LightSwap: An Atomic Swap Does Not Require Timeouts At Both Blockchains

ABSTRACT

Security and privacy issues with centralized exchange services have motivated the design of *atomic swap* protocols for decentralized trading across currencies. These protocols follow a standard blueprint similar to the 2-phase commit in databases: (i) both users first lock their coins under a certain (cryptographic) condition and a timeout; (ii-a) the coins are swapped if the condition is fulfilled; or (ii-b) coins are released after the timeout. The quest for these protocols is to minimize the requirements from the scripting language supported by the swapped coins, thereby supporting a larger range of cryptocurrencies. The recently proposed universal atomic swap protocol [IEEE S&P'22] demonstrates how to swap coins whose scripting language only supports the verification of a digital signature on a transaction. However, the timeout functionality is cryptographically simulated with verifiable timelock puzzles, a computationally expensive primitive that hinders its use in battery-constrained devices such as mobile phones. In this state of affairs, we question whether the 2-phase commit paradigm is necessary for atomic swaps in the first place. In other words, is it possible to design a secure atomic swap protocol where the timeout is not used by (at least one of the two) users?

In this work, we present LightSwap, the first secure atomic swap protocol that does not require the timeout functionality (not even in the form of a cryptographic puzzle) by one of the two users. LightSwap is thus better suited for scenarios where a user, running an instance of LightSwap on her mobile phone, wants to exchange coins with an online exchange service running an instance of LightSwap on a computer. We show how LightSwap can be used to swap Bitcoin and Monero, an interesting use case since Monero does not provide any scripting functionality support other than linkable ring signature verification.

KEYWORDS

Blockchain and Atomic swap and Bitcoin and Monero and Lightweight applications and Adaptor signatures.

1 INTRODUCTION

The functionality of atomic swaps [21] was introduced for trading assets between two parties such that each of them holds assets in a different blockchain. The concept of atomicity in such a setting is inspired by database systems where either a multi-step transaction gets committed or it is rolled back in its entirety. In the blockchain setting, it holds similar relevance guaranteeing that the swap either fully occurs or fails entirely [20, 48].

As an illustrative example, consider that a user *Alice* has asset α in blockchain $\mathcal{B}_A$ and user *Bob* has asset β in blockchain $\mathcal{B}_B$. An atomic swap is said to be successful when *Bob* transfers asset β to *Alice* on $\mathcal{B}_B$ contingent to the transfer of asset α by *Alice* to *Bob* on $\mathcal{B}_A$. If *Alice* decides to cancel the swap, a refund will be initiated. Upon asset refund, *Alice* will retain α in $\mathcal{B}_A$ and *Bob* will retain β in $\mathcal{B}_B$. A successful swap thereby leads to an exchange of asset's ownership [46]. Hence both the parties need to have accounts in each of the blockchains to enable transfer of ownership [31].

While one can easily envision an atomic swap functionality leveraging a trusted server, the blockchain community has put significant efforts into decentralized protocols for atomic swaps [1, 21, 29, 32, 33, 38–40, 48, 49]. In a nutshell, these different protocols follow a standard blueprint based on two building blocks: (i) a (cryptographic) locking mechanism that allows one user to locks coins for another user in a given blockchain; and (ii) a timeout mechanism that allows the creator of a lock to release it after a certain time has expired. With these building blocks, current atomic swap protocols are based on the following blueprint: first, *Alice* locks α in $\mathcal{B}_A$ for *Bob* and establishes an expiration time of T_A to such lock. Afterward, *Bob* locks β in $\mathcal{B}_B$ to *Alice* with an expiration time of $T_B : T_A > T_B$. At this point, the atomic swap has been committed and one of the following two outcomes can happen: (i) *Bob* allows *Alice* to unlock β in $\mathcal{B}_B$, which in turn "automatically" allows *Bob* to unlock α in $\mathcal{B}_A$; or (ii) both parties decide to abort the swap by allowing to release the locks at times T_B and T_A respectively.

This blueprint framework used by atomic swaps is based on two crucial properties. First, the (cryptographic) locks should allow to "relate" one to another in the sense that if one party opens one lock in one blockchain, such opening operation automatically reveals enough information to the other party to open her own lock in the other blockchain. Such "correlated locks" have been implemented in practice using different techniques such as leveraging the Turing-complete scripting language of blockchains like Ethereum [43] or more specific scripting functionality like Hash-time lock contract [9, 15, 21, 33], using a third blockchain [24, 25, 45] as the coordinator or bridge of the two blockchains [4, 26, 27, 37, 47] used for the swap , leveraging trusted hardware [8], or designing cryptographic schemes crafted for this purpose such as adaptor signatures [16, 40].

The second crucial property is that locked funds must be released to the original owner after a certain time has expired. Surprisingly, all alternative protocols previously mentioned share only two techniques with regard to handling the timelock functionality. They either (i) rely on the scripting language of the underlying blockchain to implement it; or (ii) rely on a cryptographic timelock puzzle [13, 36, 41] where a secret to unlock the locked funds are saved under a cryptographic puzzle that can be solved after a certain number of serial cryptographic operations are carried out. Unfortunately, both of these techniques clearly hinder the adoption of atomic swaps. On the one hand, timelock based on the scripting language restricts its use to those cryptocurrencies that do not have such support, such as Monero [34] or Zcash (shielded addresses) [23]. On the other hand, cryptographic puzzles impose a computation burden on the users that need to compute such a puzzle for each of the atomic swaps that they are involved in. Such a scheme is not suitable for lightweight applications as it would drain the battery of a smartphone or would add a non-trivial cost if outsourced to a third party (e.g., Amazon Web Services [14]).

