lock-tx, sid, $\mathtt{ctx}_A$, aID^B_{AB}, Y^*)
Receive (lock-req, sid, $\mathtt{ctx}_A$, aID^B_{AB}, Y^*) from $\mathcal{G}^B_{\text{LedgerLocks}}$		Receive (lock-req, sid, $\mathtt{ctx}_B$, aID^A_{AB}, Y^*) from $\mathcal{G}^A_{\text{LedgerLocks}}$
Send (lock-rep, sid, $b^B_c := 1$, aID^B_{AB}) to $\mathcal{G}^B_{\text{LedgerLocks}}$		Send (lock-rep, sid, $b^A_c := 1$, aID^A_{AB}) to $\mathcal{G}^A_{\text{LedgerLocks}}$
Invoke $\mathcal{G}^A_{\text{LedgerLocks}}$ with (auth-tx, sid, $\mathtt{dtx}_A$, aID_A)		**while** $\neg$inState$(\mathtt{dtx}_A, \text{state}^A) \wedge \lvert\text{state}^B\rvert < h_B - 2 \cdot \#^B_{safe}$
Invoke $\mathcal{G}^A_{\text{LedgerLocks}}$ with (submit, sid, $\mathtt{dtx}_A$)		Invoke $\mathcal{G}^A_{\text{LedgerLocks}}$ and $\mathcal{G}^B_{\text{LedgerLocks}}$ with (read, sid)
		Receive (read, sid, state^A) and (read, sid, state^B)
while $\neg$inState$(\mathtt{dtx}_B, \text{state}^B) \wedge \lvert\text{state}^B\rvert < h_B - \#^B_{safe}$		Invoke $\mathcal{G}^B_{\text{LedgerLocks}}$ with (auth-tx, sid, $\mathtt{dtx}_B$, aID_B)
Invoke $\mathcal{G}^A_{\text{LedgerLocks}}$ and $\mathcal{G}^B_{\text{LedgerLocks}}$ with (read, sid)		Invoke $\mathcal{G}^B_{\text{LedgerLocks}}$ with (submit, sid, $\mathtt{dtx}_B$)
Receive (read, sid, state^A) and (read, sid, state^B)		

Figure 21: Setup protocol of atomic swap in $(\mathcal{G}^{R,f_{merge}}_{\text{Cond}}, \mathcal{G}_{\text{LedgerLocks}} := (\mathcal{G}^A_{\text{LedgerLocks}}, \mathcal{G}^B_{\text{LedgerLocks}}))$-hybrid world.

Note that getOutput should return either $\emptyset$ indicating that state does not contain a transaction with the referred output or a singleton set containing the information for the unique output in state.

$$\begin{aligned}
&\text{getOutput}((id_{out}, j, rtl), \text{state}) := \\
&\quad \{(id_{out}, j, aID, v, h) \mid \exists\, b\ \text{state}_{pre}\ \text{state}_{post}\ \vec{in}\ \vec{out}\ \mathbb{A}\ tl. \\
&\qquad \text{state} = \text{state}_{pre} \| b \| \text{state}_{post} \\
&\qquad \wedge\ h = |\text{state}_{pre}| \\
&\qquad \wedge\ (\mathbb{A}, ((id_{out}, \vec{in}, \vec{out}), tl) \in b \\
&\qquad \wedge\ out_j = (aID, v)\}
\end{aligned}$$

Protocol $\Pi^{R}_{\text{AtomicSwap}}$

Alice(aID_A, Y^*, y, y_O)	Bob(aID_B, Y^*)
$y^* \leftarrow f_{merge}(wit, R, (y, y_O))$	
Invoke $\mathcal{G}^B_{\text{LedgerLocks}}$ with $(\text{release-tx}, \text{sid}, \texttt{ctx}_A, aID^B_{AB}, Y^*, y^*)$	
Receive $(\text{release-tx}, \text{sid}, aID^B_{AB}, b)$ from $\mathcal{G}^A_{\text{LedgerLocks}}$	
If $b \neq 1$ then abort	**while** $\neg\text{inState}(\texttt{ctx}_A, \text{state}^B) \wedge \lvert\text{state}^B\rvert < h_B$
	Invoke $\mathcal{G}^B_{\text{LedgerLocks}}$ with $(\text{read}, \text{sid})$
	Receive $(\text{read}, \text{sid}, \text{state})$ from $\mathcal{G}^B_{\text{LedgerLocks}}$
	Invoke $\mathcal{G}^B_{\text{LedgerLocks}}$ with $(\text{signal-tx}, \text{sid}, aID^B_{AB}, \texttt{ctx}_A, Y^*)$
	Receive $(\text{signal-tx}, \text{sid}, y^*)$ from $\mathcal{G}^B_{\text{LedgerLocks}}$
	Invoke $\mathcal{G}^A_{\text{LedgerLocks}}$ with $(\text{release-tx}, \text{sid}, \texttt{ctx}_B, aID^A_{AB}, Y^*, y^*)$
	Receive $(\text{release}, \text{sid}, aID^A_{AB}, b := 1)$ from $\mathcal{G}^A_{\text{LedgerLocks}}$

Figure 22: Atomic swap in $(\mathcal{G}^{R,f_{merge}}_{\text{Cond}}, \mathcal{G}_{\text{LedgerLocks}} := (\mathcal{G}^A_{\text{LedgerLocks}}, \mathcal{G}^B_{\text{LedgerLocks}}))$-hybrid world.

With the help of getOutput, we now define the CheckBase$^{\mathbb{C}}$ in Figure 23. The individual conditions of the functions correspond to the checks discussed before.

Timelocks. As already discussed in §2, one of the most delicate points for protocol security are the concrete timelocks of the refund transactions and the corresponding reaction times of the participants in the protocol. More precisely, the timelocks need to ensure that a user u_i can always claim the funds from user u_{i-1} that the next user u_{i+1} takes from them. To explain how the timelocks need to be set to ensure this, we consider the following worst-case scenario: Consider that user u_{i+1} is not responding to user u_i so that u_i at time $t[i]$ (the timelock of its refund transaction $\texttt{rtx}^i$ for the money locked on the channel with u_{i+1}), u_i wants to submit $\texttt{rtx}^i$. Then u_i needs to consider that $\texttt{rtx}^i$ can only be published once the channel with u_{i+1} has been closed, meaning that both the commit transaction $\texttt{tx}^i_c$ and the split transaction $\texttt{tx}^i_s$ of this channel must have been published. Since publishing $\texttt{tx}^i_c$ takes up to $\#_{safe}$ (from the perspective of u_i) and after that (due to its relative timelock) $\texttt{tx}^i_s$ can only be submitted after additional $\#_{safe}$ blocks and may take $\#_{safe}$ again till being published, u_i needs to start closing the channel at least at block height $t[i] - 3 \cdot \#_{safe}$ to be sure that $\texttt{tx}^i_s$ will be published at $t[i]$ so that u_i can submit $\texttt{rtx}^i$.

Now, at this point, u_i cannot be sure that $\texttt{rtx}^i$ will also be published on the blockchain since $\texttt{rtx}^i$ can still be outrun by $\texttt{ctx}^i$ (published by u_{i+1}). However, u_i is guaranteed that by $t[i] + \#_{safe}$ either $\texttt{rtx}^i$ or $\texttt{ctx}^i$ will be included in the ledger.

