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

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

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

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

ABSTRACT

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

To help this, we present LedgerLocks, a framework for the secure design of AS-based blockchain applications in the presence of a realistic blockchain. LedgerLocks defines the concept of AS-locked transactions, transactions whose publication is bound to the knowledge of a cryptographic secret. We argue that AS-locked transactions are the common building block of AS-based blockchain protocols and we define $G_{\text{LedgerLocks}}$, 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.

CCS CONCEPTS

- Security and privacy → Cryptography.

KEYWORDS

Adaptor Signatures, Universal Composability, Security

ACM Reference Format:

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 [31] 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, 16], 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, 10, 17, 19, 22–25, 31], 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, 19, 24, 28, 31].

However, the subtleties of the ledger model have a significant influence on the blockchain protocol security and neglecting these aspects can easily result in undetected security issues as we will
Figure 1: Overview of the LedgerLocks framework. Cryptographic protocols for creating AS-locked transactions in the state of the art (in green) are modeled by an ideal functionality $G_{\text{LedgerLocks}}$.  

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 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 [16], 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 blockchain 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
Asian 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 its 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 $dx_A$ and $dx_B$). 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 $rx_A$ and $rx_B$ 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 $rx_A$). 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 $rx_A$ or $rx_B$ respectively. Alice’s refund transaction $rx_A$ is equipped with a timelock that ensures that it can only be published after time $t$. This restriction prevents Alice from simultaneously publishing $rx_A$ and $tx_B$ to retrieve the assets on both chains. Instead, if Alice has not claimed Bob’s assets (through $tx_A$) until time right before $t$, Bob can publish $rx_B$ 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 $rx_A$ will be included immediately after Bob sent it to the network. In practice, the ledger only guarantees, that $rx_A$ 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 $rx_A$, and, thus, prevent its inclusion. Specifically, a malicious Alice could send $tx_B$ right after observing $rx_A$ on the network. As a consequence, $tx_B$ could be included in the blockchain first, canceling $rx_A$.

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 $t-\Delta$, 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 $tx_A$, which is guaranteed to be included before $t$, the time starting from which Alice could publish $rx_A$.

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 $tx_B$) before Alice could refund her assets using $rx_A$ at time $t$.

This issue can be mitigated when introducing an additional timelock for Bob’s refund transaction $rx_B$ 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 [24]. Payment channel networks, like Bitcoin’s Lightning Network [26], 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

---

**Figure 2:** Transactions in an atomic swap protocol.

**Figure 3:** Atomic swap in a ledger with delayed inclusion (top). Attack in an atomic swap in a realistic ledger (bottom).
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 \( \text{ctx}_i \) on the channel funds that can be published once the channel is closed. The conditions \( c_i \) are set up such that if user \( u_{i+1} \) claims \( \text{ctx}_i \) this will reveal a secret \( x_i \) to \( u_i \) allowing them to satisfy \( c_{i-1} \) and claim \( \text{ctx}_{i-1} \) in turn. Once all conditional payments are set up, \( S \) initiates the payment by revealing a secret \( s_R \) to \( R \) that allows them to open \( c_{n-1} \). Next, the payment gets propagated through the payment path: Collaborative users \( u_i \) and \( u_{i+1} \) can update their payment channels offchain after revealing the secret that would enable \( \text{ctx}_i \). If \( u_i \) is not collaborating, \( u_{i+1} \) can close the channel and use \( \text{ctx}_i \) to enforce the conditional payment based on the last channel state.

To ensure that a malicious sender cannot indefinitely lock the funds of intermediaries on the path, the users, in addition to \( \text{ctx}_i \) prepare a refund transaction \( \text{ctx}_i \) 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 \( \text{ctx}_i \). However, even after successful channel closure while waiting for \( \text{ctx}_i \) to be included in the blockchain, \( u_{i+1} \) may still decide to publish \( \text{ctx}_i \) instead. In this case \( u_i \) needs to observe \( \text{ctx}_i \) on the blockchain, learn \( x_i \) and use it to continue the payment (either offchain or onchain). For this, it needs to be ensured that \( \text{ctx}_{i-1} \) is still valid at this point and so that \( \text{ctx}_{i-1} \) has not been published yet.

