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 cryptography. 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 re-proving 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 \( G_{\text{LedgerLocks}} \), 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.

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 undiscovered 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...
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 \textit{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 \(G_L\)) 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 \(G_{\text{LedgerLocks}}\) 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 \(G_{\text{LedgerLocks}}\) 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 \(G_{\text{LedgerLocks}}\) and is adaptable to the protocol needs. Further, to show that \(G_{\text{LedgerLocks}}\) 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 \(G_{\text{LedgerLocks}}\) 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 \(G_{\text{Cond}}\), which encodes operations over conditions, such as their composition. \(G_{\text{Cond}}\) 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 \(F_{\text{AdaptSig}}\) 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 \(G_{\text{Ledger}}\) 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 \(G_{\text{LedgerLocks}}\), an ideal functionality that models a ledger with generic AS-locked transactions. We provide a protocol \(\Pi_{\text{LedgerLocks}}\) that UC-realizes \(G_{\text{LedgerLocks}}\) in the presence of \(G_{\text{Ledger}}\) from [8] and \(F_{\text{AdaptSig}}\) (§7).
- We demonstrate the flexibility of our framework, by using it to describe an enhanced atomic swap protocol \(\Pi_{\text{AtomicSwap}}\) and a multi-hop payment protocol \(\Pi_{\text{MultiHop}}\) over payment channels \(\Pi_{\text{Channel}}\), all of them protocols relying on AS-locked transactions. To this end, we instantiate \(G_{\text{LedgerLocks}}\) with support for transaction timelocks, which are crucial for atomic swap and multi-hop payment security. The description of \(\Pi_{\text{AtomicSwap}}, \Pi_{\text{Channel}}\) and \(\Pi_{\text{MultiHop}}\) 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 \(G_{\text{LedgerLocks}}\) (§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 \(A\) (blue), \(B\) (orange) and two users Alice (A) and Bob (B), holding assets in \(A\) and \(B\), respectively. A correct atomic swap protocol ensures that Alice receives Bob’s assets on \(B\) and Bob receives Alice’s assets on \(A\) 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 \(AB\) in the respective chain (through deposit transaction \(dtx_A\) and \(dtx_B\)). 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 \(tx_A\) and \(tx_B\) 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 \(rtx_A\) and \(rtx_B\) 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 \(B\) (using \(tx_B\)). Then, Bob can read \(x\) from \(B\) and use it to claim the assets in the shared account on \(A\) (using \(tx_A\)). Alternatively (e.g., if the other user fails to cooperate), Alice and Bob can refund their assets by publishing \(tx_A\) or \(tx_B\) respectively. Alice’s refund transaction \(tx_A\) is equipped with a timelock that ensures that it can only be published after time \(t\). This restriction prevents Alice from simultaneously publishing \(tx_A\) and \(tx_B\) to retrieve the assets on both chains. Instead, if Alice has not claimed Bob’s assets (through \(tx_B\)) until time right before \(t\), Bob can publish \(tx_B\) to be refunded before \(tx_A\) 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 \(tx_A\) will be included immediately after Bob sent it to the network. In practice, the ledger only guarantees that \(tx_B\) will be included in the blockchain within a time delay \(\Delta\). During this time, other transactions (even if submitted later) may be included in the blockchain, invalidate \(tx_B\) and, thus, prevent its inclusion. Specifically, a malicious Alice could send \(tx_B\) right after observing \(tx_A\) on the network. As a consequence, \(tx_A\) could be included in the blockchain first, canceling \(tx_B\).

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 \(tx_B\) (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 \(tx_A\). In the latter case, Bob learns \(x\) and publishes \(tx_B\), which is guaranteed to be included before \(t\), the time starting from which Alice could publish \(tx_A\).

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 \(tx_A\)), a malicious Bob can learn \(x\) and still outrun \(tx_A\). With \(tx_B\), then, Bob could claim Alice’s assets (publishing \(tx_A\)) before Alice could refund her assets using \(tx_B\) at time \(t\).

This issue can be mitigated when introducing an additional timelock for Bob’s refund transaction \(tx_B\) 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 $\mathbf{c}_{x_{i}}$ 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 $\mathbf{c}_{x_{i}}$, this will reveal a secret $x_i$ to $u_i$ allowing them to satisfy $c_{i-1}$ and claim $\mathbf{c}_{x_{i-1}}$ in turn. Once all conditional payments are set up, $S$ initiates the payment by revealing a secret $s_k$ 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 $\mathbf{c}_{x_{i}}$. If $u_i$ is not collaborating, $u_{i+1}$ can close the channel and use $\mathbf{c}_{x_{i}}$ 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 $\mathbf{c}_{x_{i}}$ prepare a refund transaction $\mathbf{c}_{x_{i}}$ 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 $\mathbf{c}_{x_{i}}$. However, even after successful channel closure while waiting for $\mathbf{c}_{x_{i}}$ to be included in the blockchain, $u_{i+1}$ may still decide to publish $\mathbf{c}_{x_{i}}$ instead. In this case, $u_i$ needs to observe $\mathbf{c}_{x_{i}}$ 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 $\mathbf{c}_{x_{i-1}}$ is still valid at this point and so that $\mathbf{c}_{x_{i-2}}$ 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 $\mathbf{c}_{x_{i}}$ needs to be adjusted such that $t_i < t_{i-1} + 2\Delta + \Delta_{close}$ (for $t_{i-1}$ being the timelock of $\mathbf{c}_{x_{i-1}}$ and $\Delta_{close}$ being the channel closing time) to ensure that $u_i$ after closing their outgoing channel and publishing $\mathbf{c}_{x_{i}}$ at $t_i$, when learning (latest) at $t_i + \Delta$ whether $\mathbf{c}_{x_{i}}$ or $\mathbf{c}_{x_{i-1}}$ got included in the blockchain there is still enough time to close their ingoing channel and publish $\mathbf{c}_{x_{i-1}}$ 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 $\mathbf{c}_{x_{i}}$ 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 $s_k$ 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 $\Delta_{close}$) and then publish $\mathbf{c}_{x_{n-1}}$ at time $t < t_{n}$. At $t_{n}$, however, a malicious $u_{n-1}$ already submitted $\mathbf{c}_{x_{n-1}}$ to outrun $\mathbf{c}_{x_{n}}$ resulting in $u_{n-1}$ being refunded while still learning $s_k$. With the knowledge of $s_k$, $u_{n-1}$ 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 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 $\mathcal{G}_{ledger}$ 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 $\mathcal{L}_{\Lambda}$ that let the attacker delay the inclusion of a transaction up to delay $\Delta$ [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 $t_1 \leq t_0 + \Delta$ (the exact value of $t_1$ 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 $t_1$. 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 $\mathcal{G}_{ledger}$ 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.
Figure 5: Overview of the infrastructure of LedgerLocks. Dark-gray components indicate novel functionalities and protocols introduced in this work. The \( \prec \) relation denotes that a protocol \( \Pi \) (left) realizes a functionality \( \mathcal{F} \lvert \mathcal{G} \) (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 \( \Pi \) is proven secure w.r.t. an ideal functionality \( \mathcal{F} \), the security of protocols using \( \Pi \) 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 (\( \mathcal{G}_{\text{Cond}} \)), adapter 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 \( (\mathbb{G}, q, g) \) (where \( g \) is a group element (the statement) \( Y = g^\eta \)), it is hard to compute the exponent \( \eta \) (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 \((y_i^{\text{mask}})_{i \in \{1, 10\}}\) and sends the corresponding statements \((y_i^{\text{guess}})_{i \in \{1, 10\}}\) to Bob. Bob himself prepares one witness \( y_{\text{win}} \) and ten witnesses \((y_i^{\text{blank}})_{i \in \{1, 10\}}\). For the guess \( j \), Bob prepares \( y_{\text{guess}}^j = (y_{\text{guess}}^j)_{i \in \{1, 10\}} \) such that for \( i \neq j \), \( y_i^{\text{guess}} = (y_i^{\text{mask}})_{i \in \{1, 10\}} \) and \( y_j^{\text{guess}} = q^{y_{\text{win}}} \). At this point, Bob knows the witness for \( y_{\text{guess}}^j \), while he cannot know the witness for any statement \( y_{\text{guess}}^i \) with \( i \neq j \). Since for this, Bob would require Alice’s secret masking values \((y_i^{\text{mask}})_{i \in \{1, 10\}}\). Bob proves in zero-knowledge to Alice that \( y_{\text{guess}}^j \) is well-formed. Now, Alice chooses a number \( m \) and prepares a payment to Bob based on \( y_m^{\text{guess}} \). Alice at this point, cannot know which condition Bob can open, only that Bob can open exactly one 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., \( y_{\text{guess}}^j \)) and protocols using these conditions (e.g., the payment from Alice to Bob based on \( y_m^{\text{guess}} \)).

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) Composite conditions, which combine two existing conditions. The concrete composition operation depends on the underlying hard relation and is specified as a parameter \( f_{\text{merge}} \) 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 \( f_{\text{merge}} \) as a parameter to \( \mathcal{G}_{\text{Cond}} \). This is needed to enable composability with the adapter 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 \( \Pi_{\text{Cond}} \).

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 \( \Pi \), even if \( \Pi \) 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 \( \Pi \) 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 \( \mathcal{G}_{\text{Cond}} \)
We can show that all adaptor signature instances resulting from the transformation realize $F$ for the same $f$ for condition generation. Without fixing $f_{\text{adapt}}$, it would be required to make assumptions on the way that protocols relying on $F_{\text{AdaptSig}}$ use $G_{\text{Cond}}$ for condition generation. We include a more detailed discussion of this aspect in §6.

In more detail, $G_{\text{LedgerLocks}}$ 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 learns the witness already upon transaction submission. Account owners learn the corresponding witness, while a malicious account owner learns the witness already upon transaction submission.

We build $G_{\text{LedgerLocks}}$ from $G_{\text{Ledger}}$ 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.

Lock-enabling ledger. In §7, we define $G_{\text{LedgerLocks}}$, an ideal functionality for a distributed ledger with AS-locked transactions. In the design of $G_{\text{LedgerLocks}}$, we follow the technique in [8]: The authors in [8] provide $G_{\text{Ledger}}$, 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 $\Pi_{\text{Ledger}}$, a description of the Bitcoin backbone protocol and prove that it UC-realizes $G_{\text{Ledger}}$.

$G_{\text{Ledger}}$ and $\Pi_{\text{Ledger}}$ 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 $G_{\text{Ledger}}$ in two ways: (i) instantiating isValidTx predicate to account for the specific transaction formats and ledger logics; and (ii) extending the API of $G_{\text{Ledger}}$ to account for further ledger features. Our ideal functionality $G_{\text{LedgerLocks}}$ follows this blueprint to model multi-party account-based transaction authorization.

Erkan Tairi, Pedro Moreno-Sanchez, and Clara Schneidewind

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

DLOG
R
§
DLOG
ECDSA
DLOG
Katz-Wang
DLOG
Schnorr
R
R
§
R
§
R
§
DLOG
R
R

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 $F_{\text{AdaptSig}}$, an ideal functionality for adaptor signatures. Following the two-party adaptor signatures scheme defined by Erwig et al. [18], our ideal functionality $F_{\text{AdaptSig}}$ 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 $\sigma$ on $m$ with condition $Y$ and the corresponding full signature $\sigma'$ 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 $F_{\text{AdaptSig}}$ relies on $G_{\text{Cond}}$ to handle conditions. This means that for signature adaption $F_{\text{AdaptSig}}$ queries $G_{\text{Cond}}$ for determining whether the correct witness to open a condition was provided. Similar to $G_{\text{Cond}}$, $F_{\text{AdaptSig}}$ is parameterized by a hard relation $R$, and additionally a function $f_{\text{adapt}}$, which transforms a pre-signature $\sigma$ and a witness $y$ into a corresponding full signature $\sigma'$. This parametrization enables $F_{\text{AdaptSig}}$ to use conditions as provided by $G_{\text{Cond}}$ in a truly modular fashion. Without fixing $f_{\text{adapt}}$, it would be required to make assumptions on the way that protocols relying on $F_{\text{AdaptSig}}$ use $G_{\text{Cond}}$ for condition generation. We include a more detailed discussion of this aspect in §6.

In practice, the parametrization by $f_{\text{adapt}}$ does not pose a restriction. Indeed, we can show that $F_{\text{AdaptSig}}$ 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 $f_{\text{adapt}}$ with certain generic properties. We can show that all adaptor signature instances resulting from the transformation realize $F_{\text{AdaptSig}}$ for the same $f_{\text{adapt}}$ function as used in the transformation. More precisely, we give a generic wrapper protocol $\Pi_{\text{AdaptSig}}$ around the algorithmic interface of two-party adaptor signature schemes (as defined in [18]) and prove this protocol to realize $F_{\text{AdaptSig}}$ 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 $F_{\text{AdaptSig}}$.

LedgerLocks. In §6, we define $G_{\text{LedgerLocks}}$, an ideal functionality for a distributed ledger with AS-locked transactions. The design of $G_{\text{LedgerLocks}}$ follows this blueprint in [8]: The authors in [8] provide $G_{\text{Ledger}}$, an ideal functionality modeling the subtleties of real-world blockchain consensus, in particular, realistic guarantees about the inclusion of transactions into the ledger.