In this state of affairs, we raise the following question: *Is the timelock functionality a necessary condition to design atomic swap*

protocols? Or in other words, *is it possible to design an atomic swap protocol such that the timelock functionality is not required in (at least one of) the two involved blockchains?*

1.1 Our contribution

In this work, we present for the first time a secure, decentralized, and trustless atomic swap protocol that does not require any type of timelock in one of the cryptocurrencies. In particular, we present LightSwap, a lightweight atomic swap between Bitcoin and Monero. Similar to previous works, LightSwap leverages adaptor signatures to implement the cryptographic condition that correlates the locks over the committed coins. The crux of the contribution in LightSwap is to depart from the 2-phase paradigm. Instead, we propose a novel paradigm that maintains the security for the users (i.e., an honest user does not lose coins) while removing the need to use timeouts in any form for one of the two cryptocurrencies.

2 NOTATION AND BACKGROUND

Transactions in UTXO model. In this work, we focus on the UTXO transaction model, as it is followed by both Bitcoin and Monero.

For readability, transaction charts are used to visualize the transactions, their ordering, and usage in any protocol. We follow the notation in [6]. The charts must be read from left to right as per the direction of the arrows. A transaction is represented as a rectangular box with a rounded corner, input to such transactions is denoted by incoming arrows and output by outgoing arrows. Each rectangular box has square boxes drawn within. These boxes represent the output of the transaction, termed as *output boxes*, and the value within represents the number of coins. Conditions for spending these coins are written on the output arrows going out of these boxes. The notations and the illustration of the transaction charts are provided in Figure 1.

The parties that can spend these coins present in the output box are represented below the outgoing arrows in form of a signature. Usually, these are represented as the public keys which can verify this signature. Additional conditions for spending the coins are written above the arrow. Conditions are encoded in a script supported by the underlying cryptocurrency. For our paper, we use the notation "$+t$" or $\texttt{RelTime}(t)$ which denotes the waiting time before a transaction containing an output can be published on-chain. This is termed as the *relative locktime*. If absolute locktime is used, then it is represented as "$\geq t$" or $\texttt{AbsTime}(t)$. It means the condition for spending the output is satisfied if the height of the blockchain is at least t. For representing multiple conditions, if it is a disjunction of several conditions, i.e. $\phi = \phi_1 \vee \phi_2 \vee \ldots \vee \phi_n$, a diamond-shaped box is used in the output box and each sub condition ϕ_i is written above the output arrow. The conjunction of several conditions is represented as $\phi = \phi_1 \wedge \phi_2 \wedge \ldots \wedge \phi_m$.

Adaptor signatures. We recall the functionality for generation and verification of adaptor signature with respect to a hard relation. This becomes one building block in our approach to substitute the functionality of HTLC. In more detail, given a hard relation $R : (x, X) \in R$, where X is the statement and x is a witness, public key pk having secret key sk, the language L_R and a signature scheme

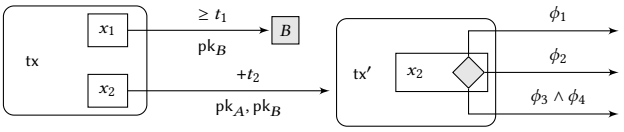

Figure 1: (Left) Transaction tx **has two outputs, one of value x_1 that can be spent by B (indicated by the gray box) with a transaction signed w.r.t.** pk_B **at (or after) round t_1, and one of value x_2 that can be spent by a transaction signed w.r.t.** pk_A **and** pk_B **but only if at least t_2 rounds passed since** tx **was accepted on the blockchain. (Right) Transaction** tx′ **has one input, which is the second output of** tx **containing x_2 coins and has only one output, which is of value x_2 and can be spent by a transaction whose witness satisfies the output condition $\phi_1 \vee \phi_2 \vee (\phi_3 \wedge \phi_4)$. The input of** tx **is not shown.**

$\Sigma = (\texttt{Gen}, \texttt{Sign}, \texttt{Vrfy})$, an adaptor signature is defined using four algorithms $\Xi_{R,\Sigma} = (\texttt{pSign}, \texttt{pVrfy}, \texttt{Adapt}, \texttt{Ext})$ as follows [5]:

- $\texttt{pSign}(sk, m, X)$: A probabilisitc polynomial time algorithm which on input of secret key sk, message $m \in \{0,1\}^*$ and statement $X \in L_R$, outputs an a pre-signature $\hat{\sigma}$.
- $\texttt{pVrfy}(pk, m, X, \hat{\sigma})$: A deterministic polynomial time algorithm which on input the public key pk, the message $m \in \{0,1\}^*$, the statement $X \in L_R$, and pre-signature $\hat{\sigma}$, outputs a bit b. If $b = 1$, $\hat{\sigma}$ is a valid pre-signature on message m.
- $\texttt{Adapt}(\hat{\sigma}, x)$: A deterministic polynomial time algorithm which on input the witness for the statement X, i.e. x and the pre-signature $\hat{\sigma}$, outputs a signature σ.
- $\texttt{Ext}(\sigma, \hat{\sigma}, X)$: A deterministic polynomial time algorithm which on input signature σ, pre-signature $\hat{\sigma}$ and the statement $X \in L_R$, outputs a witness $x : (x, X) \in R$ or $\perp$.

In this work, we leverage the threshold adaptor signature for ECDSA [30] for the Bitcoin side and the instance defined in [32, 42] for Monero. In a threshold adaptor signature instance, the secret key sk is shared by two participants, in our case Alice and Bob.

3 PROBLEM DEFINITION

Given a user *Alice* and the service provider *Bob*, the former holds x XMR in Monero blockchain and *Bob* holds y BTC in Bitcoin blockchain. *Alice* wants to exchange x XMR for *Bob*'s y BTC. A generic atomic swap protocol follows a 2-phase commit protocol similar to that used in databases: (i) each user commits their assets and (ii) each user claims the assets of the counterparty. To initiate an atomic swap, both the parties need to lock their coins and set a timeperiod within which the swap must be completed. If *Alice* wants to cancel the swap, she will initiate a refund and the locked coins are refunded to the original owner after the designated timeperiod.