This illustrates how the protocol design is closely intertwined with the precise guarantees that the underlying ledger provides: (i) The protocol transactions need to have timelocks that respect the ledger inclusion times. In particular, timelock \( t_i \) of \( \text{ctx}_i \) needs to be adjusted such that \( t_i < t_{i-1} + 2\Delta + \Delta_{\text{close}} \) (for \( t_{i-1} \) being the timelock of \( \text{ctx}_{i-1} \) and \( \Delta_{\text{close}} \) being the channel closing time) to ensure that \( u_i \) after closing their outgoing channel and publishing \( \text{ctx}_i \) at timelock \( t_i \), when learning (latest) at \( t_i + \Delta \) whether \( \text{ctx}_{i-1} \) or \( \text{ctx}_{i-1} \) got included in the blockchain there is still enough time to close their incoming channel and publish \( \text{ctx}_{i-2} \) on the blockchain (which may take up to \( \Delta_{\text{close}} + \Delta \)). (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 \( \text{ctx}_i \) 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 \( s_R \) only after \( t_R + \Delta_{\text{close}} \), \( R \) 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 \( \Delta_{\text{close}} \)) and then publish \( \text{ctx}_n \) at time \( t < t_R \). At \( t_R \), however, a malicious \( u_{n-1} \) already submitted \( \text{ctx}_n \) to outrun \( \text{ctx}_n \) resulting in \( u_{n-1} \) being refunded while still learning \( s_R \). With the knowledge of \( s_R \), \( u_{n-1} \) completes the payment and receives the funds meant for \( R \). Similar to the atomic swap example, such an attack could not be detected in the presence of a ledger model with instant transaction inclusion or without modeling that the attacker may learn transaction details (such as \( x_i \)) before the transaction's inclusion in the blockchain.

Ledger 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 [20] who propose a formal security analysis of Bitcoin’s Lightning Network in the presence of the ledger model \( G_{\text{ledger}} \) from [8]. The authors showcase that for precisely specifying the Lightning Network protocol, it is inevitable to rely on the exact timing guarantees obtained from the ledger in [8]. Further, they prove that the simplified models that are used in the security analysis of [12–15, 31] 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 unrealistic) ledgers \( L_{\text{inav}} \) with immediate inclusion guarantees [12–15, 31]; or (ii) ledgers \( L_{\Delta} \) that let the attacker delay the inclusion of a transaction up to delay \( \Delta \) [1, 2, 4, 5, 19, 27, 28, 30, 32]. Even in protocols from the second category, the exact way that the attacker can exercise their power to delay transactions stays vague. E.g., in [1], it is stated that upon a message being posted by the user, the ledger should "wait until round \( \tau_1 \leq \tau_0 + \Delta \) (the exact value of \( \tau_1 \) 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 \( \tau \). 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.

### 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 [11]. 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 II is proven secure w.r.t. an ideal functionality \( \mathcal{F} \), the security of protocols using II as a subroutine can be analyzed assuming \( \mathcal{F} \) as subroutine instead.

The LedgerLocks framework provides ideal functionalities to characterize the security of cryptographic conditions (\( \mathcal{G}_{\text{Cond}} \)), adaptor signatures (\( \mathcal{G}_{\text{AdaptSig}} \)), 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 II_{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 \((\mathbb{Z}/q\mathbb{Z}, g, q)\) (with generator \( g \) and order \( q \)), where given a group element (the statement) \( Y = g^y \), it is hard to compute the exponent \( y \) (the witness).

At first sight, it may seem counter-intuitive to define conditions as a standalone ideal functionality. The reason for doing so is that the prerequisites for a party to craft a witness related to a statement often emerge from a cryptographic protocol for the condition creation. As an example, consider the following scenario. Alice plays a simple guessing game with Bob. If Bob can guess a number between 1 and 10 then he gets a prize from Alice. To implement this based on DLOG, Alice prepares ten secret witnesses \((y_i^{\text{mask}})_{i\in[1,10]}\) and sends the corresponding statements \(y_i^{\text{mask}} = (g^y)_{i\in[1,10]}\) to Bob. Bob himself prepares one witness \(y_i^{\text{win}}\) and ten witnesses \((y_i^{\text{blank}})_{i\in[1,10]}\).