In this manner, the UC-realization proof holds for any instantiation of this predicate. Moreover, one can leverage the results in [8] by extending $G_{\text{Ledger}}$ in two ways: (i) instantiating isValidTx predicate to account for the specific transaction formats and ledger logics; and (ii) extending the API of $G_{\text{Ledger}}$ to account for further ledger features. Our ideal functionality $G_{\text{LedgerLocks}}$ follows this blueprint to model multi-party account-based transaction authorization.
and 3) instantiating the valid predicate to check for correct transaction authorization. The operation for releasing an AS-locked transaction thereby makes use of $G_{\text{Cond}}$ 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 $G_{\text{LedgerLocks}}$, e.g., support for timelocks as we will show in §8.

Finally, we show how to realize $G_{\text{LedgerLocks}}$ with a protocol $\Pi_{\text{LedgerLocks}}$, which uses $G_{\text{Ledger}}$ and $F_{\text{AdaptSig}}$. Thanks to our modeling of $G_{\text{Cond}}$ 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 $Y_0$ chosen by the oracle so that the oracle needs to communicate its witness for $Y_0$ to Alice for enabling the swap.

To express this protocol (denoted by $\Pi_{\text{AtomicSwap}}$), we (partially) instantiate the CheckBase predicate of $G_{\text{LedgerLocks}}$ to encode a timelock check. Since $G_{\text{LedgerLocks}}$ is an extension of $G_{\text{Ledger}}$, 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 $\Pi_{\text{AtomicSwap}}$ does not only rely on $G_{\text{LedgerLocks}}$ but also on $G_{\text{Cond}}$ for the condition creation. The swap conditions are created by composing two conditions $Y$ and $Y_0$. Since LedgerLocks is fully modular with respect to $G_{\text{Cond}}$, it is not even necessary to fix the protocol to create $Y_0$.

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 $\Sigma_2$ 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 $L_R := \{ Y \ | \ \exists y \text{ s.t. } (Y, y) \in R \}$. We say that $R$ is a hard if:

- There exists a PPT sampling algorithm $\text{GenR}(1^k)$ that on input the security parameter $\lambda$ outputs a statement/witness pair $(Y, y) \in R$.
- The relation is poly-time decidable.
- For all PPT adversaries $A$ there exists a negligible function $\negl$, such that:
  \[
  \Pr \left( (Y, y') \in R \mid (Y, y) \leftarrow \text{GenR}(1^k), y' \leftarrow A(Y) \right) \leq \negl(\lambda),
  \]
  where probability is over the randomness of $\text{GenR}$ and $A$.

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 $Y$. 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 $\Sigma_2 = (\text{Setup}, \text{Gen}, \text{Sig}, \text{KAgg}, \text{Vf})$. It is run between parties $P_0, P_1$ and consists of a tuple $\Sigma_2^{\text{Ext}} = (\text{PreSig}, \text{Adapt}, \text{PreVf}, \text{Ext})$ of efficient protocols and algorithms defined as follows:

\begin{itemize}
  \item $\Pi_{\text{PreSig}}(sk_0, sk_1, m, Y)$: is an interactive protocol with input secret keys $sk_i$ from party $P_i$ with $i \in \{0, 1\}$ and common message $m \in \{0, 1\}^*$, public keys $pk_0, pk_1$ and statement $Y \in L_R$, outputs a pre-signature $\sigma$.
  \item $\Pi_{\text{PreVf}}(apk, m, Y, \sigma)$: 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 $\sigma$, outputs bit $b$.
  \item $\Pi_{\text{Adapt}}(apk, \sigma, y)$: is a DPT algorithm with input an aggregated public key apk, pre-signature $\sigma$ and witness $y$, outputs a signature $\sigma$.
  \item $\Pi_{\text{Ext}}(apk, \sigma, \sigma, Y)$: is a DPT algorithm with input an aggregated public key apk, a signature $\sigma$, pre-signature $\sigma$ and statement $Y \in L_R$, outputs a witness $y$ s.t. $(Y, y) \in R$, or $\bot$.
\end{itemize}

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 a two-party existential unforgeability under chosen message attacks (2-EUF-CMA) but additionally requires that producing a forgery $\sigma$ 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 $(Y, y) \in R$. 

7

LedgerLocks: A Security Framework for Blockchain Protocols Based on Adaptor Signatures
Finally, witness extractability guarantees that a valid signature/pre-signature pair \((\sigma, \delta)\) 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 \(\operatorname{EXEC}_{\pi,\mathcal{A},\mathcal{E}}(1^k)\) 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 \(\pi\) be a protocol. The output of an environment \(E\) interacting with protocol \(\pi\) and an adversary \(\mathcal{A}\), on input the security parameter \(1^k\) and auxiliary input \(z\), is denoted as \(\operatorname{EXEC}_{\pi,\mathcal{A},\mathcal{E}}(1^k, z)\). Let \(\phi_{\pi}\) be the ideal protocol for an ideal functionality \(\mathcal{F}\), i.e., \(\phi_{\pi}\) is a trivial protocol in which the parties simply forward their inputs to the ideal functionality \(\mathcal{F}\). The output of an environment \(E\) interacting with protocol \(\phi_{\pi}\) and an adversary \(\mathcal{S}\) (also called the simulator), on input the security parameter \(1^k\) and auxiliary input \(z\), is denoted as \(\operatorname{EXEC}_{\phi_{\pi},\mathcal{S},\mathcal{E}}(1^k, z)\).

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

**Definition 4 (UC Security).** We say a protocol \(\pi\) UC-realizes an ideal functionality \(\mathcal{F}\), if for every adversary \(\mathcal{A}\) there exists an adversary \(\mathcal{S}\) such that
\[
\left\{ \operatorname{EXEC}_{\pi,\mathcal{A},\mathcal{E}}(1^k, z) \right\}_{\lambda \in \mathbb{N}} \approx_e \left\{ \operatorname{EXEC}_{\phi_{\pi},\mathcal{S},\mathcal{E}}(1^k, z) \right\}_{\lambda \in \mathbb{N}},
\]
where \(\approx_e\) 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 \(\mathcal{G}_{\text{Cond}}\).

**Global conditions functionality.** We illustrate the (global) ideal functionality \(\mathcal{G}_{\text{Cond}}\) for conditions in Figure 7. \(\mathcal{G}_{\text{Cond}}\) is parameterized by a hard relation \(R\) and merging function \(\gamma_{\text{merge}}\). \(\mathcal{G}_{\text{Cond}}\) 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 \(L\). 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 \(\gamma_{\text{merge}}\). This function is split into an operation \(\cdot\) 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 \(L\)).

**Global conditions protocol.** We describe the global conditions protocol \(\Pi_{\text{Cond}}\) in Figure 8 for DLOG. The protocol is parameterized with a group description \((\mathbb{G}, g, q)\) and the discrete logarithm (DLOG) relation \(R_{\text{DLOG}}\) over it, i.e., \((Y, y) \in R_{\text{DLOG}} \iff Y = g^y\). We assume that the group \(\mathbb{G}\) is a DLOG-hard group here. The function \(\gamma_{\text{merge}}\) defines the witness operation \((+\cdot)\) as addition and the statement 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 \(R_{\text{DLOG}}\)) 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 \(R_{\text{DLOG}}\), and returns the output bit \(b\).
The functionality is parameterized by a hard relation $R$ and an adaptation function $f_{\text{adapt}}$. It maintains the list $K$ that stores all generated keys: the list $Q$ that stores tuples $(m, \sigma, v)$ representing message, signature, key and a verification flag, and the list $P$ that stores tuples $(m, \sigma, \alpha, v, y, f)$ 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 (keygen, sid) from $P_0$ and $P_1$, verify that sid = $(P_0, P_1, \text{sid}')$ for some sid'. If not, ignore the request. Else, send (keygen, sid) to $S$. Upon receiving (verification-key, sid, v) from $S$, add v into $K[\text{sid}]$ and send (verification-key, sid, v) to $P_0$ and $P_1$.

**Adaptation:** Upon receiving (adapt, $\sigma$, v, y) from some party $P$, check if there is an entry $f = (m, \delta, \alpha, v, \gamma, 1) \in P[\text{sid}]$. If not, then ignore this request. Else, send (open-cond, sid, (Y, y)) to $G_{\text{merge}}^{\text{cond}}$. Upon receiving (open-cond, sid, b), if $b = 0$, then abort. Else set $\sigma = f_{\text{adapt}}(\delta, y)$ and update $f = (m, \delta, \sigma, v, Y, 1) \in P[\text{sid}]$, and send (adapted-signature, signature, key, condition, witness) to $P$. (This guarantees pre-signature adaptability: any valid pre-signature $\sigma$ can be adapted into a valid full signature $\sigma$ using the witness $y$.)

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

**Output**

- If $f \in P[\text{sid}]$, then the verification succeeds.
- Else, if $f \not\in P[\text{sid}]$, then the verification fails.

Our signature functionality $\Pi_{\text{AdaptSig}}$ is required to stay fully modular with respect to $G_{\text{Cond}}$. By using $G_{\text{Cond}}$ 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 $\sigma$ and the corresponding adapted signature $\sigma = f_{\text{adapt}}(\delta, y)$ are distinctly connected. For showing that a protocol $\Pi_{\text{AdaptSig}}$ UC-realizes $F_{\text{AdaptSig}}$ without assuming any knowledge about which party knows $y$, we need to ensure that signatures created via $F_{\text{AdaptSig}}$’s adaptation interface cannot be distinguished

**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 $G$ be a DLOG-hard group, then the protocol $\Pi_{\text{RLOG}_\text{Cond}}^F$ UC-realizes the ideal functionality $G_{\text{Cond}}^{\text{cond}}$, for $R = R_{\text{DLOG}}$ and $f_{\text{merge}}$ 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 $G_{\text{Cond}}$ would stay unaffected by such extensions and, hence, proofs conducted with respect to the current version of $G_{\text{Cond}}$ 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 $R_{\text{SBG}}$ relation (as used in Dillithium-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 $\Xi^R_2$ securely realizes our ideal functionality.

Two-party adaptor signature functionality. Our signature functionality $F_{\text{AdaptSig}}$ (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 $f_{\text{adapt}}$. 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 $f_{\text{adapt}}$ and global conditions functionality $G_{\text{Cond}}$.

The parametrization with $f_{\text{adapt}}$ is required to stay fully modular with respect to $G_{\text{Cond}}$. By using $G_{\text{Cond}}$ 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 $\sigma$ and the corresponding adapted signature $\sigma = f_{\text{adapt}}(\delta, y)$ are distinctly connected. For showing that a protocol $\Pi_{\text{AdaptSig}}$ UC-realizes $F_{\text{AdaptSig}}$ without assuming any knowledge about which party knows $y$, we need to ensure that signatures created via $F_{\text{AdaptSig}}$’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 FAadaptSig compute the correct signature based on y, to which end we must fix fadapt.

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) into a protocol ΠAadaptSig in Appendix E.2. Note that the construction is parametric with respect to GCond where GCond needs to support the relation R of \( \mathbb{Z}_2^{R \times \mathcal{X}} \). The security is established with the following theorem, which we prove in Appendix E.2.

**Theorem 2.** Let \( \mathbb{Z}_2^{R \times \mathcal{X}} \) 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 \( \mathcal{S}_2 \), then ΠAadaptSig UC-realizes the ideal functionality FAadaptSig.

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 \( S \) to produce a valid full signature \( \sigma \) without having access to the corresponding witness \( y \). One potential way to achieve this is inside the adaptation interface of FAadaptSig to make a call to the simulator \( S \), and let \( S \) return a fresh full signature \( \sigma' \) that is independent of the witness \( y \). Then, we can argue that \( \sigma' \) is indistinguishable from a full signature \( \sigma \), 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 \( G_{\mathcal{L}_{\text{LedgerLocks}}} \). Figure 10 shows how \( G_{\mathcal{L}_{\text{LedgerLocks}}} \) is constructed from the base ledger \( G_{\mathcal{L}_{\text{Ledger}}} \) defined in [8].

**Lock-enabling ledger functionality.** We formally describe functionality \( G_{\mathcal{L}_{\text{LedgerLocks}}} \) in Figure 11. \( G_{\mathcal{L}_{\text{LedgerLocks}}} \) builds upon \( G_{\mathcal{L}_{\text{Ledger}}} \) by adding interfaces, introducing an additional state, and refining the transaction validation check (by instantiating the predicate isValidTx). \( G_{\mathcal{L}_{\text{LedgerLocks}}} \) keeps three lists to model account management (\( L_{\text{AccId}} \)), authorization (\( L_{\text{Auths}} \)) and locking of transactions (\( L_{\text{TxsCond}} \)). The new validity predicate CheckCond operates on transactions of the form \((A, t', x')\) where \( A \) denotes the set of accounts controlling the transaction \( t' \). For checking the validity of a transaction, it checks \( L_{\text{Auths}} \) for authorization of all accounts in \( A \) and invokes CheckBase for further validity checks. In addition to the interfaces for submitting and reading transactions provided by \( G_{\mathcal{L}_{\text{Ledger}}} \), \( G_{\mathcal{L}_{\text{LedgerLocks}}} \) provides five interfaces: The account generation interface allows multiple parties to jointly generate an account (added to \( L_{\text{AccId}} \)). Although the ideal functionality models multiparty 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 \( L_{\text{TxsCond}} \). 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 \( L_{\text{TxsCond}} \) to \( L_{\text{Auths}} \). 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 can we 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 \( G_{\mathcal{L}_{\text{Ledger}}} \). 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...
LedgerLocks: A Security Framework for Blockchain Protocols Based on Adaptor Signatures

**Ideal Functionality \( \mathcal{G}_{\text{LedgerLocks}} \)**

The functionality interacts with an adversary \( S \) 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 \((\text{AccountId}, t, x, y, \perp)\), \( \mathcal{L}_{\text{Auths}} \) with entries of the form \((\text{account}, t, x)\), and \( \mathcal{L}_{\text{TxsCond}} \) with entries of the form \((\text{sid}, t, x, y)\). 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 \((\text{acc-rep}, \text{sid}, P_i, \ldots, P_m)\) to \( P_i \) and receive \((\text{acc-rep}, \text{sid}, b_i)\). If any \( b_i = 0 \), then ignore the request.
- Send \((\text{account-rep}, \text{sid}, P_1, \ldots, P_m)\) to \( S \), and upon receiving a reply \((\text{account-rep}, \text{sid}, \text{AccountId})\), add \((\text{AccountId}, P_1, \ldots, P_m)\) in \( \mathcal{L}_{\text{AccId}} \) and return \((\text{create-account}, \text{sid}, \text{AccountId})\) to all \( P_1, \ldots, P_m \) and \( P \).