Existing atomic swap protocols and their drawbacks. We discuss existing approaches as solution for the problem defined above. We denote *Alice* as **A** and *Bob* as **B**.

(i) *Using HTLC based approach.* The simplest trustless exchange protocol widely used across several cryptocurrency exchange is based on *Hash Timelocked Contract or HTLC*. We discuss an HTLC based solution where both **A** and **B** hold their coins at time t_0. The script used in HTLC takes the tuple $(\alpha, h, t, \mathbf{A}, \mathbf{B})$, where α is the asset to be transferred, h is the hash value, and t is the contract's

timeout period. The contract states that **A** will transfer α to **B** contingent to the knowledge r where $h = \mathsf{H}(r)$ where H is a standard cryptographic hash function if the contract is invoked within the timeout period t. If the timeperiod elapses and **B** fails to invoke the contract, the asset α is refunded to user **A**.

A can initiate exchange of x XMR in $\mathcal{B}_A$ for y BTC in $\mathcal{B}_B$ using HTLC. The former chooses a random value r and generates $h = \mathsf{H}(r)$. She next proceeds to lock x XMR in the contract $H_1 = HTLC(x, h, t_5, \mathbf{A,B})$ at time t_1, where $t_1 > t_0$, and sends h, t_5 to **B**. The timeout period of the contract is t_5. Now **B** will reuse the same terms of the contract but set the timeperiod as $t_4 : t_4 < t_5$. We will explain why the timeout period must be less than the previous contract. **B** locks y BTC in the contract $H_2 = HTLC(y, h, t_4, \mathbf{B,A})$ at time t_2, where $t_2 > t_1$. **A** knows the preimage of h and claim the coins from **B** by invoking H_2 at time $t_3 : t_2 < t_3 < t_4$. **B** gets the preimage r which he can use for claiming coins from **A**. If he had used the timeout period t_5 for H_2, then it is quite possible that **A** delays and claims the coins from **B** just at time t_5. This would lead to a race condition and **B** might fail to acquire the coins from **A** if the time at which H_1 is invoked exceeds t_5. Hence he sets the timeout period of the contract H_2 less than the timeout period of contract H_1. **B** claims the coins from **A** by invoking H_1 at time $t_4 : t_3 < t_4 < t_5$. By time t_5, **A** holds y BTC in $\mathcal{B}_B$ and **B** holds x XMR in $\mathcal{B}_A$. This depicts the situation when the swap succeeds and the state transition from time t_0 to t_5 discussed above is termed as *happy path*. If either of the party decides not to co-operate then it will lead to failure of swap.

Incompatibility of HTLC in scriptless cryptocurrencies (e.g., Monero). HTLC-based approach requires the use of timelock on both the Monero side as well as the Bitcoin side. The timeout mechanism is essential to allow users to recover their assets in the case the swap does not go through. Thus we require two main building blocks to implement atomic swaps for cryptocurrencies: an atomic locking mechanism and a timeout. However, the main challenge is that Monero does not support hashlock and timelock. Without these two features, it will not be possible for **A** to lock her coins at time t_1. The use of timelock puzzles will make our protocol unsuitable for lightweight applications. Hence none of the paths can be initiated.

(ii) *Without using HTLC for Monero.* A fix for the challenges faced in HTLC based protocol would be to design a protocol without having any hashlock and timelock at Monero side, but **B** uses HTLC for locking y BTC in $\mathcal{B}_B$. In Monero, coins locked in the address can be spend only by the party possessing the private key of that particular address. The modified protocol allows **A** to lock her coins in an address say pk, whose secret key is solely possessed by her. This will allow **A** to initiate a refund at her will. Let the secret key be s. She locks x XMR in address pk at time t_1. Using this secret key, she generates $h_s : h_s = \mathsf{H}(s)$. She shares h_s with **B**. The latter locks y BTC into $HTLC(y, h_s, t_4, \mathbf{B,A})$ at time t_2. For a successful swap, **A** invokes HTLC using the secret s at time t_3 and claims y BTC. **B** uses the secret key s to spend x XMR locked in address pk at t_4 and transfers it to his address in $\mathcal{B}_A$.

Attack on this approach. Apparently, it might look like we can accomplish the swap using this approach. However, the problem is now **A** can initiate a refund at any time she wants. Even if she initiates a refund after t_2, she can still invoke the HTLC as $t_2 < t_4$, and claim y BTC from **B**. The service provider **B** will lose his coins. To counter this problem, we can resort to 2-of-2 secret sharing where each half of the secret key s of address pk will be shared with **A** and **B**. This will make **A** dependent on **B** for issuing a refund, violating our objective. If **B** does not lock his coins at t_2, **A**'s coins will remain locked forever.

From the above discussion, it is clear that designing an efficient protocol without any kind of timeout in one of the two chains is a challenging task. We provide a high-level overview of our proposed solution in the next section.

4 OUR APPROACH

4.1 Solution overview

Our protocol must ensure that the party moving first is allowed to issue a refund without depending on the counterparty. However, it must also be ensured that if the swap is canceled, both the party must get a refund. Since Monero does not support timelocks, we need to design a protocol that leverages the timelock used in the Bitcoin script. We use threshold adaptor signature for seamless redemption and refund of coins without any party suffering a loss in the process.