If indeed $\texttt{ctx}^i$ was published, u_i still needs to have sufficient time to claim the payment from u_{i-1}. In the optimistic case, this can be settled by an offchain channel update. However, if u_{i-1} does not collaborate, u_i also needs to close the channel with u_{i-1} (taking up to $3 \cdot \#_{safe}$ blocks) and afterwards publish $\texttt{ctx}^{i-1}$ for claiming the money locked with u_{i-1} on this channel (taking other $\#_{safe}$ blocks). Consequently, it may take until $5 \cdot \#_{safe}$ blocks until u_i claims their funds in this way. To ensure that $\texttt{ctx}^{i-1}$ is guaranteed to be published, the timelock $t[i-1]$ of $\texttt{rtx}^{i-1}$ needs to prevent that u_{i-1} could publish $\texttt{rtx}^{i-1}$ before (and in this way outrun $\texttt{ctx}^{i-1}$). For this reason, the users check during the setup protocol (Figure 30) that $t[i-1] > t[i] + 5 \cdot \#_{safe}$. Further, they ensure that during the payment protocol (Figure 33), they publish the claim transaction $\texttt{ctx}^{i-1}$ at least $4 \cdot \#_{safe}$ before the timelock $t[i-1]$ (so that it will be included before $\texttt{rtx}^i$ is enabled).

Note that it is also crucial for security that an honest user u_i starts closing the channel with u_{i+1} (if that user does not collaborate) latest at $t[i] - 3 \cdot \#_{safe}$ to make sure that at latest at $t[i] + \#_{safe}$ u_i knows whether they need to initiate the forceful claiming. If this would be learned only later, the difference between the timelocks may not be sufficient to ensure a secure execution.

$$\begin{aligned}
&\mathsf{CheckBase}^{\mathbb{C}}((\mathbb{A}, (id, \vec{in}, \vec{out})), \mathsf{state}) := \\
&\quad \left(\neg\exists \mathtt{tx}\ \mathbb{A}'\ id'\ \vec{in}'\ \vec{out}'\ tl'.\ \mathtt{tx} = (\mathbb{A}', ((id', \vec{in}', \vec{out}'), tl')) \ \wedge\ \mathsf{inState}(\mathtt{tx}, \mathsf{state}) \ \wedge\ id = id'\right) && (1)\\
&\quad \wedge\ \left(\forall (id_{out}, j, rtl)\ (id'_{out}, j', rtl') \in \vec{in}.\ (id_{out}, j) \neq (id'_{out}, j')\right) && (2)\\
&\quad \wedge\ \left(\mathbb{A} = \left\{ aID \mid (id, j, aID, v, h) \in \bigcup_{in \in \vec{in}} \mathsf{getOutput}(in, \mathsf{state}) \right\}\right) && (3)\\
&\quad \wedge\ \left(\sum_{(aID, v) \in \vec{out}} v \leq \sum_{(id', j', aID', v', h') \in \bigcup_{in \in \vec{in}} \mathsf{getOutput}(in, \mathsf{state})} v' \right) && (4)\\
&\quad \wedge\ \Big(\forall (id_{out}, j, rtl) \in \vec{in}.\ \exists\ aID'\ v'\ h'.\ \mathsf{getOutput}((id_{out}, j, rtl), \mathsf{state}) = \{(id_{out}, j, aID', v', h')\} \\
&\qquad \wedge\ |\mathsf{state}| - h \geq rtl\Big) && (5)\\
&\quad \wedge\ \Big(\forall (id_{out}, j, rtl) \in \vec{in}. \neg\exists \mathtt{tx}\ \mathbb{A}'\ id'\ \vec{in}'\ \vec{out}'\ tl'.\ \mathtt{tx} = (\mathbb{A}', ((id', \vec{in}', \vec{out}'), tl'))\ \wedge\ \mathsf{inState}(\mathtt{tx}, \mathsf{state}) \\
&\qquad \wedge\ \exists (id'_{out}, j', rtl') \in \vec{in}'.\ (id'_{out}, j') = (id_{out}, j)\Big) && (6)
\end{aligned}$$

Figure 23: Definition of the CheckBase$^{\mathbb{C}}$ predicate.

```
GenFund($\mathtt{tx}_A, aID_A, \mathtt{tx}_B, aID_B, aID_{AB}$)
    parse $\mathtt{tx}_A$ as $(\mathbb{A}_A, ((id_A, \vec{in}_A, [(aID_A, v_A)], tl_A))$
    parse $\mathtt{tx}_B$ as $(\mathbb{A}_B, ((id_B, \vec{in}_B, [(aID_B, v_B)], tl_B))$
    $id^* \leftarrow H(id_A \| id_B)$
    return $((\{aID_A, aID_B\}, ((id^*, [(id_A, 0, 0), (id_B, 0, 0)], [(aID_{AB}, v_A + v_B)]), 0)), (v_A, v_B))$
```

```
GenCommit($\mathtt{tx}_f, aID_{AB}$)
    parse $\mathtt{tx}_f$ as $(\mathbb{A}, ((id, \vec{in}, [(aID_{AB}, v)], tl'))$
    $id^* \leftarrow H(id)$
    return $(\{aID_{AB}\}, ((id^*, [(id, 0, 0)], [(aID_{AB}, v)]), 0))$
```

```
GenSplit($\mathtt{tx}_c, \vec{d}$)
    parse $\mathtt{tx}_c$ as $(\mathbb{A}, ((id, \vec{in}, [(aID_{AB}, v)], tl'))$
    $id^* \leftarrow H(id)$
    return $(\{aID_{AB}\}, ((id^*, [(id, 0, \#_{safe})], \vec{d}), 0))$
```

```
GenPunish($\mathtt{tx}_c, aID_A$)
    parse $\mathtt{tx}_c$ as $(\mathbb{A}, ((id, \vec{in}, [(aID_{AB}, v)], tl'))$
    $id^* \leftarrow H(id)$
    return $(\{aID_{AB}\}, ((id^*, [(id, 0, 0)], [(aID_A, v)]), 0))$
```

```
GenPay($\mathtt{tx}_s, aID_{AB}, aID, tl$)
    parse $\mathtt{tx}_s$ as $(\mathbb{A}, ((id, \vec{in}, [(aID_{AB}, v), out_A, out_B], tl'))$
    $id^* \leftarrow H(id)$
    return $(\{aID_{AB}\}, ((id^*, [(id, 0, 0)], [(aID, v)]), tl))$
```

```
ComputeBalance($\mathtt{tx}_f, \mathtt{tx}_s$)
    parse $\mathtt{tx}_f$ as $(\mathbb{A}_f, ((id_f, \vec{in}_f, \vec{out}_f, tl_f))$
    parse $\mathtt{tx}_s$ as $(\mathbb{A}_s, ((id_s, \vec{in}_s, \vec{out}_s), tl_s))$
    $id^* \leftarrow H(id_f)$
    return $(\{aID_{AB}\}, ((id^*, [(id_f, 0, 0)], \vec{out}_s), 0))$
```

Figure 24: Definition of transaction constructors used in the channel and multi-hop payment protocol.