For the guess \( j \), Bob prepares \( Y_i^{\text{guess}} = (Y_i^{\text{guess}})_{i\in[1,10]}\) such that for \( i \neq j \), \( Y_i^{\text{guess}} = (Y_i^{\text{mask}})_{i\neq j} \) and \( Y_j^{\text{guess}} = y_i^{\text{win}} \). At this point, Bob knows the witness for \( Y_j^{\text{guess}} \), while he cannot know the witness for any statement \( Y_i^{\text{guess}} \) 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 \( Y_i^{\text{guess}} \) 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 out of the ten provided conditions. If Alice and Bob chose the same number \( (j = m) \), Bob can complete the payment, otherwise, the money stays with Alice.

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

Table 1: Overview of features of ledger models used for the analysis of blockchain protocols. \( X^* \) denotes that the corresponding ledger feature is underspecified, \( X^† \) indicates that the realizability of \( L_\Delta \) is unknown.
In §6, we define \( w.r.t. \) to \( R \). This means that for signature adaption \( F \) characterized in terms of game-based security definitions, modular reasoning with respect to conditions, relation \( R \) provided. Similar to \( G \), determining whether the correct witness to open a condition was obtained through adaptation with \( y \) if, for any given message \( m \), they can provide a valid pre-signature \( \hat{\sigma} \) 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. [16] 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 [16]) 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 \( F_{\text{AdaptSig}} \).

**Lock-enabling ledger.** In §7, we define \( G_{\text{Ledger}} \) as an ideal functionality for a distributed ledger with AS-locked transactions. In the design of \( G_{\text{Ledger}} \), we follow the technique in [8]: The authors in [8] provide \( G_{\text{Ledger}} \) as an ideal functionality modeling the subtilies 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;

---

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

---

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 \( P = \text{Bob} \) and \( U = \{ \text{Alice} \} \).

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

Within our LedgerLocks framework, we treat \( G_{\text{Cond}} \) as a global subroutine [6]. In the UC framework, a global subroutine \( G \) is an ideal functionality that can be securely used by some protocol \( \Pi \), even if \( \Pi \) relies on another functionality \( F \), which makes use of \( G \) as a subroutine. One can think of \( G \) as constituting some safely shared state between \( \Pi \) and \( F \). An example for this usage of \( G_{\text{Cond}} \) is shown in Figure 6: Here, the protocol \( \Pi_{\text{AtomicSwap}} \) uses the functionalities \( G_{\text{Cond}} \) (for creating conditions) and \( G_{\text{Ledger}} \) again uses \( G_{\text{Cond}} \) as a subroutine (for checking conditions). Having \( G_{\text{Cond}} \) 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 \( F_{\text{AdaptSig}} \) as an ideal functionality for adaptor signatures. Following the two-party adaptor signatures scheme defined by Erwig et al. [16], 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 \( \hat{\sigma} \) 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
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.

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 can complete the authorization by providing an adequate witness. If such an AS-locked transaction is published on the ledger, honest account owners learn the corresponding witness, while a malicious owner learns the witness already upon transaction submission.

We build \( G_{\text{LedgerLocks}} \) 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; and 3) instantiating the valid predicate to check for correct transaction authorization. The operation for releasing a 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 \( G_{\text{LedgerLocks}} \) 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 the full version [29]. These examples also 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 discuss privacy in §6 and give (game-based) definitions for adaptor signature privacy and a resulting privacy notion achieved by lock-enabling ledgers in the full version [29].