Signing refund transaction in Monero. Consider an atomic swap where *Alice* (or **A**) wants to exchange her monero for *Bob*'s (or **B**) bitcoin. If she locks her coins in an address whose secret key is known to her, she can spend the coins at any time. It is better if the secret key is shared where each half is possessed by **A** and **B**. However, this would mean that **A** has to depend on **B** for initiating a refund. If **B** does not cooperate, then **A**'s coin will remain locked forever. Hence both of them must collaborate and sign the refund transaction even before **A** locks her coins. The signature generated uses *threshold version of adaptor signature*. To generate such a signature, **B** uses his portion of the secret key as well as a cryptographic condition, say R, to generate the incomplete signature. **A** can complete the signature using her share of the secret key and upon fulfilling the hard relation R inserted by **B**. On the Bitcoin side, once **A** invokes the redeem transaction, the coins can be redeemed by her only after a certain timeperiod, say t, elapses. In the meantime, if **B** finds that **A** has refunded her coins but still invoked the redeem transaction at the Bitcoin side, then he can publish his refund transaction within the timeperiod t. A valid signature for a refund transaction can be generated by providing a witness to the relation R. Once **A** has published her refund transaction on $\mathcal{B}_A$, **B** will know the witness and hence, he can claim a refund easily.

We now describe our proposed two-party atomic swap protocol ensuring that none of the parties lose coins in the process.

4.2 Protocol description

We present an atomic swap protocol where **A** wants to swap x_A coins for y_B coins of **B** locks his bitcoins. The protocol consists of six phases: *setup, lock, redeem, cancel, emergency refund* and *punish*. The transaction schema for BTC to XMR atomic swap is shown in Figure 2. x_A coins are held in blockchain $\mathcal{B}_A$ and y_B coins are held in blokchain $\mathcal{B}_B$.

Setup phase. In this phase, **A** and **B** jointly create the public key pk in $\mathcal{B}_A$ **B**'s collaboration. **A** uses pk to generate an address for locking her coins. Each party will generate one-half of the secret key,

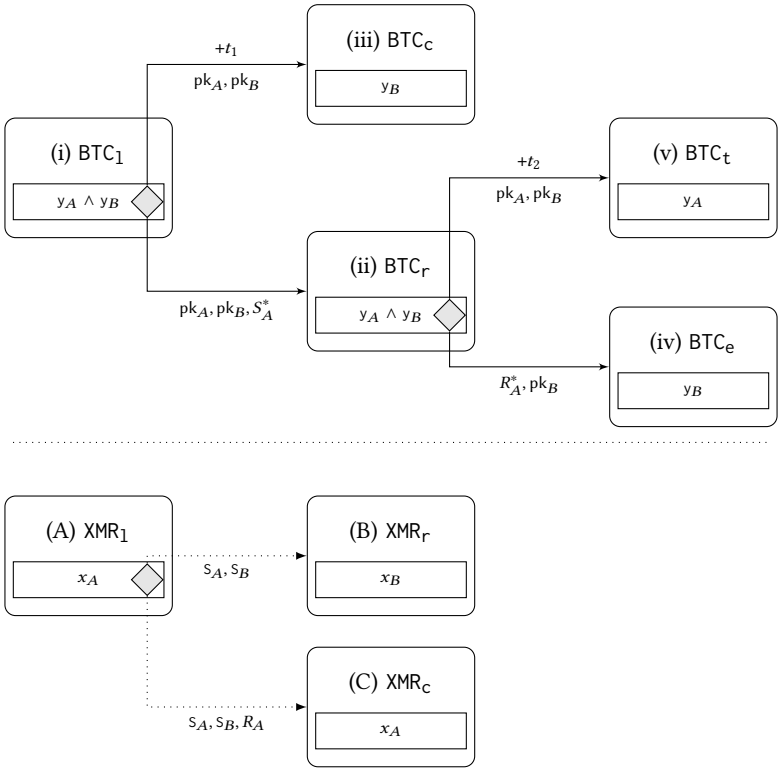

Figure 2: New transaction schema for BTC to XMR atomic swaps. *Top*: Transaction schema for Bitcoin. *Bottom*: Transaction schema for Monero. Here x_A and x_B denotes the fact that x Monero coins belong to either Alice or Bob correspondingly. Similarly with y_A and y_B in Bitcoin.

i.e., **A** will generate s_A, and **B** will generate s_B. A linear combination of their secret key will result in s. The latter serves as the private key of the address pk. Additionally, **A** samples an additional secret r_A and generates the statements R_A for $\mathcal{B}_A$ and R_A^* for $\mathcal{B}_B$. Thus, r_A is the witness to both the statements R_A and R_A^* Similarly, using one half of secret key, s_A, **A** generate the statements S_A and S_A^* for the blockchains $\mathcal{B}_A$ and $\mathcal{B}_B$ respectively. Similarly, **B** generates the statement S_B from s_B. Both parties share $(R_A, R_A^*, S_A, S_A^*, S_B)$.

Pre-signing of Monero Refund transaction: **A** creates a Monero refund transaction $\mathtt{XMR_c}$, box (C) in Figure 2, where x_A coins locked in address pk is send to another address on $\mathcal{B}_A$ controlled by **A**.

$$\mathtt{XMR_c} : \mathsf{pk} \xrightarrow{x_A} \mathbf{A}$$

Later, **A** and **B** collaborate and pre-sign $\mathtt{XMR_c}$ based on the statement R_A. Both the parties provide their share of private spend keys in the process of generating the adaptor signature without revealing it explicitly. This allows **A** to opt for a refund anytime she wants.