Protocol Π_{Channel}

Create Channel

Alice($\mathtt{tx}_A$, aID_A)		Bob($\mathtt{tx}_B$, aID_B)
Sample $(Y_P^A, y_P^A) \leftarrow \mathsf{GenR}(1^\lambda)$		Sample $(Y_P^B, y_P^B) \leftarrow \mathsf{GenR}(1^\lambda)$
Sample $(Y_R^A, y_R^A) \leftarrow \mathsf{GenR}(1^\lambda)$		Sample $(Y_R^B, y_R^B) \leftarrow \mathsf{GenR}(1^\lambda)$
	$\xrightarrow{(Y_P^A, Y_R^A)}$	
	$\xleftarrow{(Y_P^B, Y_R^B)}$	
Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$ on input (create-account, sid, B)		
		Receive (acc-req, sid, (B, A)) from $\mathcal{G}_{\mathsf{LedgerLocks}}$
		Send (acc-rep, sid, $b := 1$) to $\mathcal{G}_{\mathsf{LedgerLocks}}$
Receive (create-account, sid, aID_{AB}) from $\mathcal{G}_{\mathsf{LedgerLocks}}$		Receive (create-account, sid, aID_{AB}) from $\mathcal{G}_{\mathsf{LedgerLocks}}$
Set $(\mathtt{tx}_f, (v_A, v_B)) := \mathit{GenFund}(\mathtt{tx}_A, aID_A, \mathtt{tx}_B, aID_B, aID_{AB})$		Set $(\mathtt{tx}_f, (v_A, v_B)) := \mathit{GenFund}(\mathtt{tx}_A, aID_A, \mathtt{tx}_B, aID_B, aID_{AB})$
Set $\mathtt{tx}_c := \mathit{GenCommit}(\mathtt{tx}_f, aID_{AB})$		Set $\mathtt{tx}_c := \mathit{GenCommit}(\mathtt{tx}_f, aID_{AB})$
Set $\mathtt{tx}_s := \mathit{GenSplit}(\mathtt{tx}_c, [(aID_A, v_A), (aID_B, v_B)])$		Set $\mathtt{tx}_s := \mathit{GenSplit}(\mathtt{tx}_c, [(aID_A, v_A), (aID_B, v_B)])$
Set $\mathtt{tx}_p^A := \mathit{GenPunish}(\mathtt{tx}_c, aID_A)$		Set $\mathtt{tx}_p^B := \mathit{GenPunish}(\mathtt{tx}_c, aID_B)$
Invoke $\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}$ on input (create-ind-cond, sid, (Y_P^A, y_P^A))		Invoke $\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}$ on input (create-ind-cond, sid, (Y_P^B, y_P^B))
Receive (created-ind-cond, sid, Y_P^A) from $\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}$		Receive (created-ind-cond, sid, Y_P^B) from $\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}$
Invoke $\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}$ on input (create-ind-cond, sid, (Y_R^A, y_R^A))		Invoke $\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}$ on input (create-ind-cond, sid, (Y_R^B, y_R^B))
Receive (created-ind-cond, sid, Y_R^A) from $\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}$		Receive (created-ind-cond, sid, Y_R^B) from $\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}$
Invoke $\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}$ on input (create-merged-cond, sid, (Y_P^B, Y_R^B))		Invoke $\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}$ on input (create-merged-cond, sid, (Y_P^A, Y_R^A))
Receive (create-merged-cond, sid, Y^{B*}) from $\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}$		Receive (create-merged-cond, sid, Y^{A*}) from $\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}$
Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$ with (lock-tx, sid, $\mathtt{tx}_c$, aID_{AB}, Y_P^A)		Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$ with (lock-tx, sid, $\mathtt{tx}_c$, aID_{AB}, Y_P^B)
Receive (lock-req, sid, $\mathtt{tx}_c$, $(aID_{AB}, (A, B))$) from $\mathcal{G}_{\mathsf{LedgerLocks}}$		Receive (lock-req, sid, $\mathtt{tx}_c$, $(aID_{AB}, (A, B))$) from $\mathcal{G}_{\mathsf{LedgerLocks}}$
Send (lock-rep, sid, $b_c^A := 1$) to $\mathcal{G}_{\mathsf{LedgerLocks}}$		Send (lock-rep, sid, $b_c^B := 1$) to $\mathcal{G}_{\mathsf{LedgerLocks}}$
Receive (lock-tx, sid, b_c^{AB}) from $\mathcal{G}_{\mathsf{LedgerLocks}}$		Receive (lock-tx, sid, b_c^{AB}) from $\mathcal{G}_{\mathsf{LedgerLocks}}$
Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$ with (auth-tx, sid, $\mathtt{tx}_s$, aID_{AB})		
		Receive (auth-req, sid, $\mathtt{tx}_s$, $(aID_{AB}, (A, B))$) from $\mathcal{G}_{\mathsf{LedgerLocks}}$
		Send (auth-rep, sid, $b_s^B := 1$, aID_{AB}) to $\mathcal{G}_{\mathsf{LedgerLocks}}$
Receive (auth-tx, sid, b_s^B) from $\mathcal{G}_{\mathsf{LedgerLocks}}$		
Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$ with (lock-tx, sid, $\mathtt{tx}_p^A$, aID_{AB}, Y^{B*})		Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$ with (lock-tx, sid, $\mathtt{tx}_p^B$, aID_{AB}, Y^{A*})
Receive (lock-req, sid, $\mathtt{tx}_p^B$, $(aID_{AB}, (A, B))$) from $\mathcal{G}_{\mathsf{LedgerLocks}}$		Receive (lock-req, sid, $\mathtt{tx}_p^A$, $(aID_{AB}, (A, B))$) from $\mathcal{G}_{\mathsf{LedgerLocks}}$
Send (lock-rep, sid, $b_p^A := 1$) to $\mathcal{G}_{\mathsf{LedgerLocks}}$		Send (lock-rep, sid, b_p^B) to $\mathcal{G}_{\mathsf{LedgerLocks}}$
Receive (lock-tx, sid, b_p^B) from $\mathcal{G}_{\mathsf{LedgerLocks}}$		Receive (lock-tx, sid, b_p^A) to $\mathcal{G}_{\mathsf{LedgerLocks}}$
If $b_s^B \wedge b_c^{AB} \wedge b_p^B \neq 1$ then abort		If $b_c^{AB} \wedge b_p^A \neq 1$ then abort
Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$ with (auth-tx, sid, $\mathtt{tx}_f$, aID_A)		Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$ with (auth-tx, sid, $\mathtt{tx}_f$, aID_B)
Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$with (read, sid)		Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$with (read, sid)
Receive (read, sid, state) from $\mathcal{G}_{\mathsf{LedgerLocks}}$		Receive (read, sid, state) from $\mathcal{G}_{\mathsf{LedgerLocks}}$
Set $h_f := \lvert\mathsf{state}\rvert$		Set $h_f := \lvert\mathsf{state}\rvert$
Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$ with (submit, sid, $\mathtt{tx}_f$)		Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$ with (submit, sid, $\mathtt{tx}_f$)
while$\lvert\mathsf{state}\rvert < h_f + \#_{safe}$:		**while**$\lvert\mathsf{state}\rvert < h_f + \#_{safe}$:
Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$with (read, sid)		Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$with (read, sid)
Receive (read, sid, state) from $\mathcal{G}_{\mathsf{LedgerLocks}}$		Receive (read, sid, state) from $\mathcal{G}_{\mathsf{LedgerLocks}}$
if inState($\mathtt{tx}_f$, state) :		**if** inState($\mathtt{tx}_f$, state) :
chState[0] := $(\mathtt{tx}_c, \mathtt{tx}_s, \mathtt{tx}_p^A, Y_P^A, y_P^A, Y_P^B, Y_R^A, y_R^A, Y_R^B)$		chState[0] := $(\mathtt{tx}_c, \mathtt{tx}_s, \mathtt{tx}_p^B, Y_P^B, y_P^B, Y_P^A, Y_R^B, y_R^B, Y_R^A)$
else		**else**
Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$ with (submit, sid, $\mathtt{tx}_r^A$)		Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$ with (submit, sid, $\mathtt{tx}_r^B$)
while$\lvert\mathsf{state}\rvert < h_f + 2 \cdot \#_{safe}$:		**while**$\lvert\mathsf{state}\rvert < h_f + 2 \cdot \#_{safe}$:
Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$with (read, sid)		Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$with (read, sid)
Receive (read, sid, state) from $\mathcal{G}_{\mathsf{LedgerLocks}}$		Receive (read, sid, state) from $\mathcal{G}_{\mathsf{LedgerLocks}}$
if inState($\mathtt{tx}_f$, state) :		**if** inState($\mathtt{tx}_f$, state) :
Set $bal := [(aID_A, v_A), (aID_B, v_B)]$		Set $bal := [(aID_A, v_A), (aID_B, v_B)]$
chState[0] := $(\mathtt{tx}_c, \mathtt{tx}_s, \mathtt{tx}_p^A, Y_P^A, y_P^A, Y_P^B, Y_R^A, y_R^A, Y_R^B, bal)$		chState[0] := $(\mathtt{tx}_c, \mathtt{tx}_s, \mathtt{tx}_p^B, Y_P^B, y_P^B, Y_P^A, Y_R^B, y_R^B, Y_R^A, bal)$

Figure 25: Create channel protocol in $(\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}, \mathcal{G}_{\mathsf{LedgerLocks}})$-hybrid world. Here, *GenFund*, *GenCommit*, *GenSplit*, and *GenPunish* denote the constructors for $\mathtt{tx}_f$, $\mathtt{tx}_c$, $\mathtt{tx}_s$, and $\mathtt{tx}_s$, respectively as described in Figure 24.