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

**Lock TX:** Upon receiving \((\text{lock-tx}, \text{sid}, t, \text{AccountId}, Y)\) from \( P \), do the following:
- Extract the pair \( \alpha = \langle \text{AccountId}, \{P^*\} \rangle \) from \( \mathcal{L}_{\text{AccId}} \). If it does not exist, then ignore the request.
- Set \( \text{lock-flag} = 1 \). For \( P_j \in \{P^*\} \setminus \{P\} \), send \((\text{lock-rep}, \text{sid}, t, \alpha, Y)\) to \( P_j \) and receive \((\text{lock-rep}, \text{sid}, b_j)\) from \( P_j \). If \( b_j = 0 \), set \( \text{lock-flag} := 0 \).
- If \( \text{lock-flag} = 1 \), store \((\text{AccountId}, t, Y, \perp)\) in \( \mathcal{L}_{\text{TxsCond}} \).
- Return \((\text{lock-tx}, \text{sid}, \text{lock-flag})\) to \( P \).

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

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

**Validate Predicate:** Our predicate \( \text{CheckCond}(t, \text{state}) \) instantiates isValid\(tx(t, \text{state})\) of [8] as follows:
- Parse \( tx := (A, t') \). Then, for \( \text{AccountId}_i \in A \) set \( b_{i, j} := \langle (\text{AccountId}_i, t) \in \mathcal{L}_{\text{Auths}} \rangle \).
- Set \( b_2 := \text{CheckBase}(t, \text{state}) \).
- Return \( b_1 \land \ldots \land b_{1, j} \land b_2 \).

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, \( \text{inState}(t, \text{state}) := \exists B \in \text{state}, t \in \text{Blockify}^{-1}(B), \text{where Blockify is a predicate to parse transactions into a block} [8] \).

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 $\Pi^R_{\text{LedgerLocks}}$

<table>
<thead>
<tr>
<th>Protocol $\Pi^R_{\text{LedgerLocks}}$</th>
<th>Each party has a list $\mathcal{K}$ with entries $(P, vk)$, a list $\mathcal{P}$ with entries $(tx, vk, Y, \sigma)$, and a list $\mathcal{Q}$ with entries $(tx, vk, \sigma)$.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Account Generation:</strong> Party $P$ upon receiving (create-account, sid, $P'$) from $E$:</td>
<td>• Party $P$: Compute $sid' := (sid, P, P')$, send $(sid')$ to $P'$, and invoke $\mathcal{F}^{R,adapt}<em>{\text{AdaptSig}}$ on input (keygen, sid'). Receive (verification-key, sid, vk) from $\mathcal{F}^{R,adapt}</em>{\text{AdaptSig}}$ and store $(P, vk)$ in $\mathcal{K}$.</td>
</tr>
<tr>
<td><strong>Authorize TX:</strong> Party $P$ upon receiving (auth-tx, sid, tx, $P_b, P_1, P_2, vk$) from $E$:</td>
<td>• Party $P$: Send (auth-req, sid, $(P_b, P_1)$) to $P_b$ and $P_1$ and receive (auth-req, sid, $f_b$) from each $P_b$. If $f_b = 0$, abort.</td>
</tr>
<tr>
<td></td>
<td>• Party $P$: Compute $sid'' := (sid, P_b, P_1)$ and send $(sid'', vk)$ to $P_b$ and $P_1$.</td>
</tr>
<tr>
<td></td>
<td>• Party $P_b$ (symmetrically party $P_1$): Receive $(sid', vk)$ from $P$. Parse $sid' := (sid, P_b, P_1)$. Extract $(P_1, -vk, \sigma)$ from $\mathcal{K}$, and otherwise abort. Invoke $\mathcal{F}^{R,adapt}_{\text{AdaptSig}}$ on input (sig, sid', tx, vk, $\perp$, signature) and receive (signature, sid', $\sigma$). Store $(tx, vk, \sigma)$ in $\mathcal{Q}$, and send $\sigma$ to $P$.</td>
</tr>
<tr>
<td><strong>Lock TX:</strong> Party $P$ upon receiving (lock-tx, sid, tx, $Y, P_b, P_1, P_2, vk$) from $E$:</td>
<td>• Party $P$: Send (pre-auth-req, sid, $(P_b, P_1)$) to $P_b$ and $P_1$ and receive (auth-rep, sid, $f_b$) from each $P_b$. If $f_b = 0$, abort.</td>
</tr>
<tr>
<td></td>
<td>• Party $P$: Compute $sid'' := (sid, P_b, P_1)$ and send $(sid'', vk)$ to $P_b$ and $P_1$.</td>
</tr>
<tr>
<td></td>
<td>• Party $P_b$ (symmetrically party $P_1$): Receive $(sid', vk)$ from $P$. Parse $sid' := (sid, P_b, P_1)$. Extract $(P_1, -vk, \sigma)$ from $\mathcal{K}$, otherwise abort. Invoke $\mathcal{F}^{R,adapt}_{\text{AdaptSig}}$ on input (sig, sid', tx, Y, $pre$-signature) and receive (signature, sid', $\sigma$). Store $(tx, vk, \sigma)$ in $\mathcal{P}$, and send $\sigma$ to $P$.</td>
</tr>
<tr>
<td><strong>Release TX:</strong> Party $P$ upon receiving (release-tx, sid, tx, Y, $P, P', vk$) from $E$:</td>
<td>• Party $P$: Compute $sid' := (sid, P, P')$, extract entry $(tx, vk, Y, \sigma)$ from $\mathcal{P}$, invoke $\mathcal{F}^{R,adapt}_{\text{AdaptSig}}$ on input (adapt, sid', $\sigma$, tx, $\sigma$, v, $\sigma$)$\mathcal{Q}$, receive (adapted-signature, sid', $\sigma$), and store $(tx, vk, \sigma)$ in $\mathcal{Q}$.</td>
</tr>
<tr>
<td></td>
<td>• Involve $\mathcal{G}_{\text{edge}}$ on input (submit, sid, (tx, $\sigma$)).</td>
</tr>
<tr>
<td><strong>Signal Witness:</strong> Party $P$ upon receiving (signal-tx, sid, tx, Y, $\sigma$, vk) from $E$:</td>
<td>• Extract the entry $(tx, Y, \sigma)$ from $\mathcal{P}$, otherwise abort.</td>
</tr>
<tr>
<td></td>
<td>• Involve $\mathcal{G}_{\text{edge}}$ on input (read, sid) and receive the current state.</td>
</tr>
<tr>
<td></td>
<td>• Check if isState($(vk_1, \ldots, vk_n)$, $tx'$, $(\sigma_1, \ldots, \sigma_n)$, state) and $vk_i = \sigma_k$ for some $vk_i$, otherwise abort.</td>
</tr>
<tr>
<td></td>
<td>• Involve $\mathcal{F}^{R,adapt}_{\text{AdaptSig}}$ on input (extract, $\sigma_i$, $\sigma$, $\sigma$, $\sigma$), receive (verify, sid, $\sigma$)$\mathcal{Q}$, and return $\sigma$.</td>
</tr>
<tr>
<td><strong>Ledger Read:</strong> Party $P$ upon receiving (read, sid) from $E$, do the following:</td>
<td>• Involve $\mathcal{G}_{\text{edge}}$ on input (read, sid) and receive the current state state := $st_1[1] \ldots st_n$.</td>
</tr>
<tr>
<td></td>
<td>• state' := $st_1$. Then, for $st_2, \ldots, st_n$: Extract $(tx_1, (\sigma_1, \ldots, \sigma_{L_n})) \ldots (tx_m, (\sigma_{m,1}, \ldots, \sigma_{m,n}))$.</td>
</tr>
<tr>
<td></td>
<td>• Define new block content $\mathcal{X} := tx_{i[1]} \ldots tx_{i[n]}$. Set state' := state' $\mathcal{B}$blockify($\mathcal{X}$) and return (read, sid, state').</td>
</tr>
<tr>
<td><strong>Submit TX:</strong> Party $P$ upon receiving (submit, sid, tx) from $E$:</td>
<td>• Farse tx := $(\nu_k, \ldots, \nu_k, tx')$ and check that each $\nu_k$ is in $\mathcal{K}$. Otherwise, ignore the request.</td>
</tr>
<tr>
<td></td>
<td>• Read the state from $\mathcal{G}_{\text{edge}}$ as above. For each $\nu_k$, extract the entry $(tx, \nu_k, \sigma)$ from $\mathcal{Q}$. If any of them is missing, then abort.</td>
</tr>
<tr>
<td></td>
<td>• Involve $\mathcal{G}_{\text{edge}}$ on input (submit, sid, $(tx, (\sigma_1, \ldots, \sigma_n))$).</td>
</tr>
<tr>
<td><strong>Validate Predicate:</strong> Our predicate CheckAdapt(tx, state) instantiates isValid(tx, state) in [8] as follows:</td>
<td>• Farse tx := $(\nu_k, \ldots, \nu_k, tx')$. $(\sigma_1, \ldots, \sigma_n)$. For each pair $(\nu_k, \sigma)$, invoke $\mathcal{F}^{R,adapt}<em>{\text{AdaptSig}}$ on input (verify, sid, tx, $\nu_k$, $\nu_k$, $\perp$, signature), receive (verified, sid, $f_i$), and set $b</em>{i,k}$ := $f_i$.</td>
</tr>
<tr>
<td></td>
<td>• Set $b_2 := CheckBase((\nu_k, \ldots, \nu_k), tx')$, state) and return $b_{i,1} \wedge \ldots \wedge b_{i,n} \wedge b_2$.</td>
</tr>
</tbody>
</table>

Figure 12: Protocol $\Pi^R_{\text{LedgerLocks}}$ in the $(\mathcal{F}^{R,adapt}_{\text{AdaptSig}}, G_{\text{ledger}})$-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 $G_{\text{cond}}$ and instances of $G_{\text{LedgerLocks}}$ representing the blockchain 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 $G_{\text{LedgerLocks}}$. For the given protocol, we use two instances of $G_{\text{LedgerLocks}}$ (called $A$ and $B$ here in the following) representing the two blockchains between which assets shall be swapped. Thanks to the generality of LedgerLocks, $G_{\text{LedgerLocks}}$ 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 \( G_{\text{LedgerLocks}} \) (which is inherited from \( G_{\text{Ledger}} \)). To obtain \( A \) and \( B \) with timelock support from \( G_{\text{LedgerLocks}} \), we fix their transaction format to pairs of the form \( (tx', tl) \), where \( tl \) denotes the timelock and by defining their CheckBase predicate as follows to support the timelock check:

\[
\text{CheckBase}((A, (tx', tl)), \text{state}) = [\text{state}] \geq tl \\
\land \text{CheckBase}^C((A, tx'), \text{state})
\]

For generality, we again only specify checks for the desired feature and leave further validity checks to another ledger-specific predicate CheckBase\(^C\).

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 \( G_{\text{Cond}} \). 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 \( G_{\text{Cond}} \). Next, Alice and Bob create joint accounts on the two ledgers \( A \) and \( B \). 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 back the original owner). Importantly, \( ctx_A \) and \( rtx_A \) (along with \( ctx_B \) and \( rtx_B \), 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 \( A \) and \( B \) are independent blockchains that can provide different inclusion guarantees and proceed at different block creation rates. Inherited from \( G_{\text{Ledger}} \), the relevant behavior of a \( G_{\text{LedgerLocks}} \) instance \( C \) is determined by the parameters windowsize\(^C\) (the maximum amount of blocks that an honest user can be lacking behind the current state of the blockchain), \( \text{minTime\_window}^C \) (the minimal amount of time for including windowsize\(^C\) blocks) and \( \text{maxTime\_window}^C \) (the maximal amount of time for including windowsize\(^C\) blocks). Based on these parameters, we can define \( \#_{\text{safe}}^C = 5 \cdot \text{windowsize}^C \) to be the maximal amount of time that it takes an honest user on ledger \( C \) to include a transaction in reaction to a change in the blockchain state\(^1\).

\[w_{C_1 \rightarrow C_2}(n) = \left[ \frac{n}{\text{windowsize}^C} \right] \maxTime\_window^C\]

\[\text{safe}(h_A, h_B, n_A, n_B) = h_B < n_B + w_{C_1 \rightarrow C_2}(\#_{\text{safe}}^C) + 2 \cdot \#_{\text{safe}}^C\]

\[\land h_A < n_A + w_{C_1 \rightarrow C_2}(h_B + \#_{\text{safe}}^C - n_B) + \#_{\text{safe}}^C\]

**Figure 13: Notions for safe timelocks in atomic swaps.**

\(^1\)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 \( h + \text{windowsize} - 1 \).