Exchanging signatures for the transactions on Bitcoin side: **B** shares his funding source, tx_{fund}, with **A**. The source has a balance of at least y_B coins. The transaction $\mathtt{BTC_1}$, box (i) in Figure 2, is created where **B** will lock his coins in a 2-of-2 multisig redeem script, $\mathsf{pk}_{A,B}^{lock}$. The output is denoted as $y_A \wedge y_B$.

$$\mathtt{BTC_1} : tx_{fund} \xrightarrow{y_A \wedge y_B} \mathsf{pk}_{A,B}^{lock}$$

The coins can either be redeemed by **A** or refunded by **B** after a certain timeperiod t_1. **A** can publish the transaction $\mathtt{BTC_r}$, box (ii) in Figure 2, spends the output of $\mathtt{BTC_1}$ and again locks into a 2-of-2 multisig redeem script, $\mathsf{pk}_{A,B}^{redeem}$.

$$\mathtt{BTC_r} : \mathsf{pk}_{A,B}^{lock} \xrightarrow{y_A \wedge y_B} \mathsf{pk}_{A,B}^{redeem}$$

The output of $\mathtt{BTC_r}$ can either be refunded to **B**, if there is an emergency, or it can be claimed by **A** after a certain timeperiod t_2. **A** creates the transaction $\mathtt{BTC_t}$, box (v) in Figure 2, which will allow her to spend the output of $\mathtt{BTC_r}$ after timeperiod t_2 and shares it with **B**.

$$\mathtt{BTC_t} : \mathsf{pk}_{A,B}^{redeem} \xrightarrow{y_A} \mathbf{A}$$

The latter signs the transaction and sends it to **A**. Next, **B** creates the transaction $\mathtt{BTC_c}$, box (iii) in Figure 2, that will allow him to refund the output $y_A \wedge y_B$ coins of $\mathtt{BTC_1}$.

$$\mathtt{BTC_c} : \mathsf{pk}_{A,B}^{lock} \xrightarrow{y_A} \mathbf{B}$$

He shares the transaction with **B**, the latter signs and sends it to **A**. **B** verifies the transaction, pre-signs the transaction $\mathtt{BTC_r}$ based on the statement S_A^* and sends the partially signed transaction to **A**. Now **A** will sign the transaction $\mathtt{BTC_1}$ and send it to **B**.

Lock phase. **A** creates the transaction $\mathtt{XMR}_1$, box (A) in Figure 2 where she locks x_A coins into address pk.

$$\mathtt{XMR}_1 : \mathbf{A} \xrightarrow{x_A} \mathsf{pk}$$

B, upon verification that **A** has locked the coins, proceeds with publishing $\mathtt{BTC}_1$ and locks his coins as well.

Redeem phase. **A** knows the witness s_A for the statement S_A^* and thus she generates a valid signature for $\mathtt{BTC}_\mathsf{r}$. She publishes the transaction but cannot spend the output before a timperiod of t_2 has elapsed. Meanwhile, **B** extracts s_A from the signature on $\mathtt{BTC}_\mathsf{r}$. He will create the transaction $\mathtt{XMR}_\mathsf{r}$, box (B) in Figure 2 that will allow him to redeem the coins locked in address pk.

$$\mathtt{XMR}_\mathsf{r} : \mathsf{pk} \xrightarrow{x_B} \mathbf{B}$$

By combining the secret keys s_A and s_B, he will be able to sign $\mathtt{XMR}_\mathsf{r}$ and publish it on-chain.

Cancel swap. If **A** wants to cancel the swap, she will generate a valid signature for $\mathtt{XMR}_\mathsf{c}$ using the witness r_A and publish it to claim her coins. Meanwhile, **B** can wait till t_1 has elapsed since $\mathtt{BTC}_1$ was published and **A** has not initiated the swap. He publishes $\mathtt{BTC}_\mathsf{c}$ and unlocks his coins.

Emergency refund. Suppose **A** has initiated the swap by publishing $\mathtt{BTC}_\mathsf{r}$ but she has unlocked her coins by publishing $\mathtt{XMR}_\mathsf{c}$. Once $\mathtt{XMR}_\mathsf{c}$ is published, **B** extracts r_A from the signature on $\mathtt{XMR}_\mathsf{c}$. He will create transaction $\mathtt{BTC}_\mathsf{e}$, box (iv) in Figure 2 and spend $y_A \wedge y_B$ coins locked in $\mathsf{pk}_{A,B}^{redeem}$.

$$\mathtt{BTC}_\mathsf{e} : \mathsf{pk}_{A,B}^{redeem} \xrightarrow{y_B} \mathbf{B}$$

Now he will sign the transaction using r_A and publish the transaction on-chain before t_2 elapses.

From the above discussion on *emergency refund*, we emphasize the utility of not allowing **A** to redeem the coins locked by **B**. Instead, a waiting time of t_2 allows **B** to recover his coins, if **A** is malicious. On one hand, **A** can initiate a refund any time she wants but on the other hand, she cannot claim the bitcoins instantly.

Punish. If **B** has published $\mathtt{XMR}_\mathsf{r}$ and claimed x_B coins, then **A** waits for t_2 timeperiod to elapse after publishing $\mathtt{BTC}_\mathsf{r}$. She will publish $\mathtt{BTC}_\mathsf{t}$ and claim y_A coins.

Now, consider that **B** has stopped responding and has neither claimed x_B coins nor initiated a refund. In that case, **A** can *punish* him for remaining inactive by publishing $\mathtt{BTC}_\mathsf{t}$. Hence, this phase is called *punish* phase and **B** loses his bitcoins. A detailed description of the protocol can be found in the full version of our paper [3].

4.3 Security and privacy goals

- **Correctness**: If both parties are honest, with one party willing to exchange x units of coin for y units of coins of the other party, then the protocol terminates with each party obtaining the desired amount.
- **Soundness**: An honest party must not lose funds while executing the protocol with an adversary.
- **Unlinkability**: Any party not involved with the atomic swap must not be able to link two cross-chain transactions responsible for the atomic swap, except with negligible probability.
- **Fungibility**: An adversary must not be able to distinguish between a normal transaction and a transaction for atomic swap in Monero Blockchain, except with negligible probability.