Protocol Π_{Channel}

Update Channel

Alice($\mathtt{tx}_f$, aID_A, *split-info*)		Bob($\mathtt{tx}_f$, aID_B, *split-info*)
Sample $(Y_P^A, y_P^A) \leftarrow \mathsf{GenR}(1^\lambda)$		Sample $(Y_P^B, y_P^B) \leftarrow \mathsf{GenR}(1^\lambda)$
Sample $(Y_R^A, y_R^A) \leftarrow \mathsf{GenR}(1^\lambda)$		Sample $(Y_R^B, y_R^B) \leftarrow \mathsf{GenR}(1^\lambda)$
	$\xrightarrow{(Y_P^A, Y_R^A)}$	
	$\xleftarrow{(Y_P^B, Y_R^B)}$	
Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$ on input (create-account, sid, B)		
		Receive (acc-req, (B, A)) from $\mathcal{G}_{\mathsf{LedgerLocks}}$
		Send (acc-rep, $b := 1$) to $\mathcal{G}_{\mathsf{LedgerLocks}}$
Receive (create-account, sid, aID_{AB}) from $\mathcal{G}_{\mathsf{LedgerLocks}}$		Receive (create-account, sid, aID_{AB}) from $\mathcal{G}_{\mathsf{LedgerLocks}}$
Set $\mathtt{tx}_c := \mathit{GenCommit}(\mathtt{tx}_f, aID_{AB})$		Set $\mathtt{tx}_c := \mathit{GenCommit}(\mathtt{tx}_f, aID_{AB})$
Set $\mathtt{tx}_s := \mathit{GenSplit}(\mathtt{tx}_c, \mathit{split\text{-}info})$		Set $\mathtt{tx}_s := \mathit{GenSplit}(\mathtt{tx}_c, \mathit{split\text{-}info})$
Set $\mathtt{tx}_p^A := \mathit{GenPunish}(\mathtt{tx}_c, aID_A)$		Set $\mathtt{tx}_p^B := \mathit{GenPunish}(\mathtt{tx}_c, aID_B)$
Invoke $\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}$ on input (create-ind-cond, sid, (Y_P^A, y_P^A))		Invoke $\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}$ on input (create-ind-cond, sid, (Y_P^B, y_P^B))
Receive (created-ind-cond, sid, Y_P^A) from $\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}$		Receive (created-ind-cond, sid, Y_P^B) from $\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}$
Invoke $\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}$ on input (create-ind-cond, sid, (Y_R^A, y_R^A))		Invoke $\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}$ on input (create-ind-cond, sid, (Y_R^B, y_R^B))
Receive (created-ind-cond, sid, Y_R^A) from $\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}$		Receive (created-ind-cond, sid, Y_R^B) from $\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}$
Invoke $\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}$ on input (create-merged-cond, sid, (Y_P^B, Y_R^B))		Invoke $\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}$ on input (create-merged-cond, sid, (Y_P^A, Y_R^A))
Receive (create-merged-cond, sid, Y^{B*}) from $\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}$		Receive (create-merged-cond, sid, Y^{A*}) from $\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}$
Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$ with (lock-tx, sid, $\mathtt{tx}_c$, aID_{AB}, Y_P^A)		Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$ with (lock-tx, sid, $\mathtt{tx}_c$, aID_{AB}, Y_P^B)
Receive (lock-req, sid, $\mathtt{tx}_c$, $(aID_{AB}, (A, B))$) from $\mathcal{G}_{\mathsf{LedgerLocks}}$		Receive (lock-req, sid, $\mathtt{tx}_c$, $(aID_{AB}, (A, B))$) from $\mathcal{G}_{\mathsf{LedgerLocks}}$
Send (lock-rep, sid, $b_c^A := 1$) to $\mathcal{G}_{\mathsf{LedgerLocks}}$		Send (lock-rep, sid, $b_c^B := 1$) to $\mathcal{G}_{\mathsf{LedgerLocks}}$
Receive (lock-tx, sid, b_c^{AB}) from $\mathcal{G}_{\mathsf{LedgerLocks}}$		Receive (lock-tx, sid, b_c^{AB}) from $\mathcal{G}_{\mathsf{LedgerLocks}}$
Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$ with (auth-tx, sid, $\mathtt{tx}_s$, aID_{AB})		
		Receive (auth-req, sid, $\mathtt{tx}_s$, $(aID_{AB}, (A, B))$) from $\mathcal{G}_{\mathsf{LedgerLocks}}$
		Send (auth-rep, sid, $b_s^B := 1$) to $\mathcal{G}_{\mathsf{LedgerLocks}}$
Receive (auth-tx, sid, b_s^B) from $\mathcal{G}_{\mathsf{LedgerLocks}}$		
Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$ with (lock-tx, sid, $\mathtt{tx}_p^A$, aID_{AB}, Y^{B*})		Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$ with (lock-tx, sid, $\mathtt{tx}_p^B$, aID_{AB}, Y^{A*})
Receive (lock-req, sid, $\mathtt{tx}_p^B$, $(aID_{AB}, (A, B))$) from $\mathcal{G}_{\mathsf{LedgerLocks}}$		Receive (lock-req, sid, $\mathtt{tx}_p^A$, $(aID_{AB}, (A, B))$) from $\mathcal{G}_{\mathsf{LedgerLocks}}$
Send (lock-rep, sid, $b_p^A := 1$) to $\mathcal{G}_{\mathsf{LedgerLocks}}$		Send (lock-rep, sid, $b_p^B := 1$) to $\mathcal{G}_{\mathsf{LedgerLocks}}$
Receive (lock-tx, sid, b_p^B) from $\mathcal{G}_{\mathsf{LedgerLocks}}$		Receive (lock-tx, sid, b_p^A) to $\mathcal{G}_{\mathsf{LedgerLocks}}$
If $b_s^B \wedge b_c^{AB} \wedge b_p^B \neq 1$ then abort		If $b_c^{AB} \wedge b_p^A \neq 1$ then abort
Read $(\overline{Y_R^A}, \overline{y_r^A})$ from $\mathsf{chState}[\lvert\mathsf{chState}\rvert - 1]$		Read $(\overline{Y_R^B}, \overline{y_r^B})$ from $\mathsf{chState}[\lvert\mathsf{chState}\rvert - 1]$
	$\xrightarrow{(\overline{Y_R^A}, \overline{y_r^A})}$	
	$\xleftarrow{(\overline{Y_R^B}, \overline{y_r^B})}$	
Invoke $\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}$ on input (open-cond, sid, $(\overline{Y_R^B}, \overline{y_R^B})$)		Invoke $\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}$ on input (open-cond, sid, $(\overline{Y_R^A}, \overline{y_R^A})$)
Receive (opened-cond, sid, b_0^A) from $\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}$		Receive (opened-cond, sid, b_0^B) from $\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}$
$b_1^A := \overline{Y_R^B} \in \mathsf{chState}[\lvert\mathsf{chState}\rvert - 1]$		$b_1^B := \overline{Y_R^A} \in \mathsf{chState}[\lvert\mathsf{chState}\rvert - 1]$
if $b_0^A \wedge b_1^A \neq 1$		If $b_0^B \wedge b_1^B \neq 1$
Go to $\mathsf{ForceClose}(\lvert\mathsf{chState}\rvert - 2)$		Go to $\mathsf{ForceClose}(\lvert\mathsf{chState}\rvert - 2)$
else		**else**
$\mathsf{chState}[\lvert\mathsf{chState}\rvert] := (\mathtt{tx}_c, \mathtt{tx}_s, \mathtt{tx}_p^A, Y_P^A, y_P^A, Y_P^B, Y_R^A, y_R^A, Y_R^B, \mathit{split\text{-}info})$		$\mathsf{chState}[\lvert\mathsf{chState}\rvert] := (\mathtt{tx}_c, \mathtt{tx}_s, \mathtt{tx}_p^B, Y_P^B, y_P^B, Y_P^A, Y_R^B, y_R^B, Y_R^A, \mathit{split\text{-}info})$

Figure 26: Update channel protocol in $(\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}, \mathcal{G}_{\mathsf{LedgerLocks}})$-hybrid world. Here, *GenCommit*, *GenSplit* and *GenPunish* denote the constructors for $\mathtt{tx}_c$, $\mathtt{tx}_s$ and $\mathtt{tx}_p$, respectively as described in Figure 24.