**Figure 14: Setup of the Atomic Swap.** inState is a predicate checking for the inclusion of a transaction in the ledger state. \( dtx_{2U} \), \( ctx_{2U} \), and \( rtx_{2U} \) denote constructors for the deposit, claim, and refund transactions of user \( U \).

The function \( ub_{C_1 \rightarrow C_2} \) computes an upper bound on the number of blocks that can be added in ledger \( C_2 \) while \( n \) blocks are added to \( C_1 \). Using this, safe can check that given the current blockheights \( n_B \).

After submission, \( G_{\text{Ledger}} \) (and hence \( G_{\text{LedgerLocks}} \)) guarantees a valid transaction to appear in \( U \)’s view within \( 4 \cdot \text{windowsize} \) blocks.
on ledger \(B\), there is still enough time to include transactions \(dtx_A\) on \(A\) (taking up to \(ub_{A \rightarrow B}(\#_{safe}^A)\) blocks) and to include transactions \(dtx^B\) and \(ctx^A\) on \(B\) (taking up to \(2 \cdot \#_{safe}^B\) blocks) before reaching timelock \(h_B\) (first conjunct). The second conjunct ensures that given current blockheight \(n_A\) on ledger \(A\), at the point that an honest Bob learns whether \(rtx_B^A\) that they submitted after \(h_B\) has been included on \(B\) (latest after \(h_B + \#_{safe}^B\) blocks), the blockheight on \(A\) (computed by \(n_A + ub_{B \rightarrow A}(h_B + \#_{safe}^B - n_B)\)) is still at least \(\#_{safe}^A\) 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 \(A\), and once Bob sees it published on \(A\), they submit \(dtx_B\) to \(B\) and finish the setup. Note that the setup should be completed in a timely fashion. [state] \(\sim h_B\) is reached. Thanks to \(B\)'s transaction inclusion guarantees, Bob can enforce that by \(h_B + \#_{safe}^B\) 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 (\(\#_{safe}^A\)) for Bob to submit \(ctx_B\) to \(A\) 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 \(rtx_B^A\) in before \(ctx_B\). Similarly, in an attack as shown in \(\S 2\), if \(h_B\) is not set properly, Bob could outrun Alice's \(ctx_A\) transaction with \(rtx_B^A\) 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 \(\Pi^*\) (such as \(\Pi_{AtomicSwap}\)) with the help of LedgerLocks, \(\Pi^*\) can be defined in a \(\mathcal{G}_{Cond}\)-\(\mathcal{G}_{LedgerLocks}\)-hybrid world, meaning that it may interact with \(\mathcal{G}_{Cond}\) and \(\mathcal{G}_{LedgerLocks}\). Intuitively, all (adaptor-)signature-related operations of the protocols can be replaced with calls to the corresponding interfaces of \(\mathcal{G}_{LedgerLocks}\). \(\mathcal{G}_{Cond}\) allows for a logical separation between the creation of conditions and their usage to restrict transaction publication on \(\mathcal{G}_{LedgerLocks}\).

In cases where \(\Pi^*\) involves several blockchains, the description of \(\Pi^*\) uses multiple instances of \(\mathcal{G}_{LedgerLocks}\) (such as \(A\) and \(B\) 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 \(\Pi^*\) requires the creation of conditions with additional properties (that go beyond simple conditions, merged conditions, or 1-out-of-n conditions), then \(\mathcal{G}_{Cond}\)'s condition creation interface can be extended to \(\mathcal{G}_{Cond}^*\) to support the new condition types. In this case, for some relation \(R\) (known to realize \(\mathcal{G}_{Cond}\)) one needs to give a protocol \(\Pi_B\) and prove that it realizes the newly added interfaces in \(\mathcal{G}_{Cond}^*\). Alternatively, one can immediately give a protocol \(\Pi_B^*\) for some new relation \(R^*\) (that is known to be supported by an adaptor signature scheme) and show it to realize all of \(\mathcal{G}_{Cond}^*\).

#### 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 \(\Pi^*\) can be described by extending \(\mathcal{G}_{LedgerLocks}\) with \(\mathcal{G}_{Cond}\), by instantiating the CheckBase predicate that determines which transactions will be considered valid in a faithful protocol execution. Similar to how we extended \(\mathcal{G}_{Cond}\) to \(\mathcal{G}_{LedgerLocks}\), \(\mathcal{F}^*\) 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 \(\mathcal{G}_{LedgerLocks}\) functionalities.

#### Proving UC-realization

Finally, one needs to prove \(\Pi^*\) to UC-realize \(\mathcal{F}^*\). This proof should not involve cryptographic reductions but focus on how the interactions of \(\Pi^*\) with \(\mathcal{G}_{LedgerLocks}\) and \(\mathcal{G}_{Cond}^*\) are translated into interactions with \(\mathcal{F}^*\). As \(\mathcal{F}^*\) uses
LedgerLocks: A Security Framework for Blockchain Protocols Based on Adaptor Signatures

$\mathcal{G}$ is a component, the proof’s essence should lie in showing that the way that transactions are created, locked and released within $\Pi^*$ enforces the transaction inclusion logic encoded in $F^*$.

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 $\mathcal{G}$ 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 $\mathcal{G}$) 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 IJC2020-156363 with Constant Collateralist in Bitcoin-Compatible Payment Channel Networks. In ACM CCS 2019.

References


A EXTENDED GLOBAL CONDITIONS

In this section, we present an extension of the global conditions functionality $G_{Cond}$ 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 $\text{index}$, 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 $\text{ID} = (\text{IGen}, P, \text{ChSet}, V)$, where

- $\text{IGen}(1^\lambda)$: is a PPT algorithm that on input a security parameter $\lambda$ outputs a key pair $(sk, pk)$. We assume that $pk$ defines the challenge set $\text{ChSet}$.
- $P$: is a PPT algorithm composed of $P_1$ and $P_2$:
  - $P_1(sk)$: on input a secret key $sk$, outputs a commitment $R \in \mathcal{D}_{\text{rand}}$ and a state $st$.
  - $P_2(sk, R, h, st)$: on input a secret key $sk$, commitment $R \in \mathcal{D}_{\text{commit}}$, challenge $h \in \text{ChSet}$ and state $st$, outputs a response $s \in \mathcal{D}_{\text{resp}}$.
- $V(pk, R, h, s)$: is a PPT 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 $(sk, pk) \in \text{IGen}(1^\lambda)$, all $(R, st) \in P_1(sk)$, all $h \in \text{ChSet}$ and all $s \in P_2(sk, R, h, st)$, we have that $V(pk, R, h, s) = 1$.

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 $\Sigma = (\text{KGen}, \text{Sig}, Vf)$ defined as:

- $\text{KGen}(1^\lambda)$: is a PPT algorithm that on input a security parameter $\lambda$, outputs a key pair $(sk, pk)$.
- $\text{Sig}(sk, m)$: is a PPT algorithm that on input a secret key $sk$ and message $m \in \{0, 1\}^*$, outputs a signature $\sigma$.
- $Vf(pk, m, \sigma)$: is a PPT algorithm that on input a public key $pk$, message $m \in \{0, 1\}^*$ and signature $\sigma$, outputs a bit $b$.

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

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

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 $\Sigma$ is EUF-CMA secure if for every PPT adversary $\mathcal{A}$ there exists a negligible function $\text{negl}$ such that

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

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

**Ideal Functionality $G_{Cond}$**

The functionality interacts with an adversary $S$ 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 $f_{\text{merge}}$ for which the following invariant holds: $(Y_1, y_1) \in R \wedge (Y_2, y_2) \in R \Rightarrow (f_{\text{merge}}(\text{stmt}, R, (Y_1, Y_2)), f_{\text{merge}}(\text{wit}, R, y_1, y_2)) \in R$.

**Individual Conditions:** Upon receiving (create-ind-cond, 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 (create-cond-ind, sid, $Y$) to $P$ and $S$.

**1-out-of-n Conditions:** Upon receiving (create-1-of-n-cond, sid, $(Y, y)$) from some party $P$, do the following:

- If $(Y, y) \notin R$, then ignore the request. Otherwise, continue.
- For all $i \in [n] \setminus \{i\}$, sample random $(Y_i, y_i) \in R$.
- Set $Y^* := (Y_1, \ldots, Y_{\text{index}} := Y_i, \ldots, Y_n)$.
- For all $P^* \in \mathcal{P}$, send $(\text{join}-1-of-n-cond, sid, P, Y^*)$ and receive back $(\text{joined}-1-of-n-cond, sid, P_i)$. If any $P_i$ fails, then abort. Otherwise, continue.
- Set $\mathcal{L}[Y^*] := (\ldots, y_{\text{index}} := y_i, \ldots)$.
- Send (create-1-of-n-cond, sid, $Y^*$) to $P$ and $S$.

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

**Open Conditions:** Upon receiving (open-cond, sid, $(Y', y')$) from some party $P^*$, set $b := (\mathcal{L}[Y^*], Y', y')$ and send (opened-cond, sid, $b$) to $P^*$ and $S$.
The protocol is parameterized by group description \((G, q, g)\), and the corresponding discrete logarithm (DLOG) relation \(\mathcal{R}_{\text{DLOG}}\) over it, i.e., \((Y, y) \in \mathcal{R}_{\text{DLOG}} \iff Y = g^y\).

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

**1-out-of-\(n\) Conditions:**

- Party \(P\) upon receiving (create-1-of-\(n\)-cond, \((T, y)\), index, \(n, \{P_i\}\)) from \(E\):
  - For all \(P^* \in \{P_i\}\), send \((n, \{P_i\})\) to \(P^*\).
  - Party \(P^*\) upon receiving \((n, \{P_i\})\) from \(P\):
    - For all \(j \in [n]\), sample \(s_i[j] \leftarrow \mathbb{Z}_q\).
    - For all \(j \in [n]\) and \(j \neq \text{index}\), compute \(h_i[j] := g^{s_i[j]}\).
    - Send vector \(h_i := \{h_i[j]\}_{j \in [n]}\) to \(P\) and \(\{P_i\} \setminus \{P^*\}\).

- Party \(P\) upon receiving \(h_i\) from \(P^* \in \{P_i\}\):
  - For all \(j \in [n]\), sample \(s_i[j] \leftarrow \mathbb{Z}_q\).
  - For all \(j \in [n]\) and \(j \neq \text{index}\), compute \(h_i[j] := g^{s_i[j]}\).
  - Set \(\text{index} := Y\) and \(s_i[\text{index}] := y\).
  - Compute \(\pi_{\text{index}} \leftarrow \text{NIZK.PF}((\exists i, j) \mid \text{index} = g^{s_i[j]} \land (j \in [n] \land j \neq \text{index}, h_i[j] = f_j \cdot g^{s_i[j]}))\), \(2\cdot\text{index})\).
  - Return \((E, \{f_j\}_{j \in [n]}, \pi_{\text{index}}, y)\).

**Merged Conditions:** Party \(P\) upon receiving (create-and-cond, sid, \((Y, y)\), \((T, y')\)) from \(E\), compute \(Y' := Y \cdot Y_2\) and return \(Y'\).

**Open Conditions:** Party \(P\) upon receiving (open-cond, sid, \((Y', y')\)) from \(E\), return \((Y', y') \in \mathcal{R}_{\text{DLOG}}\).

**Definition of** \(\text{merge}\):

\[
\text{merge}(\text{stmt}, R, (Y_1, Y_2)) := Y_1 \cdot Y_2
\]

\[
\text{merge}(\text{wit}, R, (y_1, y_2)) := y_1 + y_2
\]

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

<table>
<thead>
<tr>
<th>(\text{SignForge}_{\Sigma_x}(\lambda))</th>
<th>(\mathcal{O}_S(m))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Q \leftarrow 0)</td>
<td>(\sigma \leftarrow \text{Sig}(sk, m))</td>
</tr>
<tr>
<td>((sk, pk) \leftarrow \text{KGen}(1^\lambda))</td>
<td>(Q := Q \cup {m}) return (\sigma)</td>
</tr>
<tr>
<td>((m, \sigma) \leftarrow \mathcal{O}_S^{\text{DLOG}}(pk))</td>
<td></td>
</tr>
<tr>
<td>return ((m \notin Q \land \text{Vf}(pk, m, \sigma)))</td>
<td></td>
</tr>
</tbody>
</table>

The advantage of the adversary \(A\) playing the game \(\text{StrongSigForge}\) is defined as follows:

\[
\text{Adv}_{\Sigma_x}^{\text{StrongSigForge}}(A) = \Pr[\text{StrongSigForge}_{\Sigma_x}(\lambda) = 1].
\]

**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 \(\Sigma_2 = (\text{Setup}, \text{KGen}, \Pi_{\text{Sig}}, \Pi_{\text{KAgg}}, \Pi_{\text{Vf}})\)

- **Setup** \((1^\lambda)\):
  - is a PPT algorithm that, on input a security parameter \(\lambda\), outputs public parameters \(pp\).
  - \(\Pi_{\text{KGen}}(pp)\): a PPT algorithm that, on input public parameters \(pp\), outputs a key pair \((sk, pk)\).
  - \(\Pi_{\text{Sig}}(sk, sk_1, m)\): an interactive PPT protocol that, on input secret keys \(sk_i\) from party \(P_i\) with \(i \in \{0, 1\}\) and common values messages \(m \in \{0, 1\}^\ast\) and public keys \(pk_0, pk_1\), outputs a signature \(\sigma\).
  - \(\Pi_{\text{KAgg}}(pk_0, pk_1)\): a DPT algorithm that on input two public keys \(pk_0, pk_1\), outputs an aggregated public key apk.
  - \(\Pi_{\text{Vf}}(apk, m, \sigma)\): a DPT algorithm that, on input a public key \(apk\), message \(m \in \{0, 1\}^\ast\) and signature \(\sigma\), outputs a bit \(b\).