We discuss how the security properties defined above holds for our proposed protocol:

- **Correctness**: If both parties **A** and **B** are honest, then the atomic swap protocol ensures that if party **A** is able to redeem y_A coins then party B can redeem x_B coins as well within a bounded timeperiod. This is possible since when **A** publishes $\mathtt{BTC}_\mathsf{r}$, **B** extracts the secret s_A from signature on $\mathtt{BTC}_\mathsf{r}$ and uses the same for signing transaction $\mathtt{XMR}_\mathsf{r}$.
- **Soundness**: If party **A** initiates the swap but publishes $\mathtt{XMR}_\mathsf{c}$ before **B** publishes $\mathtt{XMR}_\mathsf{r}$, then a relative locktime of t_2 on spending the output of $\mathtt{BTC}_\mathsf{r}$ allows **B** to opt for an emergency refund by publishing $\mathtt{BTC}_\mathsf{e}$ and refund his coins.
- **Linkability**: Since Monero transactions are confidential and signatures on transactions are generated from random values, any malicious party observing both the Monero and Bitcoin blockchains will be able to link a pair of Bitcoin and Monero transactions involved in the swap with negligible probability.
- **Fungibility**: There is no structural difference between a normal Monero transaction and a Monero transaction constructed for LightSwap. Any malicious party observing the Monero blockchain can distinguish between such a pair of transactions with negligible probability.

5 OUR CONSTRUCTION

We state the security and privacy objectives to be realized by our proposed protocol, followed by the system model and cryptographic building blocks.

5.1 Security and privacy goals

- **Correctness**: If both parties are honest, with one party willing to exchange x units of coin for y units of coins of the other party, then the protocol terminates with each party obtaining the desired amount.
- **Soundness**: An honest party must not lose funds while execution of the protocol with an adversary.
- **Unlinkability**: Any party not involved with the atomic swap must not be able to link two cross-chain transactions responsible for the atomic swap, except with non-negligible probability.
- **Fungibility**: An adversary must not be able to distinguish between a normal monero transaction and a transaction for atomic swap, except with negligible probability in Monero.

5.2 System assumptions

We assume that any transaction broadcasted, will get eventually added in the ledger within a certain timeframe. Public keys of the parties involved in the swap are known to the rest of the participants. Any honest party willing to execute the swap will remain online until it acquires the coins of the counterparty.

5.3 Cryptographic building blocks

Monero Ring Signature for Untraceability. Monero account possess two private keys - *spend key* and *view key* [44]. Private view key allows a user to view the fund locked in the account and private spend key allows the user to spend the fund. A one-time address can be constructed using the public keys generated using pair of private spend and view keys. Spender of monero fund uses set of random one-time public addresses from the blockchain creates a one-time ring signature using the private key of his address. Anyone verifying the signature will know coins present in one of the address has been spent. The *Cryptonote* Protocol [44] mentions the use of type of one-time linkable spontaneous anonymous group (LSAG) signature. Monero uses multilayered LSAG (MLSAG) in order to handle anonymous transaction.

Monero Address Generation. We briefly describe the signing and verification phase LSAG [34]:

- Signature Generation
  - Let $G = \langle g \rangle$ be a group of prime order q. We define two statistically independent hash functions $\mathcal{H}_1 : \{0,1\}^* \rightarrow \mathbb{Z}_q$ and $\mathcal{H}_2 : \{0,1\}^* \rightarrow G$.
  - The user chooses $n-1$ distinct public keys $P_1, P_2, \ldots, P_{j-1}, P_{j+1} \ldots, P_n$ where $n > 1$ where each $P_i = x_i G, x_i \in \mathbb{Z}_q^*$. He adds P_j, his own public key, to the set. The key image corresponding to P_j is $I = x_j \mathcal{H}_2(P_j)$.
  - The user selects $q_i \xleftarrow{\$} \mathbb{Z}_q, i \in [1,n]$. He sets $L_j = q_j G$, $R_j = q_j \mathcal{H}_2(P_j)$ and sets $c_{j+1} = \mathcal{H}_1(m, L_j, R_j)$.
  - He sets $L_k = q_k G + c_k P_k$, $R_k = q_k \mathcal{H}_2(P_k) + c_k I$ and $c_{k+1} = \mathcal{H}_1(m, L_k, R_k), k \in [j+1, n]$ where $c_1 = c_{n+1}$.
  - He sets $L_i = q_i G + c_i P_i$, $R_i = q_i \mathcal{H}_2(P_i) + c_i I$ and $c_{i+1} = \mathcal{H}_1(m, L_i, R_i), i \in [1, j-1]$. Hence, $L_j = q_j G - c_j P_j + c_j P_j = (q_j - c_j x_j)G + c_j P_j$ and $R_j = q_j \mathcal{H}_2(P_j) - c_j I + c_j I = (q_j - c_j x_j)\mathcal{H}_2(P_j) + c_j I$. Thus $q'_j = q_j - c_j x_j$.
  - The signer sends the ring signatures $\sigma = (I, c_1, q_1, q_2, \ldots, q_{j-1}, q'_j, q_{j+1}, \ldots, q_n)$.
- Signature Verification
  - Verifier constructs L_i and $R_i, i \in [1,n]$ and checks if $c_1 \stackrel{?}{=} \mathcal{H}_1(m, L_n, R_n)$. If so then the signature is considered as valid.
- Linkability: Given two valid signature for different messages, if they have the same key image then one of the signature is rejected else it would lead to double spending.

Non-interactive Zero Knowledge Proofs. In order to prove a statement which belongs to class $\mathcal{NP}$ without revealing the witness, we need the help of zero knowledge proof for convincing a verifier. Given a pair of probabilisitc polynomial time algorithm P and V, with P having statement X and witness x such that $(X, x) \in R$ where R is a *hard relation* [17]. We define the language $L_R = \{X : \exists x : (X, x) \in R\}$ as $\mathcal{NP}$-language. P outputs a proof π for the statement X. V on getting X and proof π checks if proof establishes the fact that P knows a witness x for the statement X.