Protocol Π_{Channel}

Close Channel

```
Alice($\mathtt{tx}_f, aID_{AB}$)
$(\mathtt{tx}_c, \mathtt{tx}_s, \mathtt{tx}_p^A, Y_P^A, y_P^A, Y_P^B, Y_R^A, y_R^A, Y_R^B) \leftarrow \mathsf{chState}[|\mathsf{chState}| - 1]$
Set $\mathtt{tx}_t := \mathsf{ComputeBalance}(\mathtt{tx}_\mathsf{f}, \mathtt{tx}_\mathsf{s})$
Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$ with $(\mathsf{auth\text{-}tx}, \mathsf{sid}, \mathtt{tx}_t, aID_{AB})$
Receive $(\mathsf{auth\text{-}tx}, \mathsf{sid}, b_t)$ from $\mathcal{G}_{\mathsf{LedgerLocks}}$
Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$ with $(\mathsf{read}, \mathsf{sid})$
Receive $(\mathsf{read}, \mathsf{sid}, \mathsf{state})$ from $\mathcal{G}_{\mathsf{LedgerLocks}}$
Set $h_t := |\mathsf{state}|$
Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$ with $(\mathsf{submit}, \mathsf{sid}, \mathtt{tx}_t)$
while $|\mathsf{state}| < h_t + \#_{safe}$ :
  Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$ with $(\mathsf{read}, \mathsf{sid})$
  Receive $(\mathsf{read}, \mathsf{sid}, \mathsf{state})$ from $\mathcal{G}_{\mathsf{LedgerLocks}}$
if $\neg\mathsf{inState}(\mathtt{tx}_t, \mathsf{state})$ :
  Go to ForceClose($|\mathsf{chState}| - 1$)
```

```
Bob($\mathtt{tx}_f, aID_{AB}$)
$(\mathtt{tx}_c, \mathtt{tx}_s, \mathtt{tx}_p^A, Y_P^A, y_P^A, Y_P^B, Y_R^A, y_R^A, Y_R^B) \leftarrow \mathsf{chState}[|\mathsf{chState}| - 1]$
Set $\mathtt{tx}_t := \mathsf{ComputeBalance}(\mathtt{tx}_\mathsf{f}, \mathtt{tx}_\mathsf{s})$
Receive $(\mathsf{auth\text{-}req}, \mathsf{sid}, \mathtt{tx}_t, (aID_{AB}, (A, B)))$ from $\mathcal{G}_{\mathsf{LedgerLocks}}$
Send $(\mathsf{auth\text{-}rep}, \mathsf{sid}, b_t^B := 1)$ to $\mathcal{G}_{\mathsf{LedgerLocks}}$
```

Figure 27: Close channel protocol in ($\mathcal{G}_{\mathsf{Cond}}^{R, f_{merge}}$, $\mathcal{G}_{\mathsf{LedgerLocks}}$)-hybrid world. Here, *ComputeBalance* denotes the constructor for $\mathtt{tx}_t$ as described in Figure 24.

Protocol Π_{Channel}

ForceClose Channel

```
Alice($aID_{AB}, i$)
$(\mathtt{tx}_c, \mathtt{tx}_s, \mathtt{tx}_p^A, Y_P^A, y_P^A, Y_P^B, Y_R^A, y_R^A, Y_R^B, y_R^B) \leftarrow \mathsf{chState}[i]$
Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$ with $(\mathsf{release\text{-}tx}, \mathsf{sid}, \mathtt{tx}_c, aID_{AB}, Y_P^A, y_P^A)$
Receive $(\mathsf{release\text{-}tx}, \mathsf{sid}, b)$ from $\mathcal{G}_{\mathsf{LedgerLocks}}$
Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$ with $(\mathsf{read}, \mathsf{sid})$
Receive $(\mathsf{read}, \mathsf{sid}, \mathsf{state})$ from $\mathcal{G}_{\mathsf{LedgerLocks}}$
Set $h_c := |\mathsf{state}|$
while $|\mathsf{state}| < h_c + 2 \cdot \#_{safe}$ :
  Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$ with $(\mathsf{read}, \mathsf{sid})$
  Receive $(\mathsf{read}, \mathsf{sid}, \mathsf{state})$ from $\mathcal{G}_{\mathsf{LedgerLocks}}$
Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$ with $(\mathsf{submit}, \mathsf{sid}, \mathtt{tx}_s)$
```

Punish Channel

```
Alice($aID_{AB}, i$)
$(\mathtt{tx}_c, \mathtt{tx}_s, \mathtt{tx}_p^A, Y_P^A, y_P^A, Y_P^B, Y_R^A, y_R^A, Y_R^B, y_R^B) \leftarrow \mathsf{chState}[i]$
Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$ with $(\mathsf{signal\text{-}tx}, \mathsf{sid}, \mathtt{tx}_c, aID_{AB}, Y_P^B)$
Receive $(\mathsf{signal\text{-}tx}, \mathsf{sid}, y_P^B)$ from $\mathcal{G}_{\mathsf{LedgerLocks}}$
$y^{B*} := f_{merge}(wit, R, y_P^B, y_R^B)$
Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$ with $(\mathsf{release\text{-}tx}, \mathsf{sid}, \mathtt{tx}_p^A, aID_{AB}, Y^{B*}, y^{B*})$
Receive $(\mathsf{release\text{-}tx}, \mathsf{sid}, y_P^B)$ from $\mathcal{G}_{\mathsf{LedgerLocks}}$
```

Figure 28: ForceClose and Punish algorithms in ($\mathcal{G}_{\mathsf{Cond}}^{R, f_{merge}}$, $\mathcal{G}_{\mathsf{LedgerLocks}}$)-hybrid world.

Protocol Π_{Channel}

```
Monitor Channel
Alice($aID_{AB}$, chState)
while 1 :
    Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$ with (read, sid)
    Receive (read, sid, state) from $\mathcal{G}_{\mathsf{LedgerLocks}}$
    for $i \in [0, |\mathsf{chState}| - 2]$ :
        $(\mathsf{tx}_c, \mathsf{tx}_s, \mathsf{tx}_p^A, Y_P^A, y_P^A, Y_P^B, Y_R^A, y_R^A, Y_R^B, y_R^B) \leftarrow \mathsf{chState}[i]$
        if inState($\mathsf{tx}_c$, state) :
            Go to PunishChannel($aID_{AB}, i$)
```

Figure 29: Monitor channel algorithm in $(\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}, \mathcal{G}_{\mathsf{LedgerLocks}})$-hybrid world.