### 4 PRELIMINARIES

We review adaptor signatures and defer their full definition and other basic cryptographic primitives to the full version [29]. 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 \( L_R := \{ Y \mid \exists y \text{ s.t. } (Y, y) \in R \} \). We say that \( R \) is a hard relation if:

1. There exists a PPT sampling algorithm \( \text{GenR}(\lambda) \) that on input the security parameter \( \lambda \) outputs a statement/witness pair \((Y, y) \in R \) 2. The relation is poly-time decidable. 3. For all PPT adversaries \( A \) there exists a negligible function \( \negl(\lambda) \), such that:

\[
\Pr \left[ (Y, y^*) \in R \mid (Y, y) \leftarrow \text{GenR}(\lambda), \ y^* \leftarrow A(Y) \right] \leq \negl(\lambda),
\]

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

**Definition 2** (Two-Party Adaptor Signature Scheme with Aggregatable Public Keys [16]). 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 \( \Sigma_2 = (\text{Setup}, \text{Gen}, \Pi_{\Sigma}, \text{Agg}, \text{Vf}) \). It is run between parties \( P_0, P_1 \) and consists of a tuple \( \Xi = (\Pi_{\text{PreSig}}, \text{Adapt}, \text{PreVf}, \text{Ext}) \) of efficient protocols and algorithms defined as follows:

- \( \Pi_{\text{PreSig}}(sk_i, \text{apk}_i, \text{pk}_i, m, Y) \): 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 \( \text{pk}_0, \text{pk}_1 \) and statement \( Y \in L_R \), outputs a pre-signature \( \delta \).
- \( \text{PreVf}(\text{apk}_m, m, \delta) \): is a DPT algorithm with input an aggregated public key \( \text{apk} \), a message \( m \in \{0, 1\}^* \), a statement \( Y \in L_R \) and a pre-signature \( \delta \), outputs bit \( b \).
- \( \text{Adapt}(\text{apk}, \delta, y) \): is a DPT algorithm with input an aggregated public key \( \text{apk} \), pre-signature \( \delta \) and witness \( y \), outputs a signature \( \sigma \).
- \( \text{Ext}(\text{apk}, \sigma, \delta, Y) \): is a DPT algorithm with input an aggregated public key \( \text{apk} \), a signature \( \sigma \), pre-signature \( \delta \) and statement \( Y \in L_R \), outputs a witness \( y \) s.t. \((Y, y) \in R \), or \( \bot \).

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.
The functionality interacts with an adversary $S$ and set of parties $\mathcal{P} = \{P_1, \ldots, P_k\}$. 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: $(Y, y_1) \in R \land (Y, y_2) \in R \implies (f_{\text{merge}}(\text{stmt}, R, (Y, y_1)), f_{\text{merge}}(\text{wit}, R, (Y, y_2))) \in R$.

**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-ind-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 $y^* = f_{\text{merge}}(\text{wit}, R, (\mathcal{L}[Y_1], \mathcal{L}[Y_2]))$, set $\mathcal{L}[Y^*] = y^*$ 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 $b = (\mathcal{L}[Y^*] \neq y^*)$ and send (opened-cond, sid, $b$) to $P^*$ and $S$.

**Definition 3** (Secure Two-Party Adaptor Signature Scheme). A two-party adaptor signature scheme with aggregateable public keys $\mathcal{G}^{R_{\text{merge}}}_{\text{Cond}}$ is secure if it is 2-eUF-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 [11]. 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 $\mathcal{E}$ interacting with protocol $\pi$ and an adversary $A$, on input the security parameter $1^k$ and auxiliary input $z$, is denoted as $\text{EXEC}_{\pi, A, \mathcal{E}}(1^k, z)$. Let $\phi_{\mathcal{F}}$ be the ideal protocol for an ideal functionality $\mathcal{F}$, i.e., $\phi_{\mathcal{F}}$ is a trivial protocol in which the parties simply forward their inputs to the ideal functionality $\mathcal{F}$. The output of an environment $\mathcal{E}$ interacting with protocol $\phi_{\mathcal{F}}$ and an adversary $S$ (also called the simulator), on input the security parameter $1^k$ and auxiliary input $z$, is denoted as $\text{EXEC}_{\phi_{\mathcal{F}}, \mathcal{S}, \mathcal{E}}(1^k, z)$.

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 $\phi_{\mathcal{F}}$.