Proof of Discrete Logarithm. We define the two algorithms involved in the proof:

- $\pi \leftarrow \mathsf{P}_{\mathsf{DL}}((G, X), x)$: Using a probabilistic proving algorithm P_{DL}, which on input G, statement X and the witness for statement X, termed as x, generates the proof π.
- $\{0,1\} \leftarrow \mathsf{V}_{\mathsf{DL}}((G, X), \pi)$: Using a deterministic verification algorithm, the verifier on obtaining the statement X, the generator G, and the proof for possessing the witness of statement X, denoted as π, checks whether $\mathsf{V}_{\mathsf{DL}}((G, X), \pi) \stackrel{?}{=} 1$. If the output is 0, then the proof is rejected.

Given below is the *Sigma Protocol* for the relation $\{(X, x) : X = xG\}$.

Prover(x, X)		**Verifier**(X)
$t \leftarrow \mathbb{Z}_q^*$		
$T \leftarrow tG$		
$c \leftarrow \mathcal{H}(X, T)$		
$z \leftarrow t + cx \mod q$	$\xrightarrow{T\ c\ z}$	
		$zG \stackrel{?}{=} T + cX$

Proof of Cross-Chain Discrete Logarithmic Equality. Let us denote the groups used for Monero and Bitcoin be $\mathbb{G}$ and $\mathbb{H}$ respectively. We consider G and G' as generators of group $\mathbb{G}$ and H, H' be generators of group $\mathbb{H}$, where $|G| = p$ and $|H| = q$, p and q being large prime numbers, $p \leq q$. Let us define two cryptographic hash functions $\mathsf{H}_{\mathbb{G}} : \{0,1\}^* \rightarrow \mathbb{Z}_p$ and $\mathsf{H}_{\mathbb{H}} : \{0,1\}^* \rightarrow \mathbb{Z}_q$. Given the values $X = xG'$ and $Y = xH'$ where $x \in \mathbb{Z}, 0 \leq x < p$, a prover must prove that discrete logarithm of X and Y is a representation of the same integer without revealing the value x.

We define the prover and verifier algorithm:

- $\pi \leftarrow \mathsf{P}_{\mathsf{DLEQ}}((G, X), (H, Y), x)$: Bit representation of integer x is:

$$x = \sum_{i=0}^{n-1} b_i 2^i \tag{1}$$

Each element b_i is an element of either $\mathbb{Z}_p$ or $\mathbb{Z}_q$, depending on whether x is discrete logarithm of X or Y. For the n-bit number, generate $n-1$ pair of blinding values $f_i \in \mathbb{Z}_p$ and $g_i \in \mathbb{Z}_q, i \in [0, n-2]$. The values f_{n-1} and g_{n-1} are set as follows:

$$\begin{aligned} f_{n-1} &= -(2^{n-1})^{-1} \textstyle\sum_{i=0}^{n-2} f_i 2^i \in \mathbb{Z}_p \\ g_{n-1} &= -(2^{n-1})^{-1} \textstyle\sum_{i=0}^{n-2} g_i 2^i \in \mathbb{Z}_q \end{aligned} \tag{2}$$

This gives us the relation $\sum_{i=0}^{n-1} f_i 2^i = \sum_{i=0}^{n-1} g_i 2^i = 0$. The values $f_i, g_i \in [0, n-1]$ are used for computing the Pedersen Commitments of each bit b_i.

$$\begin{aligned} C_i^G &= b_i G' + f_i G \in \mathbb{G} \\ C_i^H &= b_i H' + g_i H \in \mathbb{H} \end{aligned} \tag{3}$$

Taking the weighted sum, we have the following relations: $X = \sum_{i=0}^{n-1} 2^i C_i^G = xG'$ and $Y = \sum_{i=0}^{n-1} 2^i C_i^H = xH'$. Ring signature is constructed on each bit to show that it is either 0 or 1, and next it is shown that the value is same in both the groups. We analyze for both the cases (i) if $b_i = 0, i \in [0, n-1]$, (ii) if $b_i = 1, i \in [0, n-1]$.

– $b_i = 0$: Consider pairs of random values $j_i \in \mathbb{Z}_p$ and $k_i \in \mathbb{Z}_q$, $i \in [0, n-1]$ and set $e^G_{1,i}, e^H_{1,i}$ as follows:

$$e^G_{1,i} = \mathsf{H}_{\mathbb{G}}(C^G_i, C^H_i, j_iG, k_iH) \in \mathbb{Z}_p$$
$$e^H_{1,i} = \mathsf{H}_{\mathbb{H}}(C^G_i, C^H_i, j_iG, k_iH) \in \mathbb{Z}_q \tag{4}$$

Randomly select n pairs of $a_{0,i} \in \mathbb{Z}_p$ and $b_{0,i} \in \mathbb{Z}_q$, set $e^G_{0,i}, e^H_{0,i}$, $i \in [0, n-1]$.

$$e^G_{0,i} = \mathsf{H}_{\mathbb{G}}(C^G_i, C^H_i, a_{0,i}G - e^G_{1,i}(C^G_i - G'), b_{0,i}H - e^H_{1,i}(C^H_i - H')) \in \mathbb{Z}_p$$
$$e^H_{0,i} = \mathsf{H}_{\mathbb{H}}(C^G_i, C^H_i, a_{0,i}G - e^G_{1,i}(C^G_i - G'), b_{0,i}H - e^H_{1,i}(C^H_i - H')) \in \mathbb{Z}_q \tag{5}$$

- – The n pair of values $a_{1,i}$ and $b_{1,i}$ are defined as:

$$a_{1,i} = j_i + e^G_{0,i} f_i \in \mathbb{Z}_p$$
$$b_{1,i} = k_i + e^H_{0,i} g_i \in \mathbb{Z}_q \tag{6}$$