Protocol Π_{MultiHop}

```
Setup
$P_0(t^*, n)$
Set $t_n := t^*$
for $i \in [n, 1]$ :
    Set $t_{i-1} := t_i + 5 \cdot \#_{safe} + 1$
Set $\vec{t} := [t_0, t_{n-1}]$
Sample $(Y^0, y^0) \leftarrow \mathsf{GenR}(1^\lambda)$
Invoke $\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}$ on input (create-ind-cond, sid, $(Y^0, y^0)$)
Receive (created-ind-cond, sid, $Y^0$) from $\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}$
$\forall i \in [1, n-1]$ :
    Sample $(Y_{aux}, y_{aux}) \leftarrow \mathsf{GenR}(1^\lambda)$
    Invoke $\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}$ on input (create-ind-cond, sid, $(Y_{aux}, y_{aux})$)
    Receive (created-ind-cond, sid, $Y^*$) from $\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}$
    Invoke $\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}$ on input (create-merged-cond, sid, $(Y^{i-1}, Y_{aux})$)
    Receive (created-merged-cond, sid, $Y^i$) from $\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}$
return $\vec{Y}, \vec{y}, \vec{Y_{aux}}, \vec{y_{aux}}, \vec{t}$
```

```
Setup
$P_i(Y_{aux}, y_{aux}, Y^{i-1}, Y^i, t_i, t_{i-1})$
Invoke $\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}$ on input (open-cond, sid, $(Y_{aux}, y_{aux})$)
Receive (opened-cond, sid, $b_0$) from $\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}$
$b_1 := (Y^i \overset{?}{=} f_{merge}(stmt, R, (Y^{i-1}, Y^*)))$
$b_2 := t_{i-1} > t_i + 5 \cdot \#_{safe}$
if $b_0 \wedge b_1 \wedge b_2 \neq 1$ then abort
else ok
```

```
Setup
$P_n(Y, y)$
Invoke $\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}$ on input (open-cond, sid, $(Y, y)$)
Receive (opened-cond, sid, $b_0$) from $\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}$
if $b_0 \neq 1$ then abort
else ok
```

Figure 30: Setup algorithms in the $(\mathcal{G}_{\mathsf{Cond}}^{R,f_{merge}}, \mathcal{G}_{\mathsf{LedgerLocks}})$-hybrid world.

Protocol Π_{MultiHop}

Lock i-th channel

$P_i(\mathtt{tx}_f, aID_i, aID_{i+1}, Y^i, t_i, v_i, v_{i+1}, v_{lock})$	$P_{i+1}(\mathtt{tx}_f, aID_i, aID_{i+1}, Y^i, t_i, v_i, v_{i+1}, v_{lock})$
Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$ on input (create-account, sid, P_{i+1})	
	Receive (acc-req, (P_{i+1}, P_i)) from $\mathcal{G}_{\mathsf{LedgerLocks}}$
	Send (acc-rep, $b := 1$) to $\mathcal{G}_{\mathsf{LedgerLocks}}$
Receive (create-account, sid, $aID_{i,i+1}$) from $\mathcal{G}_{\mathsf{LedgerLocks}}$	Receive (create-account, sid, $aID_{i,i+1}$) from $\mathcal{G}_{\mathsf{LedgerLocks}}$
Set $\mathit{split\text{-}info} := [(aID_{i,i+1}, v_{lock}), (aID_i, v_i), (aID_{i+1}, v_{i+1})]$	Set $\mathit{split\text{-}info} := [(aID_{i,i+1}, v_{lock}), (aID_i, v_i), (aID_{i+1}, v_{i+1})]$
Invoke $\mathit{UpdateChannel}(\mathtt{tx}_f, aID_i, \mathit{split\text{-}info})$	Invoke $\mathit{UpdateChannel}(\mathtt{tx}_f, aID_{i+1}, \mathit{split\text{-}info})$
$\mathtt{ctx} := \mathit{GenPay}(\mathtt{tx}_s, aID_{i,i+1}, aID_{i+1}, 0)$	$\mathtt{ctx} := \mathit{GenPay}(\mathtt{tx}_s, aID_{i,i+1}, aID_{i+1}, 0)$
$\mathtt{rtx} := \mathit{GenPay}(\mathtt{tx}_s, aID_{i,i+1}, aID_i, t_i)$	$\mathtt{rtx} := \mathit{GenPay}(\mathtt{tx}_s, aID_{i,i+1}, aID_i, t_i)$
Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$ with (auth-tx, sid, $\mathtt{rtx}$, $aID_{i,i+1}$)	
	Receive (auth-req, sid, $\mathtt{rtx}$, $(aID_{i,i+1}, (P_i, P_{i+1}))$) from $\mathcal{G}_{\mathsf{LedgerLocks}}$
	Send (auth-rep, sid, $b := 1$) to $\mathcal{G}_{\mathsf{LedgerLocks}}$
Receive (auth-tx, sid, b) from $\mathcal{G}_{\mathsf{LedgerLocks}}$	
Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$ with (lock-tx, sid, $\mathtt{ctx}$, $aID_{i,i+1}$, Y^i)	
	Receive (lock-req, sid, $\mathtt{ctx}$, $(aID_{i,i+1}, (P_i, P_{i+1}))$, Y^i) from $\mathcal{G}_{\mathsf{LedgerLocks}}$
	Send (lock-rep, sid, $b := 1$, $aID_{i,i+1}$) to $\mathcal{G}_{\mathsf{LedgerLocks}}$
Receive (lock-tx, sid, b, $aID_{i,i+1}$) from $\mathcal{G}_{\mathsf{LedgerLocks}}$	
return $\mathtt{ctx}, \mathtt{rtx}, aID_{i,i+1}$	**return** $\mathtt{ctx}, \mathtt{rtx}, aID_{i,i+1}$

Figure 31: Lock protocol in the $(\mathcal{G}^{R,f_{merge}}_{\mathsf{Cond}}, \mathcal{G}_{\mathsf{LedgerLocks}})$-hybrid world. Here, *GenPay* denotse the constructors for $\mathtt{ctx}$ and $\mathtt{rtx}$, respectively as described in Figure 24.

Protocol Π_{MultiHop}

Off-Chain Pay i-th channel

```
$P_i(Y^{i-1}, Y_{aux}, y_{aux}, Y^i, y^i, t_{i-1})$
Invoke $\mathcal{G}^{R,f_{merge}}_{\mathsf{Cond}}$ on input (open-cond, sid, $(Y^i, y^i)$)
Receive (opened-cond, sid, $b_1$) from $\mathcal{G}^{R,f_{merge}}_{\mathsf{Cond}}$
Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$ with (read, sid)
Receive (read, sid, state) from $\mathcal{G}_{\mathsf{LedgerLocks}}$
Set $y^{i-1} := f_{merge}(wit, R, y^i, -y_{aux})$
**if** $b_1 \neq 1$ **return abort**
**else return** $y^{i-1}$
```

On-Chain Pay i-th channel

```
$P_i(\mathtt{ctx}^i, Y^i, y^i_{aux}, aID_{i,i+1})$
Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$ with (read, sid)
Receive (read, sid, state) from $\mathcal{G}_{\mathsf{LedgerLocks}}$
Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$with (read, sid)
Receive (read, sid, state) from $\mathcal{G}_{\mathsf{LedgerLocks}}$
**if** $\neg$inState($\mathtt{ctx}^i$, state)**then abort**
Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$ on input (signal-tx, sid, $aID_{i,i+1}$, $\mathtt{ctx}^i$, $Y^i$)
Receive (signal-tx, sid, $y^i$) from $\mathcal{G}_{\mathsf{LedgerLocks}}$
Set $y^{i-1} := f_{merge}(wit, R, y^i, -y^i_{aux})$
**return** $y^{i-1}$
```

Force Pay i-th channel

```
$P_i(Y^{i-1}, y^{i-1}, aID_{i-1,i}, \mathtt{ctx}^{i-1})$
Invoke $\mathit{ForceClose}(aID_{i-1,i})$
Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$ on input (release-tx, sid, $\mathtt{ctx}^{i-1}$, $aID_{i-1,i}$, $Y^{i_1}$, $y^{i-1}$)
Receive (release-tx, sid, $b$) from $\mathcal{G}_{\mathsf{LedgerLocks}}$
```

Refund Pay i-th channel

```
$P_i(aID_{i,i+1}, \mathtt{rtx}^i)$
Invoke $\mathit{ForceClose}(aID_{i,i+1})$
Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$ on input (submit-tx, sid, $\mathtt{rtx}^i$)
```

Figure 32: On-chain payment, off-chain payment, force payment and refund algorithms in the $(\mathcal{G}^{R,f_{merge}}_{\mathsf{Cond}}, \mathcal{G}_{\mathsf{LedgerLocks}})$-hybrid world.