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

$$\text{EXEC}_{\pi, A, \mathcal{E}}(1^k, z) \approx_{\text{c}} \text{EXEC}_{\phi_{\mathcal{F}}, S, \mathcal{E}}(1^k, z),$$

(whence $\approx_{\text{c}}$ denotes computational indistinguishability).

### 5 Global Conditions

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

**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 $f_{\text{merge}}$. $\mathcal{G}_{\text{Cond}}$ 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 $\mathcal{L}$. The merged conditions interface models the creation of a condition as the composition of two other conditions, where the concrete composition operations are given by $f_{\text{merge}}$. This function is split into an operation $\oplus$ on witnesses and an operation $\cdot$ on statements, which need to satisfy that $(Y_1 \cdot Y_2, y_1 + y_2) \in R$ if both $(Y_1, y_1), (Y_2, y_2) \in R$. The open condition interface allows checking if a condition/opening pair is valid (i.e., is in $\mathcal{L}$).

**Global Conditions Protocol**. We describe the global conditions protocol $\Pi_{\text{Cond}}$ in Figure 8. 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
Upon receiving \((\{\text{keygen}, \text{sid}\})\) from \(P_0\) and \(P_1\), verify that \(\text{sid} = (P_0, P_1, \text{sid}')\) for some \text{sid}'. If not, ignore the request. Else, send \((\{\text{keygen}, \text{sid}\})\) to \(S\). Upon receiving \((\text{verification-key}, \text{sid}, v)\) from \(S\), add \(v\) into \(\mathcal{K}[\text{sid}]\) and send \((\text{verification-key}, \text{sid}, v)\) to \(P_0\) and \(P_1\).

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

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

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

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

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

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

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

Else, set \(f = \phi\), add \((m, \sigma, v, \phi)\) into \(\mathcal{Q}[\text{sid}]\) (if \(\text{type} = \text{signature}\)) or \((m, \hat{\sigma} = \sigma, \bot, v, Y, \phi)\) into \(\mathcal{P}[\text{sid}]\) (if \(\text{type} = \text{pre-signature}\)).

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

---

**Figure 9: Ideal functionality \(\mathcal{F}_{\text{AdaptSig}}^{R, f_{\text{merge}}}\)**

Assume that the group \(\mathcal{G}\) is a DLOG-hard group here. The function \(f_{\text{merge}}\) defines the witness operation (\(+\)) 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 \(b\).

**Security.** The security of our construction is established with the following theorem. We provide a proof in the full version [29].

**Theorem 1.** Let \(\mathcal{G}\) be a DLOG-hard group, then the protocol \(\Pi_{\text{R}_{\text{DLOG}}}^{\text{Cond}}\) UC-realizes the ideal functionality \(\mathcal{G}_{\text{Cond}}^{R, f_{\text{merge}}}\), for \(R = \text{R}_{\text{DLOG}}\) 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 the full version [29] we show an extension to account for additional combinations of conditions, e.g., 1-out-of-n. Note that the existing interfaces of \(\mathcal{G}_{\text{Cond}}\) would stay unaffected by such extensions and, hence, proofs conducted with respect to the current version of \(\mathcal{G}_{\text{Cond}}\) 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 [16]) or the \(R_{\text{SIS}}\) relation (as used in Dilithium-based adaptor signature [18]).

**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 aggregateable public keys \(\Xi_{2}^{R}\) securely realizes our ideal functionality.

**Two-party adaptor signature functionality.** Our functionality \(\mathcal{F}_{\text{AdaptSig}}\) (cf. Figure 9) extends the digital signature functionality in [21] to adaptor signatures by considering a hard relation \(R\) and a deterministic adaptation function \(f_{\text{merge}}\). 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 \(f_{\text{merge}}\) and global conditions functionality \(\mathcal{G}_{\text{Cond}}\).
The parametrization with $f_{\text{adap}}$ 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 $\hat{\sigma} = f_{\text{adap}}(\sigma, y)$ are distinctly connected. For showing that a protocol $\Pi_{\text{AdapSig}}$ UC-realizes $F_{\text{AdapSig}}$ without assuming any knowledge about which party knows $y$, we need to ensure that signatures created via $F_{\text{AdapSig}}$’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 $\sigma$ 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 $F_{\text{AdapSig}}$ compute the correct signature based on $y$, to which end we must fix $f_{\text{adap}}$.