We can define completeness for \(\Sigma_2\) in a natural way: a two-party signature scheme with aggregatable public keys \(\Sigma_2\) satisfies completeness, if for all public parameters \(pp \leftarrow \text{Setup}(1^\lambda)\), key pair \((sk, pk)\) \(\Pi_{\text{KGen}}(pp)\) and messages \(m \in \{0, 1\}^\ast\), the protocol \(\Pi_{\text{Sig}}(sk, sk_1, m)\) outputs a signature \(\sigma\) to both parties \(P_0, P_1\), such that \(\Pi_{\text{Vf}}(apk, m, \sigma) = 1\), where \(apk := \Pi_{\text{KAgg}}(pk_0, pk_1)\).

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 \(\text{SigForge}_b^{\Sigma_2}\), 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 \(\Sigma_2\) is 2-EUF-CMA secure if for every PPT adversary \(A\) there exists a negligible function \(\text{negl}\) such that, for \(b \in \{0, 1\}\),

\[
\Pr[\text{SigForge}_b^{\Sigma_2}(\lambda) = 1] \leq \text{negl}(\lambda),
\]

where the experiment \(\text{SigForge}_b^{\Sigma_2}\) is defined as follows:
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 \( \text{f}_{\text{shift}}: \mathcal{D}_{\text{rand}} \times \mathcal{L}_R \rightarrow \mathcal{D}_{\text{rand}} \) takes as input a commitment value \( R \in \mathcal{D}_{\text{rand}} \) of the identification scheme and a statement \( Y \in \mathcal{L}_R \) of the hard relation, and outputs a new commitment value \( R' \in \mathcal{D}_{\text{rand}} \). We assume that the inverse of this function is well-defined.
- The adaptation function \( \text{f}_{\text{adapt}}: \mathcal{D}_{\text{resp}} \times \mathcal{D}_w \rightarrow \mathcal{D}_{\text{resp}} \) takes as input a pre-signature value \( \hat{s} \in \mathcal{D}_{\text{resp}} \) (which corresponds to the response value of the identification scheme) and a witness \( Y \in \mathcal{D}_w \) of the hard relation \( R \), and outputs a new value \( s \in \mathcal{D}_{\text{resp}} \).
- The witness extraction function \( \text{f}_{\text{ext}}: \mathcal{D}_{\text{resp}} \times \mathcal{D}_{\text{resp}} \rightarrow \mathcal{D}_w \) takes as input two response values \( \hat{s}, s \in \mathcal{D}_{\text{resp}} \) and outputs a witness \( y \in \mathcal{D}_w \).
- The randomness combining function \( \text{f}_{\text{com-rand}}: \mathcal{D}_{\text{rand}} \times \mathcal{D}_{\text{rand}} \rightarrow \mathcal{D}_{\text{rand}} \) that takes as input two randomness \( \hat{R}_0, R_1 \in \mathcal{D}_{\text{rand}} \) and outputs a new combined randomness \( R \in \mathcal{D}_{\text{rand}} \).
- The signature combining function \( \text{f}_{\text{com-sig}} \), that takes as input two partial signatures and returns a new combined signature.
- The randomness exchange protocol \( \Pi_{\text{Rand-Exc}} \).
- The partial signature exchange protocol \( \Pi_{\text{Exchange}} \).

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 \( \mathbb{Z}_2^R \) satisfies two-party pre-signature correctness if for every \( \lambda \in \mathbb{N} \), any message \( m \in \{0,1\}^* \), and every statement/witness pair \( (Y, y) \in R \), the following holds:

\[
\begin{align*}
\text{PreVf}(\text{apk}, m, \hat{\sigma}) &= 1 \\
\text{Pr} \quad \text{Vf}(\text{apk}, m, \sigma) &= 1
\end{align*}
\]

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 \( \mathbb{Z}_2^R \) is 2-aEUF-CMA secure if for every PPT adversary \( \mathcal{A} \) there exists a negligible function \( \text{negl} \), such that: \( \text{Pr}[\text{aSigForge}_{\mathbb{Z}_2^R}(\lambda) = 1] \leq \text{negl}(\lambda) \), where the experiment \( \text{aSigForge}_{\mathbb{Z}_2^R} \) is defined as follows:

\[
\begin{align*}
\text{SigForge}_{\mathbb{Z}_2^R}(\lambda) &\quad \mathcal{O}^{\text{b}}_{\text{F}}(m) \\
\text{Q} \quad \mathcal{Q} = Q \cup \{m\} \\
\text{pp} \quad \text{Setup}(\lambda^3) \\
\sigma \quad \Pi_{\text{sig}}^{\lambda \cdot \mathbb{Z}_2^R}(\text{apk}, \text{pk}_i, m) \\
\text{return} \quad \sigma
\end{align*}
\]

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 \( \mathbb{Z}_2^R \) satisfies two-party pre-signature adaptability if for any \( \lambda \in \mathbb{N} \), any statement/witness pair \( (Y, y) \in R \), any public keys \( \text{pk}_0, \text{pk}_1 \) and any pre-signature \( \hat{\sigma} \in \{0,1\}^* \) satisfying \( \text{PreVf}(\text{ap}\text{k}, m, \hat{\sigma}) = 1 \), where \( \text{apk} := \text{KAgg}(\text{pk}_0, \text{pk}_1) \), we have: \( \text{Pr}[\text{Vf}(\text{apk}, m, \hat{\sigma}, y) = 1] = 1 \).

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 \( \mathbb{Z}_2^R \) is two-party witness extractable if for every PPT adversary \( \mathcal{A} \), there exists a negligible function \( \text{negl} \) such that the following holds:
Π_{PreSig}(sk_x, pk_y, m, y)
Parse pk_y = ((Y, ppC, crs), pk_y), t ∈ \{0, 1\}
(R_t, st_t, R_{t-1}) ← \Pi_{Rand-Exc}(sk_x, sk_{t-1}, i) (ppC, crs)
PreVf(m, Y, \hat{\sigma} := (h, s))
PreVf(m, Y, \hat{\sigma} := (h, s))
R_{pre} := H(b, h)
return h = H(f_{adapt}(s, y), m)
Adapt apk, \sigma := (h, s), \hat{\sigma} := (h, s), Y
return f_{ext}(s, \hat{\sigma})

Figure 18: Two-party adaptor signature scheme with aggregatable public keys ΣR2 with respect to Σ2 and hard relation R.

Pr[aWitExtb_{\SigmaR2}(\lambda) = 1] ≤ negl(λ)
where the experiment aWitExtb_{\SigmaR2} is defined as follows

<table>
<thead>
<tr>
<th>aWitExtb_{\SigmaR2}(\lambda)</th>
<th>O_{\Pi}(m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q = 0, pp ← Setup(1^λ)</td>
<td>Q = Q ∪ (m)</td>
</tr>
<tr>
<td>(sk_x, pk_y, pk_{t-1}) ← KGen(pp)</td>
<td>σ ← \Pi_{\Pi_x}^{crs}(sk_x, pk_y, m)</td>
</tr>
<tr>
<td>(sk_{t}, pk_{t}) ← \Pi_{\Pi_x}^{crs}(pp, pk_{t-1})</td>
<td>return σ</td>
</tr>
<tr>
<td>(m, Y) ← A0_{\Pi}(sk_{t}, pk_{t-1}, \cdot) (m, Y)</td>
<td>Q = Q ∪ (m)</td>
</tr>
<tr>
<td>σ ← \Pi_{\Pi_x}^{crs}(sk_x, pk_y, \cdot) (m, Y)</td>
<td>\hat{\sigma} := \Pi_{\Pi_y}^{crs}(sk_x, pk_y, m, Y)</td>
</tr>
<tr>
<td>\hat{\sigma} := \Pi_{\Pi_x}^{crs}(sk_{t}, pk_{t-1}, \cdot) (m, Y)</td>
<td>return \hat{\sigma}</td>
</tr>
<tr>
<td>apk = Kagg(pk_y, pk_t)</td>
<td>y' = Ext(apk, σ, \hat{\sigma}, Y)</td>
</tr>
<tr>
<td>y' = Ext(apk, σ, \hat{\sigma}, Y)</td>
<td>return (m \notin Q ∧ (Y, y') \notin R ∧ Vf(apk, m, σ))</td>
</tr>
</tbody>
</table>

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) ∈ 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 ∈ NP (with witness relation R) is a tuple of PPT algorithms NIZK = (PGen, P, V), such that:

PGen(1^λ): on input a security parameter λ, outputs a common reference string crs.

P(crs, x, w): on input a common reference string crs, a statement x and a witness w, outputs a proof π.
V(crs, x, π): 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) ∈ R we have that
Pr[crs ← PGen(1^λ), π ← P(crs, x, w): V(crs, x, π) = 1] = 1.
Soundness. For every x \notin L, and every adversary A, we have that
Pr[crs ← PGen(1^λ), π ← A(crs, x): V(crs, x, π) = 1] ≤ negl(λ).

Zero-Knowledge. There exists a PPT algorithm S = (S_1, S_2) such that for every PPT adversary A,
Adv_{\SigmaR2}^{\Pi}(\lambda) := \left| Pr[crs ← PGen(1^λ); A(crs, \cdot) (crs) = 1] - Pr[(crs, \tau) ← S_2(crs, \tau, x) when (x, w) ∈ R] \right|
is negligible in λ, where O(crs, τ, \cdot, \cdot) is an oracle that outputs \bot on input (x, w) when (x, w) \notin R and outputs π ← S_2(crs, τ, x) when (x, w) ∈ 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
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 $\Sigma_2^{\mathcal{RS}}$ 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

$$[\Pi_{\text{Sig}}(sk_0, sk_1)(m)] \text{ and } [\text{Adapt}(apk, \Pi_{\text{PreSig}}(sk_0, sk_1)(m), Y, y)]$$

are identically distributed. Here, $pp \leftarrow \text{Setup}(1^\lambda)$, $(sk_0, pk_0) \leftarrow \text{KGen}(pp)$, $(sk_1, pk_1) \leftarrow \text{KGen}(pp)$, and $apk \leftarrow \text{KAgg}(pk_0, pk_1)$.

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) $\Sigma_2^{\mathcal{RS}}$ 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 $\Sigma_2^{\mathcal{RS}}$ 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 $\Lambda^R = (\text{Setup}, \Pi_{\text{AccGen}}, \Pi_{\text{AuthTx}}, \Pi_{\text{LockTx}}, \text{RelTx}, \text{SigTx}, \text{VerTx})$ of efficient protocols, run between parties $P_b$ for $b \in \{0, 1\}$, and algorithms defined as follows:

- **Setup($1^\lambda$)**: is a PPT algorithm that on input a security parameter $\lambda$, outputs public parameters $pp$.
- **$\Pi_{\text{AccGen}}(pp)$**: is an interactive protocol that on input public parameters, outputs secret keys $(sk_0, sk_1)$ and a verification key $vk$.
- **$\Pi_{\text{AuthTx}}(sk_0, sk_1)$** (vk, tx): is an interactive protocol that on input a verification key $vk$ and transaction $tx$, outputs a signature $\sigma$.
- **$\Pi_{\text{LockTx}}(sk_0, sk_1)$** (vk, tx, Y): is an interactive protocol that on input a verification key $vk$, transaction $tx$ and condition $Y$, outputs a pre-signature $\hat{\sigma}$.
- **RelTx(vk, 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 $\sigma$.
- **SigTx(vk, \sigma, \hat{\sigma}, Y)**: is a DPT algorithm that on input a verification key $vk$, signature $\sigma$, pre-signature $\hat{\sigma}$ and statement $Y \in L_R$, outputs a witness $y$ s.t. $(Y, y) \in R$, or $\perp$.
- **VerTx(vk, tx, \sigma)**: is a DPT algorithm that on input a verification key $vk$, transaction $tx$ and signature $\sigma$, 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 $\Pi_{\text{AccGen}}$ corresponds to key generation, transaction authorization $\Pi_{\text{AuthTx}}$ corresponds to full signature generation, and transaction locking $\Pi_{\text{LockTx}}$ corresponds to pre-signature generation using the adaptor signature scheme. Similarly, releasing transaction $\text{RelTx}$ and signaling transaction $\text{SigTx}$ correspond to adaptation and extraction operations, respectively. Our instantiation is given in Figure 19.

![Figure 19: Instantiation of LedgerLockTx $\Lambda^R$ using two-party adaptor signature scheme $\Sigma_2^{\mathcal{RS}}$.](image)

**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 $\Pi_{\text{AuthTx}}$ and $\text{RelTx}$ are identically distributed.

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

$$[\Pi_{\text{AuthTx}}(sk_0, sk_1)(vk, tx)] \text{ and } [\text{RelTx}(vk, \Pi_{\text{LockTx}}(sk_0, sk_1)(vk, tx, Y), y)]$$
are identically distributed. Here \( pp \leftarrow \text{Setup}(\lambda) \) and \((\text{sk}_0, \text{sk}_1, \text{vk}) \leftarrow \Pi_{\text{AccGen}}(pp)\).

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

**Lemma 2.** Let \( Z_2^{RX} \) 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 \( G_{\text{Clock}} \) of Badertscher et al. [8], 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 (clock-read, sid). This value is updated only once all honest parties send (clock-update, sid) request to the functionality. The functionality is given in Figure 20.