Protocol Π_{MultiHop}

$P_0(t^*, \mathsf{tx}_f^{0,1}, aID_0, aID_1, v_{pay}, v_{fee}, \mathsf{chState}_{0,1}, n)$	$P_i(\mathsf{tx}_f^{i-1,i}, \mathsf{tx}_f^{i,i+1}, aID_{i-1}, aID_i, aID_{i+1})$	$P_n(\mathsf{tx}_f^{n-1,n}, aID_{n-1}, aID_n)$
$\vec{Y}, \vec{y}, \vec{Y_{aux}}, \vec{y_{aux}}, \vec{t} := \mathsf{Setup}(t^*, n)$		
$v_n := v_{pay}$		
$v_i := v_{pay} + (n - i - 1) \cdot v_{fee}$		
$y^{n-1} := f_{merge}(wit, R, y^{n-2}, y^{n-1}_{aux})$		
$\xrightarrow{(Y^{i-1}, Y^{i-1}_{aux}, y^{i-1}_{aux}, Y^i, \vec{t}, v_{i-1}, v_i) \text{ to } P_i}$		
$\xrightarrow{(Y^{n-1}, y^{n-1}, \vec{t}, v_n) \text{ to } P_n}$		
	$\mathsf{Setup}(Y_{aux}[i], y_{aux}[i], Y[i-1], Y[i], t[i], t[i-1])$	$\mathsf{Setup}(Y[n], y[n])$
$v_0^* \leftarrow GetBal(\mathsf{chState}_{0,1}[\lvert\mathsf{chState}_{0,1}\rvert - 1], aID_0)$	$v_{i-1}^* \leftarrow GetBal(\mathsf{chState}_{i-1,i}[\lvert\mathsf{chState}_{i-1,i}\rvert - 1], aID_{i-1})$	
$v_0' := v_0^* - v_0$	$v_{i-1}' := v_{i-1}^* - v_{i-1}$	
$v_1' \leftarrow GetBal(\mathsf{chState}_{0,1}[\lvert\mathsf{chState}_{0,1}\rvert - 1], aID_1)$	$v_i' \leftarrow GetBal(\mathsf{chState}_{i-1,i}[\lvert\mathsf{chState}_{i-1,i}\rvert - 1], aID_i)$	
$x_0 := (\mathsf{tx}_f^{0,1}, aID_0, aID_1, Y[0], t[0], v_0', v_1', v_0)$	$x_{i-1} := (\mathsf{tx}_f^{i-1,i}, aID_{i-1}, aID_i, Y[i-1], t[i-1], v_{i-1}', v_i', v_{i-1})$	
$in^0 := \mathsf{Lock}(x_0)$	$in^{i-1} := \mathsf{Lock}(x_{i-1})$ with P_{i-1}	
	$v_i^* \leftarrow GetBal(\mathsf{chState}_{i,i+1}[\lvert\mathsf{chState}i, i+1\rvert - 1], aID_i)$	
	$v_i'' := v_i^* - v_i$	
	$v_{i+1}' \leftarrow GetBal(\mathsf{chState}_{i,i+1}[\lvert\mathsf{chState}i, i+1\rvert - 1], aID_{i+1})$	
	$x_i := (\mathsf{tx}_f^{i,i+1}, aID_i, aID_{i+1}, Y[i], t[i], v_i'', v_{i+1}', v_i)$	
	$in^i := \mathsf{Lock}(x_i)$ with P_{i+1}	
		$v_{n-1}^* \leftarrow GetBal(\mathsf{chState}_{n-1,n}[\lvert\mathsf{chState}_{n-1,n}\rvert - 1], aID_{n-1})$
		$v_{n-1}'' := v_{n-1}^* - v_{n-1}$
		$v_n' \leftarrow GetBal(\mathsf{chState}_{n-1,n}[\lvert\mathsf{chState}_{n-1,n}\rvert - 1], aID_n)$
		$x_{n-1} := (\mathsf{tx}_f^{n-1,n}, aID_{n-1}, aID_n, Y[n-1], t[n-1], v_{n-1}'', v_n', v_{n-1})$
		$in^{n-1} := \mathsf{Lock}(x_{n-1})$ with P_{n-1}
$\mathsf{ctx}^0, \mathsf{rtx}^0, aID_{0,1} \leftarrow in^0$	$\mathsf{ctx}^{i-1}, \mathsf{rtx}^{i-1}, aID_{i-1,i} \leftarrow in^{i-1}$	
	$\mathsf{ctx}^i, \mathsf{rtx}^i, aID_{i,i+1} \leftarrow in^{i+1}$	
		$\mathsf{ctx}^{n-1}, \mathsf{rtx}^{n-1}, aID_{n-1,n} \leftarrow in^{n-1}$
	Set $\overline{y[i-1]} := \bot$	
		$\xleftarrow{y[n-1] \text{ to } P_{n-1}}$
		$split\text{-}info := [(aID_{n-1}, v_{n-1}''), (aID_n, v_n' + v_{n-1})]$
		$UpdateChannel(\mathsf{tx}_f^{n-1,n}, aID_n, split\text{-}info)$ with P_{n-1}

```
while 1 :
    Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$ with (read, sid)
    Receive (read, sid, state) from $\mathcal{G}_{\mathsf{LedgerLocks}}$
    if $|\mathsf{state}| \geq t[0] - 3 \cdot \#_{safe} - 1$ :
        Refund($\mathsf{rtx}^0, aID_{0,1}$)
    if receive $y[0]$ from $P_1$
        $split\text{-}info := [(aID_0, v_0'), (aID_1, v_1' + v_0)]$
        $UpdateChannel(\mathsf{tx}_f^{0,1}, aID_0, split\text{-}info)$ with $P_1$
```

```
while 1 :
    Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$ with (read, sid)
    Receive (read, sid, state) from $\mathcal{G}_{\mathsf{LedgerLocks}}$
    if receive $y[i]$ from $P_{i+1}$
        $\overline{y[i-1]}$ := Off-Pay(Y[i], y[i], t[i-1])
        if $\overline{y[i-1]} \neq \bot \wedge |\mathsf{state}| < t[i-1] - 4 \cdot \#_{safe} - 1$ :
            $\xleftarrow{\overline{y[i-1]} \text{ to } P_{i-1}}$
        if $|\mathsf{state}| < t[i] - 4 \cdot \#_{safe} - 1$ :
            $split\text{-}info := [(aID_i, v_i''), (aID_{i+1}, v_{i+1}' + v_i)]$
            $UpdateChannel(\mathsf{tx}_f^{i,i+1}, aID_i, split\text{-}info)$ with $P_{i+1}$
    $\overline{y[i-1]}$ := On-Pay($\mathsf{ctx}^i, Y[i], y_{aux}[i], aID_{i,i+1}$)
    if $\overline{y[i-1]} \neq \bot \wedge |\mathsf{state}| < t[0] - 4 \cdot \#_{safe} - 1$ :
        $\xleftarrow{\overline{y[i-1]} \text{ to } P_{i-1}}$
        $split\text{-}info' := [aID_{i-1}, v_{i-1}'), (aID_i, v_i' + v_{i-1}]$
        $UpdateChannel(\mathsf{tx}_f^{i-1,i}, aID_i, split\text{-}info')$ with $P_{i-1}$
    if $\overline{y[i-1]} \neq \bot \wedge |\mathsf{state}| \geq t[i-1] - 4 \cdot \#_{safe} - 1$ :
        ForcePay($Y[i-1], \overline{y[i-1]}, \mathsf{ctx}^{i-1}, aID_{i-1,i}$)
    if $|\mathsf{state}| \geq t[1] - 3 \cdot \#_{safe} - 1$ :
        Refund($\mathsf{rtx}^i, aID_{i,i+1}$)
```

```
while 1 :
    Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$ with (read, sid)
    Receive (read, sid, state) from $\mathcal{G}_{\mathsf{LedgerLocks}}$
    if $|\mathsf{state}| \geq t[n-1] - 4 \cdot \#_{safe} - 1$ :
        ForcePay($Y[n-1], y[n-1], \mathsf{ctx}^{n-1}, aID_{n-1,n}$)
```

Figure 33: Multi-hop payment protocol in the $(\mathcal{G}^{R,f_{merge}}_{\mathsf{Cond}}, \mathcal{G}_{\mathsf{LedgerLocks}})$-hybrid world.