### 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) to a protocol $\Pi_{\text{AdapSig}}$ in the full version [29]. Note that the construction is parametric with respect to $G_{\text{Cond}}$ where $G_{\text{Cond}}$ needs to support the relation $R$ of $\Sigma_{2}$. The security is established with the following theorem, which we prove in the full version [29].

**Theorem 2.** Let $\Sigma_{2}$ 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 $\Pi_{\text{AdapSig}}$ UC-realizes the ideal functionality $F_{\text{AdapSig}}$.

### 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 the full version [29] 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 $F_{\text{AdapSig}}$ 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 $\sigma$ 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 $G_{\text{LedgerLocks}}$. Figure 10 shows how $G_{\text{LedgerLocks}}$ is constructed from the base ledger $G_{\text{Ledger}}$ defined in [8].

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

The transaction locking interface allows the parties owning an account to jointly create an authorization for a transaction, which is locked under a specified condition and can only be released using the opening information for this condition. This authorization is recorded in $L_{\text{TxLock}}$. 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 $L_{\text{TxLock}}$ to $L_{\text{Author}}$. The witness signaling interface allows the account parties to extract the condition witness from the published AS-locked transaction.

One subtlety in our model here is that witness signaling is only enabled when the previously released transaction is added to the ledger. Only then we can guarantee that any party (involved in its creation) would have seen both the AS-locked transaction and the released transaction. We model this by checking whether the
upon receiving

our predicate

and given in Figure 12. During account generation, parties obtain verification keys by making calls to the key generation interface of \( \mathcal{F}_{\text{AdaptSig}} \). An analogous construction makes transactions and locking of transactions happen with a call to the signing interface of \( \mathcal{F}_{\text{AdaptSig}} \), where in the latter case, only a pre-signature \( \hat{\sigma} \) that is conditioned on \( Y \) is computed, whereas in the former a full signature \( \sigma \) over the transaction is computed. Releasing of transactions happens by calling the adaptation interface of \( \mathcal{F}_{\text{AdaptSig}} \) with the witness \( Y \) used during the locking procedure. Lastly, the witness signaling calls the extraction interface of \( \mathcal{F}_{\text{AdaptSig}} \), which returns witness \( y \).

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

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

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

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

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

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

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

**Release TX:** Party $P$ upon receiving $(\text{release-tx}, \text{sid}, tx, Y, P, P', P, vk)$ from $E$:
- Party $P$: Compute $\text{sid'} := (\text{sid, P, P'})$, extract entry $(tx, vk, Y, \delta)$ from $\mathcal{P}$, invoke $F^{R,\text{adapt}}_{\text{AdaptSig}}$ on input $(\text{adapt, sid', \delta, \sigma, \tau, \tau, y})$, receive $(\text{adapted-signature, sid', \sigma})$, and store $(tx, vk, \sigma)$ in $\mathcal{Q}$.
- Party $P$: Invoke $G_{\text{Ledger}}$ on input $(\text{submit, (tx, \sigma)})$.

**Signal Witness:** Party $P$ upon receiving $(\text{signal-tx}, \text{sid}, tx, Y, vk)$ from $E$:
- Extract the entry $(tx, Y, \delta)$ from $\mathcal{P}$, otherwise abort.
- Party $P$: Receive $\sigma$ from $P_1$ and $P_{1-b}$, store $(tx, Y, \sigma)$ in $\mathcal{Q}$.

**Ledger Read:** Party $P$ upon receiving $(\text{read, sid})$ from $E$, do the following:
- Party $P$: Invoke $G_{\text{Ledger}}$ on input $(\text{read, sid})$, and receive the current state.
- Check if inState $((\text{vk}_1, \ldots, \text{vk}_n), \text{tx'})$, $(\text{\sigma}_1, \ldots, \text{\sigma}_n)$ state and $\text{vk}_i = \text{vk}$ for some $\text{vk}_i$, otherwise abort.
- $F^{R,\text{adapt}}_{\text{AdaptSig}}$ on input $(\text{extract, \sigma_i, \delta, \tau, \tau, y})$, receive $(\text{signature, \text{sid}', \sigma})$ and return $\text{ledger}$.