**Ledger functionality.** Lastly, we describe the ledger ideal functionality \( G_{\text{Ledger}} \) of Badertscher et al. [8], which is depicted in Figure 34. The functionality makes use of the clock functionality \( G_{\text{Clock}} \) defined in Appendix D, and is parameterized by four algorithms Validate, ExtendPolicy, Blockify, and predict-time, along with two parameters \( \text{windowSize}, \text{Delay} \in \mathbb{N} \). 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 \( G \) be a DLOG-hard group, then the protocol \( \Pi_{\text{UC}}^{\text{RLOG}} \) realizes the ideal functionality \( G_{\text{Cond}}^{R_{\text{merg}}} \), for \( R = R_{\text{LOG}} \) and \( f_{\text{merg}} \) 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 \( \Pi_{\text{Cond}} \).

Hybrid \( \mathcal{H}_1 \): All calls to non-interactive zero-knowledge proof system NIZK are simulated using the simulator \( S_{\text{NIZK}} \) for the corresponding language \( L \).

Hybrid \( \mathcal{H}_2 \): For the set of corrupted parties \( C \), if the adversary outputs (open-cond, \((Y, y')\)), such that \((Y, y') \in R_{\text{LOG}} \), for the condition \( Y \) created by party \( P \) via (create-ind-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 (open-cond, \((Y, y')\)), such that \((Y, y') \in R_{\text{LOG}} \), for the condition \( Y \) created by party \( P \) via (create-merged-cond, \((Y, Y')\)) and \( P \notin C \), then the experiment aborts by outputting fail_3.

**Simulation \( S \):** The simulator \( S \) simulates the honest parties as in the previous hybrid, except that its actions are dictated by the interaction with the ideal functionality \( G_{\text{Cond}} \). More precisely, the simulator proceeds as the execution of \( \mathcal{H}_4 \) by simulating the view of the adversary appropriately as it receives messages from the ideal functionality \( G_{\text{Cond}} \). 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

\[
\text{EXEC}_{\mathcal{H}_0, \mathcal{A}, \mathcal{E}} \approx \text{EXEC}_{\mathcal{H}_1, \mathcal{A}, \mathcal{E}}.
\]

**Proof.** The proof follows directly from the zero-knowledge property of the non-interactive zero-knowledge proof system NIZK, for which the simulator \( S_{\text{NIZK}} \) is guaranteed to exist.

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

\[
\text{EXEC}_{\mathcal{H}_1, \mathcal{A}, \mathcal{E}} \approx \text{EXEC}_{\mathcal{H}_2, \mathcal{A}, \mathcal{E}}.
\]

**Proof.** Let \( \text{fail}_1 \) 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 \( \Pr[\text{fail}_1 | \mathcal{H}_1] \geq \frac{1}{\text{poly}(\lambda)} \). To show that the probability of \( \text{fail}_1 \) 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 \( i \in [1, \lambda] \), where \( s \in \text{poly}(\lambda) \) 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 \( \text{fail}_1 \) 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 \( \text{fail}_1 \) happens it means that \((Y^*, y^*) \in R_{\text{LOG}} \), for the condition \( Y^* \), and \( P \notin C \). This implies that the reduction succeeds in breaking the DLOG. By assumption this happens with probability at least \( \frac{1}{\text{poly}(\lambda)} \), which is a contradiction and proves that \( \Pr[\text{fail}_1 | \mathcal{H}_1] \leq \text{negl}(\lambda) \).

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

\[
\text{EXEC}_{\mathcal{H}_2, \mathcal{A}, \mathcal{E}} \approx \text{EXEC}_{\mathcal{H}_3, \mathcal{A}, \mathcal{E}}.
\]

**Proof.** The proof of this lemma is analogous to that of Lemma 4, but we need to additionally account for \( \text{index}^* \). More precisely, since \((\pi[\text{index}^*], y^*) \in R_{\text{LOG}} \), we know that \( \pi[\text{index}^*] = gy^* \).
For each session we verify that the adaptation algorithm of two-party adaptor signature coincides with an additional index $H$. For each identity $P := (p, d, sid) \in \mathcal{P}$ it manages variable $dp$. For each pair $(F, sid) \in F$ it manages variable $d_{F, sid}$ (all integer variables are initially 0).

**Synchonization:**
- Upon receiving (clock-update, $sid_C$) from some party $P \in \mathcal{P}$ set $dp = 1$, execute **Round-Update** and forward (clock-update, $sid_C$, $P$) to $S$.
- Upon receiving (clock-update, $sid_C$) from some functionality $F$ in a session sid such that $(F, sid) \in F$ set $d_{F, sid} = 1$, execute **Round-Update** and return (clock-update, $sid_C$, $F$) to this instance of $F$.

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

---

**Ideal Functionality $G_{\text{Clock}}$**

The functionality manages the set $\mathcal{P}$ of registered identities, i.e., parties $P := (p, d, sid)$. 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 $\tau_{sid}$. For each identity $P := (p, d, sid) \in \mathcal{P}$ it manages variable $dp$. For each pair $(F, sid) \in F$ it manages variable $d_{F, sid}$ (all integer variables are initially 0).

**Synchronization:**
- Upon receiving (clock-update, $sid_C$) from some party $P \in \mathcal{P}$ set $dp = 1$, execute **Round-Update** and forward (clock-update, $sid_C$, $P$) to $S$.
- Upon receiving (clock-update, $sid_C$) from some functionality $F$ in a session sid such that $(F, sid) \in F$ set $d_{F, sid} = 1$, execute **Round-Update** and return (clock-update, $sid_C$, $F$) to this instance of $F$.

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

---

Figure 20: Ideal functionality $G_{\text{Clock}}$ [6, 24].

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

**Lemma 6.** For all PPT distinguishers $E$ it holds that

$$\text{EXEC}_{\mathcal{H}_c, \mathcal{A}, E} \approx \text{EXEC}_{\mathcal{H}_s, \mathcal{A}, E}.$$  

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

**Lemma 7.** For all PPT distinguishers $E$ it holds that

$$\text{EXEC}_{\mathcal{H}_c, \mathcal{A}, E} \approx \text{EXEC}_{\mathcal{G}_{\text{Clock}}, \mathcal{A}, E}.$$  

**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) $\Sigma_{\mathcal{G}_{\text{Clock}}}$ into a protocol $\Pi_{\text{AdaptSig}}$. We consider parties $P_b$ for $b \in \{0, 1\}$ running $\Pi_{\text{AdaptSig}}$, and upon each request we verify that sid $= (P_b, P_1-b, \text{sid}')$ for some sid’, and if not, then ignore the request.

- Upon receiving (keygen, sid) from $E$, generate keys $(sk_b, pk_b) \leftarrow \Sigma_2.KGen(pp)$, for some public parameters pp, store $sk_b$, and output (verification-key, sid, $\sigma := (pk_b, pk_1-b)$).
- Upon receiving (sign, sid, $m$, $v$, $\sigma$, signature) from $E$, parse $\sigma$ as $(pk_b, pk_1-b)$, execute the protocol $\sigma \leftarrow \Sigma_2:\Pi_{\text{Sig}}(sk_b, sk_1-b, (pk_b, pk_1-b, m))$ and output (signature, sid, $\sigma$).
- Upon receiving (sign, sid, $m$, $v$, $\sigma$, signature) from $E$, parse $\sigma$ as $(pk_b, pk_1-b)$, and upon each request (clock-update, $sid_C$) from any participant (including the environment on behalf of a party, the adversary, or any ideal—shared or local—functionality) return (clock-read, $sid_C$, $\tau_{sid}$) to the requestor (where sid is the session identifier of the calling instance).

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

**Theorem 2.** Let $\Sigma_{\mathcal{G}_{\text{Clock}}}$ 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 $\Sigma_2$, then $\Pi_{\text{AdaptSig}}$ UC-realizes the ideal functionality $\mathcal{F}_{\text{AdaptSig}}$.

**Proof.** We give a proof by contradiction. Assume that $\Sigma_{\mathcal{G}_{\text{Clock}}}$ does not realize $\mathcal{F}_{\text{AdaptSig}}$, i.e., there exists an environment $E$ that can differentiate whether it is interacting with $\mathcal{F}_{\text{AdaptSig}}$ and $S$ in the ideal world, or with $\Sigma_{\mathcal{G}_{\text{Clock}}}$ and $\mathcal{A}$ in the real world. We show that $\Sigma_{\mathcal{G}_{\text{Clock}}}$ violates the definition of secure two-party adaptor signature scheme from Appendix B. Since the environment $E$ succeeds for any $S$, it also succeeds for the following generic $S$, 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 $E$ is forwarded to $\mathcal{A}$, and any output from $\mathcal{A}$ is copied to $S$’s output (to be read by $E$).**
2. **(Upon receiving a message (keygen, sid) from $\mathcal{F}_{\mathcal{G}_{\text{Clock}}, \text{AdaptSig}}$:***
• If sid is not of the form \((P_{1-b}, P_b, \text{sid'})\), then \(S\) ignores the request.
• Else, \(S\) runs \((sk_{1-b}, pk_{1-b}) \leftarrow \Sigma_2 \text{KGen}(pp)\), sets \(v := (pk_{1-b}, pk_b)\), records the tuple \((\text{sid}, sk_{1-b}, \sigma)\), and returns \((\text{verification--key}, \text{sid}, v)\) to \(F_{\text{Adapt}}^{R_E}\).

(3) Upon receiving a message \((\text{sign}, \text{sid}, m, v, Y, \text{type})\) from \(F_{\text{Adapt}}^{R_E}\):
• If sid is not of the form \((P_{1-b}, P_b, \text{sid'})\) and \(v\) has not been previously recorded, then \(S\) ignores the request.
• If type = signature, then \(S\) parses \(v := (pk_{1-b}, pk_b)\), generates a run of the protocol \(\sigma \leftarrow \Sigma_2 \Pi_A^{\text{Sig}(sk_{1-b})} (pk_{1-b}, pk_b, m, Y)\) (by simulating the honest party \(P_{1-b}\)), and sends \((\text{signature}, \text{sid}, m, \sigma)\) to \(F_{\text{Adapt}}^{R_E}\).
• If type = pre-signature, then \(S\) parses \(Y := Y'\) and \(v := (pk_{1-b}, pk_b)\), generates a run of the protocol \(\sigma \leftarrow \Sigma_2 \Pi_A^{\text{PreSig}(sk_{1-b})} (pk_{1-b}, pk_b, m, Y, \text{type})\) (by simulating the honest party \(P_{1-b}\)), and sends \((\text{signature}, \text{sid}, m, \sigma)\) to \(F_{\text{Adapt}}^{R_E}\).

(4) Upon receiving \((\text{verify}, \text{sid}, m, \sigma, v, Y, \text{type})\) from \(F_{\text{Adapt}}^{R_E}\):
• If type = signature, then \(S\) parses \(v := (pk_{1-b}, pk_b)\), computes \(\text{apk} := \Sigma_2 \text{KAgg}(pk_{1-b}, pk_b)\), sets \(\phi := \Sigma_2 \text{Vf}(\text{apk}, m, \sigma)\), and returns \((\text{verified}, \text{sid}, m, \phi)\) to \(F_{\text{Adapt}}^{R_E}\).
• If type = pre-signature, then \(S\) parses \(Y := Y'\) and \(v := (pk_{1-b}, pk_b)\), computes \(\text{apk} := \Sigma_2 \text{KAgg}(pk_{1-b}, pk_b)\), sets \(\phi := \Sigma_2 \Pi_A^{\text{PreVf}(apk, m)}\), and returns \((\text{verified}, \text{sid}, m, \phi)\) to \(F_{\text{Adapt}}^{R_E}\).

(5) When \(A\) corrupts some party \(P\), then \(S\) corrupts \(P\) in the ideal world. If \(P\) is the signer, then \(S\) 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 \(\Sigma_2\) is both complete and consistent, and construct an attacker \(B\) that breaks the unforgeability. \(B\) runs a simulated copy of \(E\) and simulates the interactions with \(S\) and \(F_{\text{Adapt}}^{R_E}\). Analogous to \(S\), \(B\) also runs a simulated copy of \(A\). However, in the first activation, instead of running \(\Sigma_2 \text{KGen}\) to generate the key, \(B\) gives to \(A\) the verification key \(pk_{1-b}\) that it has received as an input from its own challenger \((\text{with } b \in \{0,1\})\) defines which of the two parties is corrupted). Whenever \(B\) needs to provide \((\text{pre-signature})\) on a message \(m\), \(B\) asks its oracles to \((\text{pre-signature})\) and obtains \((\text{pre-signature})\) \(\sigma\). Moreover, \(B\) and \(A\) jointly generate \((\text{pre-signature})\) \(\sigma\) on the challenge message \(m^*\) (provided by \(A\)), where \(B\) 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 \(E\) activates some party with \((\text{verify}, \text{sid}, m^*, \sigma, v, Y, \text{type})\), where \(\text{type} = \text{signature}\), \(B\) checks whether \((m^*, \sigma)\) constitutes a valid forgery (i.e., that \(m^*\) is the challenge message and it has never been signed before and \(\Sigma_2 \text{Vf}(\text{apk}, m^*, \sigma) = 1\), for \(\text{apk} := \Sigma_2 \text{KAgg}(pk_{1-b}, pk_b)\), where \(v := (pk_{1-b}, pk_b)\) and \(pk_b\) is the verification key of \(A\)). If \((m^*, \sigma)\) is a valid forgery, then \(B\) outputs \((\sigma)\) as its own forgery and halts. Otherwise, it continues the simulation. If \(A\) asks to corrupt the honest signer \(P_{1-b}\), then \(B\) halts with a failure.