Ideal Functionality $\mathcal{G}_{\text{Ledger}}$

General: The functionality is parameterized by four algorithms Validate, ExtendPolicy, Blockify, and predict-time, along with two parameters `windowSize`, `Delay` $\in \mathbb{N}$. The functionality manages variables state, NxtBC, buffer, τ_L and $\vec{\tau}_{\text{state}}$. Initially, state $:= \vec{\tau}_{\text{state}}$, NxtBC $:= \varepsilon$, buffer $:= \emptyset$, $\tau_L = 0$.
Party Management: The functionality maintains the set of registered parties $\mathcal{P}$, the (sub-)set of honest parties $\mathcal{H} \subseteq \mathcal{P}$, and the (sub-)set of de-synchronized honest parties $\mathcal{P}_{DS} \subset \mathcal{H}$. The sets $\mathcal{P}, \mathcal{H}, \mathcal{P}_{DS}$ are all initially set to $\emptyset$. When a new honest party is registered at the ledger, if it is registered with the clock already, then it is added to the party set $\mathcal{H}$ and $\mathcal{P}$, and the current time of registration is also recorded; if the current time is $\tau_L > 0$, it is also added to $\mathcal{P}_{DS}$. Similarly, when a party is deregistered, it is removed from both $\mathcal{P}$ (and therefore also from $\mathcal{P}_{DS}$ and $\mathcal{H}$). The ledger maintains an invariant that it is registered (as a functionality) to the clock whenever $\mathcal{H} \neq \emptyset$. A party is considered fully registered if it is registered with the ledger and the clock.

Upon receiving any input I from any party or from the adversary, send (clock-read, sid_C) to $\mathcal{G}_{\text{Clock}}$ and upon receiving response (clock-read, sid_C, τ) set $\tau_L = \tau$ and do the following:

(1) Let $\widehat{\mathcal{P}} \subseteq \mathcal{P}_{DS}$ denote the set of de-synchronized honest parties that have been registered (continuously) since time $\tau' < \tau_L - \texttt{Delay}$ (to both ledger and clock). Set $\mathcal{P}_{DS} := \mathcal{P}_{DS} \setminus \widehat{\mathcal{P}}$. On the other hand, for any synchronize party $P \in \mathcal{H} \setminus \mathcal{P}_{DS}$, if P is not registered to the clock, then $\mathcal{P}_{DS} \cup \{P\}$.
(2) If I was received from an honest party $P_i \in \mathcal{P}$:
- Set $\vec{I}_H^T = \vec{I}_H^T \| (I, P, \tau_L)$.
- Compute $\vec{N} = (\vec{N}_1, \ldots, \vec{N}_\ell) := \text{ExtendPolicy}(\vec{I}_H^T, \text{state}, \text{NxtBC}, \text{buffer}, \vec{\tau}_{\text{state}})$ and if $\vec{N} \neq \varepsilon$, set state $:=$ state$\|\text{Blockify}(\vec{N}_1)\| \cdots \|\text{Blockify}(\vec{N}_\ell)$ and $\vec{\tau}_{\text{state}} := \vec{\tau}_{\text{state}} \| \tau_L^\ell$, $\tau_L^\ell = \tau_L \| \cdots \| \tau_L$.
- For each BTX $\in$ buffer: if Validate(BTX, state, buffer) $= 0$, then delete BTX from buffer. Also, reset NxtBC $:= \varepsilon$.
- If there exists $P_j \in \mathcal{H} \setminus \mathcal{P}_{DS}$ such that $|\text{state}| - \text{pt}_j > \texttt{windowSize}$ or $\text{pt}_j < |\text{state}_j|$, then $\text{pt}_k := |\text{state}|$ for all $P_k \in \mathcal{H} \setminus \mathcal{P}_{DS}$.

(3) Depending on the input I and the ID of the sender, execute the respective code:
- *Submitting a transaction:*
  If $I =$ (submit, sid, tx) and is received from a party $P_i \in \mathcal{P}$ or from $\mathcal{S}$ (on behalf of a corrupted party P_i), do the following:
  - Choose a unique transaction ID txid and set BTX $:= (\text{tx}, \text{txid}, \tau_L, P_i)$.
  - If Validate(BTX, state, buffer) $= 1$, then buffer $:=$ buffer $\cup$ {BTX}.
  - Send (submit, BTX) to $\mathcal{S}$.
- *Reading the state:*
  If $I =$ (read, sid) is received from a fully registered party P, then set $\text{state}_i := \text{state}|_{\min \text{pt}_i, |\text{state}|}$ and return (read, sid, state_i) to the requester. If requester is $\mathcal{S}$, then send (state, buffer, $\vec{I}_H^T$) to $\mathcal{S}$.
- *Maintaining the ledger state:*
  If $I =$ (maintain-ledger, sid, minerID) is received by an honest party $P_i \in \mathcal{P}$, and (after updating $\vec{I}_H^T$ as above) predict−time$(\vec{I}_H^T) = \hat{\tau} > \tau_L$, then send (clock-update, sid_C) to $\mathcal{G}_{\text{Clock}}$. Else, send I to $\mathcal{S}$.
- *The adversary proposing the next block:*
  If $I =$ (next-block, hFlag, $(\text{txid}_1, \ldots, \text{txid}_\ell)$) is sent from the adversary, update NxtBC as follows:
  - Set listOfTxid $\leftarrow \epsilon$.
  - For $i = 1, \ldots, \ell$ do: if there exists BTX $:= (x, \text{txid}, \text{minerID}, \tau_L, P_i) \in$ buffer with ID txid $= \text{txid}_i$, then set listOfTxid $:=$ listOfTxid$\|\text{txid}_i$.
  - Finally, set NxtBC $:=$ NxtBC$\|$(hFlag, listOfTxid) and output (next-block, ok) to $\mathcal{S}$.
- *The adversary setting state-slackness:*
  If $I =$ (set-slack, $(P_{i_1}, \widehat{\text{pt}}'_{i_1}), \ldots, (P_{i_\ell}, \widehat{\text{pt}}'_{i_\ell})$), with $\{P_{i_1}, \ldots, P_{i_\ell}\} \subseteq \mathcal{H} \setminus \mathcal{P}_{DS}$ is received from the adversary $\mathcal{S}$, do the following:
  - If for all $j \in [\ell]$: $|\text{state}| - \widehat{\text{pt}}_{i_j} \leq \texttt{windowSize}$ and $\text{pt}_{i_1} := \widehat{\text{pt}}_{i_1}$ for every $j \in [\ell]$ and return (set-slack, ok) to $\mathcal{S}$.
  - Otherwise, set $\text{pt}_{i_j} := |\text{state}|$ for all $j \in [\ell]$.
- *The adversary setting the state for de-synchronized parties:*
  If $I =$ (desync-state, $(P_{i_1}, \text{state}'_{i_1}), \ldots, (P_{i_\ell}, \text{state}'_{i_\ell})$), with $\{P_{i_1}, \ldots, P_{i_\ell}\} \subseteq \mathcal{P}_{DS}$ from the adversary $\mathcal{S}$, set $\text{state}_{i_j} = \text{state}'_{i_j}$ for each $j \in [\ell]$ and return (desync-state, ok) to $\mathcal{S}$.

Figure 34: Ideal functionality $\mathcal{G}_{\text{Ledger}}$ [8].