**Submit TX:** Party $P$ upon receiving $(\text{submit, \text{sid}, tx})$ from $E$:
- Farse $tx := ((\text{vk}_1, \ldots, \text{vk}_n), \text{tx'})$ and check that each $\text{vk}_i$ is in $K$. Otherwise, ignore the request.
- Read the state from $G_{\text{Ledger}}$ as above. For each $\text{vk}_i$, extract the entry $(tx, \text{vk}_i, \sigma_i)$ from $\mathcal{Q}$. If any of them is missing, then abort.
- $G_{\text{Ledger}}$ on input $(\text{submit, (tx, (\text{\sigma}_1, \ldots, \text{\sigma}_n))})$.

**Validate Predicate:** Our predicate $\text{CheckAdapt}(tx, \text{state})$ instantiates isValidate(tx, state) in [8] as follows:
- Farse $tx' := ((\text{vk}_1, \ldots, \text{vk}_n), \text{tx'})$, $(\text{\sigma}_1, \ldots, \text{\sigma}_n))$. For each pair $(\text{vk}_i, \sigma_i)$, invoke $F^{R,\text{adapt}}_{\text{AdaptSig}}$ on input $(\text{verify, sig, tx, \sigma_i, \text{vk}_i, \perp, signature})$, receive $(\text{signature, sid, tx, f_i, \tau})$, and set $b_{i,1} := f_i$.
- Set $b_2 := \text{CheckBase}((\text{vk}_1, \ldots, \text{vk}_n), \text{tx'})$, (state), and return $b_1 \land \ldots \land b_n \land b_2$.

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

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 the full version [29]. 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 (adapter) 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 block height. The timelock height defines the number of blocks in the blockchain and can be accessed through the variable state of \( G_{\text{Ledger}} \) (which is inherited from \( G_{\text{LedgerLocks}} \)). 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:

\[
\text{CheckBase}(\langle A, (tx'', tl) \rangle, \text{state}) = \begin{cases} \text{state} \geq tl & \land \text{CheckBase}''(\langle A, tx'' \rangle, \text{state}) \\ \end{cases}
\]

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

Figure 14 shows the setup phase of the atomic swap protocol. Both parties initially know the oracle condition \( Y_0 \), 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 \( Y_0 \) 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 \( \text{dtx} \) (which spends funds of the corresponding party to the joint account), (2) a claim transaction \( \text{ctx} \) (which transfers the deposit from the joint account to the other user), and (3) a refund transaction \( \text{rtx} \) (which transfers the deposit back to the original owner). Importantly, \( \text{ctx}_A \) and \( \text{rtx}_A \) (along with \( \text{ctx}_B \) and \( \text{rtx}_B \)) 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 \( t \). These timelocks will ensure that Alice will always have enough time to safely claim \( \text{ctx}_A \) and that Bob will always have enough time to claim \( \text{ctx}_B \) once \( \text{ctx}_A \) 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 \( \text{windowsize}^C \) (the maximum amount of blocks that an honest user can be lacking behind the current state of the blockchain), \( \text{minTime}^C \) (the minimal amount of time for including \( \text{windowsize}^C \) blocks) and \( \text{maxTime}^C \) (the maximal amount of time for including \( \text{windowsize}^C \) blocks). Based on these parameters, we can define \( \#_{\text{safe}}^C = 5 \cdot \text{windowsize}^C \) to be the maximal

\[
\begin{align*}
\text{safe} & (h_A, h_B, n_A, n_B) = h_B < h_A + \#_{\text{safe}}^C + 2 \cdot \#_{\text{min}}^C \\
& \land h_A < n_A + \#_{\text{safe}}^C + (h_B + \#_{\text{safe}}^C - n_B) + \#_{\text{safe}}^C
\end{align*}
\]

Figure 13: Notions for safe timelocks in atomic swaps.
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.\footnote{Up to $\text{windowsize}$ 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 $h + \text{windowsize} - 1$. 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.}