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

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

Lastly, we observe that pre-signature adaptability is captured within the adaptation interface of the ideal functionality \(F_{\text{Adapt}}^{R_E}\), which makes use of the global ideal functionality \(G_{\text{Const}}\) and parameterized deterministic adaptation function \(f_{\text{Adapt}}\). More precisely, a valid pre-signature \(\sigma\) w.r.t. some condition \(Y := Y'\) can be adapted into a valid signature, i.e., \(\sigma := f_{\text{Adapt}}(\hat{\sigma}, y)\), using the witness \(y\) that satisfies \((Y, y) \in R\).

This concludes the proof of Theorem 2.

\[\square\]

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) \(\Sigma_2\) is perfectly unlinkable.

**Proof.** We prove this lemma by considering the deterministic adaptation function \(f_{\text{Adapt}}\) given in Appendix B for the generic transformation from an identification scheme-based two-party signature to an adaptor signature. \(f_{\text{Adapt}}\) : \(D_{\text{Resp}} \times D_w \rightarrow D_{\text{Resp}}\), takes as input a pre-signature value \(\hat{s} \in D_{\text{Resp}}\) (which corresponds to the response value of the identification scheme) and a witness \(y \in D_w\) of the hard relation \(R\), and outputs a new value \(s \in D_{\text{Resp}}\). Then, the adaptation function of \(\Sigma_2\) is defined as \(\text{Adapt}(\text{apk}, \hat{s}, y) := f_{\text{Adapt}}(\hat{s}, y)\). Since a freshly computed signature from an identification scheme is some
response value $s' \in D_{\text{resp}}$, it is immediate that the adapted signature $s$ and the fresh signature $s'$ come from the same distribution $D_{\text{resp}}$. □

### 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.

---

**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 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 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 $(\mathcal{A}, tx')$ where $\mathcal{A}$ 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 $tx'$ to $(tx'', tI)$.

We will further refine the format of $tx''$ to be of the form $(id, in, out)$ with $id$ being a transaction identifier, $in$ being a vector of inputs and $out$ being a vector of outputs. Inputs $in \in \mathcal{A}$ are of the form $(id_{\text{out}}, r, tI)$ where $(id_{\text{out}}, r)$ refers to the output that the input is spending (with $id_{\text{out}}$ being the transaction id and $r$ the offset in the transactions output vector), and $tI$ 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 $out \in \mathcal{A}$ 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 CheckBase 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 $\text{getOutput}$ that accesses information of a transaction input in the blockchain state. Given an input $in$ and the blockchain state, $\text{getOutput}$ 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.

---

**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 delayed (and replies given back to $\mathcal{A}$). The simulator maintains locally a list of keys $\mathcal{K}_{\mathcal{P}}$, list of pre-signed transactions $\mathcal{P}_{\mathcal{P}}$ and list of signed transactions $\mathcal{Q}_{\mathcal{P}}$ for an honest party $\mathcal{P}$. 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 $s' := (s, P, P')$, forward (keygen, $s'$) 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 $(P', \mathcal{K}_{\mathcal{P}})$ in $\mathcal{K}_{\mathcal{P}}$.
- Upon receiving (auth-req, sid, $tx, \alpha$) from $\mathcal{G}_{\text{LedgerLocks}}$: Parse $\alpha$ as (AccountId, $P, P'$) and $(P', \mathcal{P}_{\mathcal{P}})$, set $s'' := (s, P, P')$, and forward (sign, $s''$, $tx, \mathcal{K}_{\mathcal{P}}$, $\mathcal{Q}_{\mathcal{P}}$) to the simulated adaptor signature functionality $\mathcal{F}_{\text{AdaptSig}}$ in the name of $P$. Upon receiving (signature, $sid''$, $\sigma$) from $\mathcal{F}_{\text{AdaptSig}}$, output this answer to $\mathcal{A}$ and store $(tx, \mathcal{K}_{\mathcal{P}})$ in list $\mathcal{Q}_{\mathcal{P}}$.
- Upon receiving (lock-req, sid, $tx, \alpha, Y$) from $\mathcal{G}_{\text{LedgerLocks}}$: Parse $\alpha$ as (AccountId, $P, P'$) and $(P', \mathcal{P}_{\mathcal{P}})$, set $s'' := (s, P, P')$, forward (sign, $s''$, $tx, \mathcal{K}_{\mathcal{P}}$, $\mathcal{Q}_{\mathcal{P}}$) to the simulated adaptor signature functionality $\mathcal{F}_{\text{AdaptSig}}$ in the name of $P$. Upon receiving (signature, $sid''$, $\sigma$) from $\mathcal{F}_{\text{AdaptSig}}$, output this answer to $\mathcal{A}$ and store $(tx, \mathcal{K}_{\mathcal{P}})$ in list $\mathcal{Q}_{\mathcal{P}}$.
- Upon receiving (release-tx, sid, $y$) from $\mathcal{G}_{\text{LedgerLocks}}$, store $y$. □
Figure 21: Setup protocol of atomic swap in \((G_{\text{Cond}}^{R_{\text{merge}}}, G_{\text{LedgerLocks}} := (G_{\text{LedgerLocks}}^{A}, G_{\text{LedgerLocks}}^{B}))\)-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.

```
getOutput((id_{out}, j, rtl), state) :=
{(id_{out}, j, allD, v, h) | \exists b \text{ state}_{pre} \text{ state}_{post} \text{ in out A. tl.}
\text{state} = \text{state}_{pre} || b || \text{state}_{post} \wedge h = |\text{state}_{pre}|
\wedge (A, (id_{out}, \bar{in}, \bar{out}), tl) \in b
\wedge \text{out}_{j} = (allD, v)}
```
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 \( rtx^i \) for the money locked on the channel with \( u_{i+1} \)), \( u_i \) wants to submit \( rtx^i \). Then \( u_i \) needs to consider that \( rtx^i \) can only be published once the channel with \( u_{i+1} \) has been closed, meaning that both the commit transaction \( tx^i_c \) and the split transaction \( tx^i_s \) of this channel must have been published. Since publishing \( tx^i_c \) takes up to \( \#_{safe} \) (from the perspective of \( u_i \)) and after that (due to its relative timelock) \( tx^i_c \) 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 \( tx^i_s \) will be published at \( t[i] \) so that \( u_i \) can submit \( rtx^i \).

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

If indeed \( ctx^i \) 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 \( ctx^{i-1} \) 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 \( ctx^{i-1} \) is guaranteed to be published, the timelock \( t[i-1] \) of \( rtx^{i-1} \) needs to prevent that \( u_{i-1} \) could publish \( rtx^{i-1} \) before (and in this way outrun \( ctx^{i-1} \)). 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 \( ctx^{i-1} \) at least \( 4 \cdot \#_{safe} \) before the timelock \( t[i-1] \). If that will be included before \( rtx^i \) 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.

<table>
<thead>
<tr>
<th>Alice ((aID_A, Y^*, y, y_0))</th>
<th>Bob ((aID_B, Y^*))</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y^* \leftarrow \text{merge}(\text{wit}, R, (y, y_0)) )</td>
<td>( G^A )</td>
</tr>
<tr>
<td>Invoke ( G^B_{\text{LedgerLocks}} ) with ((\text{release-tx, sid, ctx}<em>A, aID^A</em>{AB}, Y^<em>, y^</em>))</td>
<td>( G^B_{\text{LedgerLocks}} )</td>
</tr>
<tr>
<td>Receive ((\text{release-tx, sid, aID}^B_{AB}) ) from ( G^A_{\text{LedgerLocks}} )</td>
<td>While ( \neg \text{inState}(\text{ctx}_A, \text{state}^B) \land</td>
</tr>
<tr>
<td>If ( b \neq 1 ) then abort</td>
<td>Invoke ( G^B_{\text{LedgerLocks}} ) with ((\text{read, sid}))</td>
</tr>
<tr>
<td>( G^B_{\text{LedgerLocks}} ) with ((\text{read, sid, state})) from ( G^B_{\text{LedgerLocks}} )</td>
<td>Receive ((\text{read, sid, state})) from ( G^B_{\text{LedgerLocks}} )</td>
</tr>
<tr>
<td>Invoke ( G^B_{\text{LedgerLocks}} ) with ((\text{signal-tx, sid, aID}^A_{AB}, \text{ctx}_A, Y^*))</td>
<td>Receive ((\text{signal-tx, sid, y}^*)) from ( G^B_{\text{LedgerLocks}} )</td>
</tr>
<tr>
<td>Receive ((\text{release-tx, sid, ctx}<em>B, aID^B</em>{AB}, Y^<em>, y^</em>)) from ( G^A_{\text{LedgerLocks}} )</td>
<td>Receive ((\text{release, sid, aID}^A_{AB}) ) from ( G^A_{\text{LedgerLocks}} )</td>
</tr>
</tbody>
</table>

Figure 22: Atomic swap in \((G^R_{\text{Conditional}}, G^A_{\text{LedgerLocks}} = (G^A_{\text{LedgerLocks}}, G^B_{\text{LedgerLocks}}))\)-hybrid world.
LedgerLocks: A Security Framework for Blockchain Protocols Based on Adaptor Signatures

CheckBase\(^C\)\((\mathcal{A}, (id, \vec{i}, out)), state) :=
\[
\begin{align*}
&\neg\exists \mathcal{A}' \text{ id}' \in \vec{i}' \text{ out}' t.f'. \mathcal{A} = (\mathcal{A}', ((id', \vec{i}', \text{out}'), tf')) \land \text{inState} (tx, state) \land id = id' \\
&\land \left(\forall (id_{out}, j, rtl) (id_{out}', j', rtl') \in \vec{i}. (id_{out}' \neq (id_{out}, j'))\right) \\
&\land \left(\mathcal{A} = \bigcup_{\mathcal{A}} \{alD | (id, j, alD, v, h) \in \text{getOutput}(in, state)\}\right) \\
&\land \left(\sum_{(alD, o) \in out} \mathcal{A} = \bigcup_{(alD, o) \in out} \text{getOutput}(in, state)\right) \\
&\land \left(\forall (id_{out}, j, rtl) \in \vec{i}. \exists alD' \circ h'. \text{getOutput} ((id_{out}, j, rtl), state) = \{(id_{out}, j, alD', o', h')\}\right) \\
&\land [\text{state}] = h \geq rtl \\
&\land \left(\forall (id_{out}, j, rtl) \in \vec{i}. \exists tx \mathcal{A}' \text{ id}' \in \vec{i}' \text{ out}' t.f'. tx = (\mathcal{A}', ((id', \vec{i}', \text{out}'), tf')) \land \text{inState} (tx, state)\right) \\
&\land \exists (id_{out}', j', rtl') \in \vec{i}'. (id_{out}, j') = (id_{out}, j) 
\end{align*}
\]

Figure 23: Definition of the CheckBase\(^C\) predicate.

GenFund\((tx_A, alD_A, tx_B, alD_B, alD_{AB})\)

\[
\begin{align*}
\text{parse } tx_A & \text{ as } (\mathcal{A}_A, ((id, \vec{i}_A, [alD_A, v_A]), tA)) \\
\text{parse } tx_B & \text{ as } (\mathcal{A}_B, ((id, \vec{i}_B, [alD_B, v_B]), tB)) \\
id & \leftarrow H (id_A || id_B) \\
\text{return } ((alD_{AB}, ([id_A, 0, 0], [id_B, 0, 0]), ([alD_{AB}, v_A + v_B]), 0)), (v_A, v_B))
\end{align*}
\]

GenSplit\((tx_e, \vec{a})\)

\[
\begin{align*}
\text{parse } tx_e & \text{ as } (\mathcal{A}, ((id, \vec{i}, [alD_{AB}, v]), tf)) \\
id' & \leftarrow H (id) \\
\text{return } ((alD_{AB}, ([id', 0, \#id]), \vec{a}, 0))
\end{align*}
\]

GenPay\((tx_e, alD_{AB}, alD, tl)\)

\[
\begin{align*}
\text{parse } tx_e & \text{ as } (\mathcal{A}, ((id, \vec{i}, [alD_{AB}, v], out_A, out_B], tf)) \\
id' & \leftarrow H (id) \\
\text{return } ((alD_{AB}, ([id', 0, 0], [alD, v]), tl))
\end{align*}
\]

GenPunish\((tx_e, alD_A)\)

\[
\begin{align*}
\text{parse } tx_e & \text{ as } (\mathcal{A}, ((id, \vec{i}, [alD_{AB}, v]), tf)) \\
id' & \leftarrow H (id) \\
\text{return } ((alD_{AB}, ([id', 0, 0], [alD_A, v]), 0))
\end{align*}
\]

ComputeBalance\((tx_f, tx_s)\)

\[
\begin{align*}
\text{parse } tx_f & \text{ as } (\mathcal{A}_f, ((id_f, \vec{i}_f, out_{in}, out_{tf}), tf_f)) \\
\text{parse } tx_s & \text{ as } (\mathcal{A}_s, ((id_s, \vec{i}_s, out_{in}, out_{tf}), tI_s)) \\
id' & \leftarrow H (id_f) \\
\text{return } ((alD_{AB}, ([id_f, 0, 0], \vec{i}_s, 0)), 0)
\end{align*}
\]

Figure 24: Definition of transaction constructors used in the channel and multi-hop payment protocol.
Figure 25: Create channel protocol in \((G^R_{\text{Con}} \cdot \mathcal{G}_{\text{LedgerLocks}})\)-hybrid world. Here, \textit{GenFund}, \textit{GenCommit}, \textit{GenSplit}, and \textit{GenPunish} denote the constructors for \(tx_f\), \(tx_c\), \(tx_s\), and \(tx_x\) respectively as described in Figure 24.
LedgerLocks: A Security Framework for Blockchain Protocols Based on Adaptor Signatures

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Update Channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice $(tx_c, alD_A, split-info)$ from LedgerLocks</td>
<td>Bob $(tx_f, alD_B, split-info)$ from LedgerLocks</td>
</tr>
<tr>
<td>Sample $(Y_A^1, Y_B^1)$ ← GenR$(1^1)$</td>
<td>Sample $(Y_A^2, Y_B^2)$ ← GenR$(1^2)$</td>
</tr>
<tr>
<td>Sample $(Y_A^3, Y_B^3)$ ← GenR$(1^3)$</td>
<td>Sample $(Y_A^4, Y_B^4)$ ← GenR$(1^4)$</td>
</tr>
<tr>
<td>$(Y_A^1, Y_B^1)$</td>
<td>$(Y_A^2, Y_B^2)$</td>
</tr>
<tr>
<td>$(Y_A^4, Y_B^4)$</td>
<td></td>
</tr>
</tbody>
</table>

Invoke $G_{\text{LedgerLocks}}$ on input (create-account, sid, B)

Receive (create-account, sid, alD_A) from LedgerLocks

Set $tx_c := \text{GenCommit}(tx_c, alD_A)$

Set $tx_s := \text{GenSplit}(tx_s, split-info)$

Set $tx_p := \text{GenPunish}(tx_p, alD_A)$

Invoke $G_{\text{Res}}$ on input (create-ind-cond, sid, $(Y_p^1, Y_p^2)$)

Receive (create-ind-cond, sid, $Y_p^1$) from $G_{\text{Res}}$

Invoke $G_{\text{Res}}$ on input (create-ind-cond, sid, $(Y_p^3, Y_p^4)$)

Receive (create-ind-cond, sid, $Y_p^3$) from $G_{\text{Res}}$

Invoke $G_{\text{Res}}$ on input (create-merged-cond, sid, $(Y_p^1, Y_p^2)$)

Receive (create-merged-cond, sid, $Y_p^1$) from $G_{\text{Res}}$

Invoke $G_{\text{Res}}$ with (lock-tx, sid, $tx_c, alD_A, Y_p^1$)

Receive (lock-req, sid, $tx_c, alD_A, (A, B)$) from LedgerLocks

Send (lock-req, sid, $b_p^1 = 1$) to LedgerLocks

Receive (lock-tx, sid, $b_p^{AB}$) from LedgerLocks

Invoke $G_{\text{Res}}$ with (auth-tx, sid, $tx_c, alD_A$)

Receive (auth-tx, sid, $b_p^A$) from LedgerLocks

Invoke $G_{\text{Res}}$ with (lock-tx, sid, $tx_p, alD_A, Y_p^1$)

Receive (lock-req, sid, $tx_p, (alD_B, (A, B))$) from LedgerLocks

Send (lock-req, sid, $b_p^B = 1$) to LedgerLocks

Receive (lock-tx, sid, $b_p^{AB}$) from LedgerLocks

Invoke $G_{\text{Res}}$ with (auth-tx, sid, $tx_p, (alD_B, (A, B))$)

Send (auth-rep, sid, $b_p^B = 1$) to LedgerLocks

Receive $G_{\text{Cond}}$ with (lock-tx, sid, $tx_p, alD_B, Y_p^4$)

Receive (lock-req, sid, $tx_p, (alD_B, (A, B))$) from LedgerLocks

Send (lock-req, sid, $b_p^B = 1$) to LedgerLocks

Receive (lock-tx, sid, $b_p^{AB}$) to LedgerLocks

If $b_p^{AB} \neq 1$ then abort

Read $(Y_p^1, Y_p^2)$ from chState[$\text{chState}$] = $|1|

Invoke $G_{\text{Res}}$ on input (open-cond, sid, $(Y_p^1, Y_p^2)$)

Receive (open-cond, sid, $b_p^1$) from $G_{\text{Res}}$

$b_p^1 = Y_p^1 \in \text{chState}[\text{chState}] = |1|

If $b_p^1 \neq 1$

Go to ForceClose(\text{chState}) = |2|

else

\text{chState}[\text{chState}] = (tx_c, tx_s, tx_p, Y_p^1, Y_p^2, Y_p^3, Y_p^4, Y_p^5, Y_p^6, split-info)

Figure 26: Update channel protocol in $(G_{\text{Res}} \circ G_{\text{Cond}} \circ G_{\text{LedgerLocks}})$-hybrid world. Here, $\text{GenCommit}$, $\text{GenSplit}$ and $\text{GenPunish}$ denote the constructors for $tx_c$, $tx_s$ and $tx_p$, respectively as described in Figure 24.
Figure 27: Close channel protocol in \((G_{\text{Cond}}^R, G_{\text{LedgerLocks}})\)-hybrid world. Here, \text{ComputeBalance} denotes the constructor for \(tx_t\) as described in Figure 24.

Figure 28: ForceClose and Punish algorithms in \((G_{\text{Cond}}^R, G_{\text{LedgerLocks}})\)-hybrid world.
LedgerLocks: A Security Framework for Blockchain Protocols Based on Adaptor Signatures

**Setup**

\[ return \] 

\[ set \] 

\[ if \] 

\[ while \] 

\[ for \] 

\[ sample \] 

\[ receive \] 

\[ for \] 

\[ receive \] 

\[ return \] 

\[ receive \] 

\[ receive \] 

\[ if \] 

\[ else \] ok

**Protocol** \( \Pi_{\text{Channel}} \)

**Monitor Channel**

\[ Alice(aID_{AB}, \text{chState}) \]

**Setup**

\[ P_0(t^*, n) \]

\[ for i \in \{n, 1\} \]

\[ set t_n := t^* \]

\[ for i \in \{t_n, n-1\} \]

\[ sample (y^0, y^1) \]

\[ receive \]

\[ return \]

\[ setup \]

**Protocol** \( \Pi_{\text{MultiHop}} \)

**Setup**

\[ P_1(Y_{aux}, y_{aux}, Y_{aux}^{-1}, Y^i, t_i, t_{i+1}) \]

\[ receive \]

**Figure 29: Monitor channel algorithm in \((\mathcal{G}^R_{\text{Cond}}, \mathcal{G}_{\text{LedgerLocks}})\)-hybrid world.**

**Figure 30: Setup algorithms in the \((\mathcal{G}^R_{\text{Cond}}, \mathcal{G}_{\text{LedgerLocks}})\)-hybrid world.**
Figure 31: Lock protocol in the \((G_{\text{Cond}}, G_{\text{LedgerLocks}})\)-hybrid world. Here, \textit{GenPay} denote the constructors for ctx and rtx, respectively as described in Figure 24.

Figure 32: On-chain payment, off-chain payment, force payment and refund algorithms in the \((G_{\text{Cond}}, G_{\text{LedgerLocks}})\)-hybrid world.
Figure 33: Multi-hop payment protocol in the $(\mathcal{G}_{\text{Cond}}, \mathcal{G}_{\text{LedgerLocks}})$-hybrid world.
The functionality is parameterized by four algorithms Validate, ExtendPolicy, Blockify, and predict-time, along with two parameters \( \text{WindowSize}, \text{Delay} \in \mathbb{N} \). The functionality manages variables \( \text{state}, \text{NxtBC}, \text{buffer}, \tau_L, \) and \( \tau_{\text{state}} \). Initially, \( \text{state} = \tau_{\text{state}}; \text{NxtBC} := \tau; \text{buffer} := \emptyset; \tau_L = 0 \).

**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 \( \mathcal{P}_{\text{DS}} \subseteq \mathcal{H} \). The sets \( \mathcal{P}, \mathcal{H}, \mathcal{P}_{\text{DS}} \) 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 \( \mathcal{H} \), \( \mathcal{P} \), and the current time of registration is also recorded; if the current time is \( \tau_L > 0 \), it is also added to \( \mathcal{P}_{\text{DS}} \). Similarly, when a party is deregistered, it is removed from both \( \mathcal{P} \) (and therefore also from \( \mathcal{P}_{\text{DS}} \) 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\(_C\), \( \tau \)) set \( \tau_L = \tau \) and do the following:

1. Let \( \hat{\mathcal{P}} \subseteq \mathcal{P}_{\text{DS}} \) denote the set of de-synchronized honest parties that have been registered (continuously) since time \( \tau' < \tau_L \) (to both ledger and clock). Set \( \mathcal{P}_{\text{DS}} := \mathcal{P}_{\text{DS}} \setminus \hat{\mathcal{P}} \). On the other hand, for any synchronize party \( P \in \mathcal{H} \setminus \mathcal{P}_{\text{DS}} \), if \( P \) is not registered to the clock, then \( \mathcal{P}_{\text{DS}} \cup \{ P \} \).

2. If \( I \) was received from an honest party \( P_i \in \mathcal{P} \):
   - Set \( \hat{I}_H^t = \hat{I}_H^t \parallel (I, P, \tau_L) \).
   - Compute \( \hat{N} = (\hat{N}_1, \ldots, \hat{N}_\ell) := \text{ExtendPolicy}(\hat{I}_H^t, \text{state}, \text{NxtBC}, \text{buffer}, \tau_{\text{state}}) \) and if \( \hat{N} \neq \emptyset \), set \( \text{state} := \text{state} || \text{Blockify}(\hat{N}_1) \parallel \cdots || \text{Blockify}(\hat{N}_\ell) \) and \( \tau_{\text{state}} := \tau_{\text{state}} || \tau_{\text{state},1}^t = \tau_L \parallel || \tau_L \).
   - For each \( \text{BTX} \in \text{buffer} \), if \( \text{Validate} \text{(BTX, state, buffer)} = 1 \), then delete \( \text{BTX} \) from buffer. Also, reset \( \text{NxtBC} := \emptyset \).
   - If there exists \( P_j \in \mathcal{H} \setminus \mathcal{P}_{\text{DS}} \) such that \( |\text{state}| - pt_j > \text{WindowSize} \) or \( pt_j < |\text{state}| \), then \( pt_k := |\text{state}| \) for all \( P_k \in \mathcal{H} \setminus \mathcal{P}_{\text{DS}} \).

3. Depending on the input \( I \) and the ID of the sender, execute the respective code:
   - **Submitting a transaction:**
     - \( I = \text{(submit, sid, tx)} \) and is received from a party \( P_i \in \mathcal{P} \) or from \( S \) (on behalf of a corrupted party \( P_i \)), do the following:
       - Choose a unique transaction ID \( \text{txid} \) and set \( \text{BTX} := (\text{tx}, \text{txid}, \tau_L, P_i) \).
       - If \( \text{Validate} \text{(BTX, state, buffer)} = 1 \), then buffer := buffer \cup \{ \text{BTX} \}.
       - Send (submit, BTX) to \( \mathcal{S} \).
   - **Reading the state:**
     - \( I = \text{(read, sid)} \) is received from a fully registered party \( P_i \), then set \( \tau_{\text{state},i} := \text{state} || \text{min}_{P_i,|\text{state}|} \text{pt}_i \) and return (read, sid, \( \tau_{\text{state},i} \)) to the requester.
   - If requestor is \( \mathcal{S} \), then send (state, buffer, \( \hat{I}_H^t \)) 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 \( \hat{I}_H^t \) as above) predict-time (\( \hat{I}_H^t = \hat{t} > \tau_L \)), then send (clock-update, sid\(_C\)) to \( \mathcal{G}_{\text{Clock}} \). Else, send \( I \) to \( \mathcal{S} \).
   - **The adversary proposing the next block:**
     - \( I = \text{(next-block, hFlag, (txid\(_1\), \ldots, txid\(_\ell\))) \text{ is sent from the adversary, update NxtBC as follows:} \}
       - Set \( \text{listOfTxid} := \emptyset \).
       - For \( i = 1, \ldots, \ell \) do: if there exists \( \text{BTX} := (x, \text{txid}, \text{minerID}, \tau_L, P_i) \in \text{buffer} \) with ID \( \text{txid} = \text{txid}_i \), then set \( \text{listOfTxid} := \text{listOfTxid} \cup |\text{txid}| \).
       - Finally, set \( \text{NxtBC} := \text{NxtBC} || \text{hFlag, listOfTxid} \) and output (next-block, ok) to \( \mathcal{S} \).
   - **The adversary setting state-slackliness:**
     - If \( I = \text{(set-slack, (P\(_1, p\(_1\)), \ldots, (P\(_i, p\(_i\)), \ldots)) \text{ with } (P\(_1, \ldots, P\(_\ell\)) \subseteq \mathcal{H} \setminus \mathcal{P}_{\text{DS}} \text{ is received from the adversary } S, \text{do the following:} \}
       - If for all \( j \in [\ell] \): \( |\text{state}| - p\(_{\text{state}}\) \leq \text{WindowSize} \) and \( p\(_{\text{state}}\) = p\(_{\text{state}}\) for every \( j \in [\ell] \) and return (set-slack, ok) to \( S \).
       - Otherwise, set \( p\(_{\text{state}}\) := |\text{state}| \) for all \( j \in [\ell] \).
   - **The adversary setting the state for de-synchronized parties:**
     - If \( I = \text{(desync-state, (P\(_1, \text{state}^1\)), \ldots, (P\(_\ell, \text{state}^\ell\)) \text{ with } (P\(_1, \ldots, P\(_\ell\)) \subseteq \mathcal{P}_{\text{DS}} \text{ from the adversary } S, \text{set } \text{state}^i = \text{state}^i \text{ for each } j \in [\ell] \text{ and return (desync-state, ok) to } S. \}

**Figure 34:** Ideal functionality \( \mathcal{G}_{\text{ledger}} [8] \).