The function $u_{C_1 \to C_2}$ 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 blockheight $n_B$ on ledger $B$ there is still enough time to include transactions $dtx_A$ on $A$ (taking up to $u_{B \to A}(n_B)$ blocks) and to include transactions $dtx_B$ and $ctx_A$ on $B$ (taking up to $2 \cdot v_{\text{safe}}$ 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 $rtx_B$ that they submitted after $h_B$ has been included on $B$ (latest after $h_B + v_{\text{safe}}$ blocks), the blockheight on $A$ (computed by $n_A + u_{B \to A}(h_B + v_{\text{safe}} - n_B)$) is still at least $v_{\text{safe}}$ before $h_A$ so that $ctx_B$ can be safely included. After checking the timelocks, the users authorize their refund transactions and lock their claim transactions on the condition $Y'$. Once this is done, Alice submits $dtx_A$ to $A$, 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 if $\text{state} < |h_B - v_{\text{safe}}|$ so that there is still time for Alice to claim $ctx_A$ before Bob needs to initiate the refund.

Figure 15 shows the case when a malicious Alice does not submit $ctx_A$ in time. Here, Bob needs to submit $rtx_B$ at the earliest possible point (once $h_B$ is reached). Thanks to $B$’s transaction inclusion guarantees, Bob can enforce that by $h_B + v_{\text{safe}}$ either the refund was successful, or $ctx_A$ must have been published. In the latter case (thanks to the original validity check), there is still enough time ($v_{\text{safe}}$) 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 $rtx_A$ in before $ctx_B$. Similarly, in an attack as shown in §2, if $h_B$ is not set properly, Bob could outrun Alice’s $ctx_A$ transaction with $rtx_A$ 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.

Figure 15: Atomic Swap: Malicious $A$ claims late.

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

In cases where $\Pi^*$ involves several blockchains, the description of $\Pi^*$ uses multiple instances of $G_{\text{LedgerLocks}}$ (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 $G_{\text{Cond}}$’s condition creation interface can be extended to $G_{\text{Cond}}^*$ to support the new condition types. In this case, for some relation $R$ (known to realize $G_{\text{Cond}}$) one needs to give a protocol $\Pi_R$ and prove that it realizes the newly added interfaces in $G_{\text{Cond}}^*$. Alternatively, one can immediately give a protocol $\Pi_R$ for some new relation $R^*$ (that is known to be supported by an adapter signature scheme) and show it to realize all of $G_{\text{Cond}}^*$.

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

**Proving UC-realization.** Finally, one needs to prove \( \Pi^* \) to UC-realize \( F^* \). This proof should not involve cryptographic reductions but focus on how the interactions of \( \Pi^* \) with \( \mathcal{G}_{\text{LedgerLocks}} \) and \( \mathcal{G}^*_{\text{Cond}} \) are translated into interactions with \( F^* \). As \( F^* \) uses \( \mathcal{G}_{\text{LedgerLocks}} \) as a component, the proof’s essence should lie in showing that the way that transactions are created, locked and released within \( \Pi^* \) 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 Zeroscan [9]. Further, since \( \mathcal{G}_{\text{Ledger}} \) is currently only shown to be realized by the Bitcoin (PoW) backbone protocol [8] and the Ouroboros Genesis (PoS) protocol [7], LedgerLocks (relying on the security of \( \mathcal{G}_{\text{ledger}} \)) only provides full end-to-end guarantees for ledgers using one of these protocols.

### 9 CONCLUSION

We provide foundations for the security of adaptor signatures and 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.

**Acknowledgments.** 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 Christian Doppler Laboratory Blockchain Technologies for the Internet of Things (CDL-BOT); by grant IJC2020-043991-I, by PRODIGY Project (TED2021-132464-B-I00), ESPADA project, MCIN/AEI/10.13039/501100001033, and the European Union NextGenerationEU/PRTR. Further, this work has been supported by the Heinz Nixdorf Foundation through a Heinz Nixdorf Research Group (HN-RG) and funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany’s Excellence Strategy—EXC 2092 CASA—390781972.

**REFERENCES**