– $b_i = 1$: Consider pairs of random values $j_i \in \mathbb{Z}_p$ and $k_i \in \mathbb{Z}_q$, $i \in [0, n-1]$ and set $e^G_{1,i}, e^H_{1,i}$

$$e^G_{0,i} = \mathsf{H}_{\mathbb{G}}(C^G_i, C^H_i, j_iG, k_iH) \in \mathbb{Z}_p$$
$$e^H_{0,i} = \mathsf{H}_{\mathbb{H}}(C^G_i, C^H_i, j_iG, k_iH) \in \mathbb{Z}_q \tag{7}$$

Randomly select n pairs of $a_{1,i} \in \mathbb{Z}_p$ and $b_{1,i} \in \mathbb{Z}_q$, set $e^G_{1,i}, e^H_{1,i}$, $i \in [0, n-1]$ as follows:

$$e^G_{1,i} = \mathsf{H}_{\mathbb{G}}(C^G_i, C^H_i, a_{0,i}G - e^G_{1,i}C^G_i, b_{0,i}H - e^H_{1,i}C^H_i) \in \mathbb{Z}_p$$
$$e^H_{1,i} = \mathsf{H}_{\mathbb{H}}(C^G_i, C^H_i, a_{0,i}G - e^G_{1,i}C^G_i, b_{0,i}H - e^H_{1,i}C^H_i) \in \mathbb{Z}_q \tag{8}$$

The n pair of values $a_{0,i}$ and $b_{0,i}$ are defined as:

$$a_{0,i} = j_i + e^G_{1,i} f_i \in \mathbb{Z}_p$$
$$b_{0,i} = k_i + e^H_{1,i} g_i \in \mathbb{Z}_q \tag{9}$$

The proof π is the tuple $(X, Y, \{C^G_i\}, \{C^H_i\}, \{e^G_{0,i}\}, \{e^H_{0,i}\}, \{a_{0,i}\}, \{a_{1,i}\}, \{b_{0,i}\}, \{b_{1,i}\})$

- $\{0,1\} \leftarrow \mathsf{V}_{\mathsf{DLEQ}}((G,X),(H,Y),\pi)$: The verifier upon receiving tuple checks the following conditions:

$$\textstyle\sum_{i=0}^{n-1} 2^i C^G_i \stackrel{?}{=} X \in \mathbb{G}$$
$$\textstyle\sum_{i=0}^{n-1} 2^i C^H_i \stackrel{?}{=} Y \in \mathbb{H} \tag{10}$$

For each $i \in [0, n-1]$, it computes the following:

$$e^G_{1,i} = \mathsf{H}_{\mathbb{G}}(C^G_i, C^H_i, a_{1,i}G - e^G_{0,i}C^G_i, b_{1,i}H - e^H_{0,i}C^H_i \in \mathbb{Z}_p$$
$$e^H_{1,i} = \mathsf{H}_{\mathbb{H}}(C^G_i, C^H_i, a_{1,i}G - e^G_{0,i}C^G_i, b_{1,i}H - e^H_{0,i}C^H_i \in \mathbb{Z}_q$$
$$(e^G_{0,i})' = \mathsf{H}_{\mathbb{G}}(C^G_i, C^H_i, a_{0,i}G - e^G_{1,i}(C^G_i - G'), b_{0,i}H - e^H_{1,i}(C^H_i - H')) \in \mathbb{Z}_p$$
$$(e^H_{0,i})' = \mathsf{H}_{\mathbb{H}}(C^G_i, C^H_i, a_{0,i}G - e^G_{1,i}(C^G_i - G'), b_{0,i}H - e^H_{1,i}(C^H_i - H')) \in \mathbb{Z}_q \tag{11}$$

It next checks the following conditions:

$$e^G_{0,i} \stackrel{?}{=} (e^G_{0,i})'$$
$$e^H_{0,i} \stackrel{?}{=} (e^H_{0,i})' \tag{12}$$

If all the checks hold true, then the verifier accepts the proof else it rejects it.

Adaptor Signature. We define the modules for generation and verification of adaptor signature with respect to a hard relation. Given a hard relation $R : (x, X) \in R$, where X is the statement and x is a witness, public key pk having secret key sk, the language L_R and a signature scheme $\Sigma = (\mathtt{Gen}, \mathtt{Sign}, \mathtt{Vrfy})$, an adaptor signature is defined using four algorithms $\Xi_{R,\Sigma} = (\mathtt{pSign}, \mathtt{pVrfy}, \mathtt{Adapt}, \mathtt{Ext})$ as follows [5]:

- $\mathtt{pSign}(sk, m, X)$: A probabilisitc polynomial time algorithm which on input of secret key sk, message $m \in \{0,1\}^*$ and statement $X \in L_R$, outputs an a pre-signature $\hat{\sigma}$.
- $\mathtt{pVrfy}(pk, m, X, \hat{\sigma})$: A deterministic polynomial time algorithm which on input the public key pk, the message $m \in \{0,1\}^*$, the statement $X \in L_R$, and pre-signature $\hat{\sigma}$, outputs a bit b. If $b = 1$, $\hat{\sigma}$ is a valid pre-signature on message m.
- $\mathtt{Adapt}(\hat{\sigma}, x)$: A deterministic polynomial time algorithm which on input the witness for the statement X, i.e. x and the pre-signature $\hat{\sigma}$, outputs a signature σ.
- $\mathtt{Ext}(\sigma, \hat{\sigma}, X)$: A deterministic polynomial time algorithm which on input signature σ, pre-signature $\hat{\sigma}$ and the statement $X \in L_R$, outputs a witness $x : (x, X) \in R$ or $\perp$.

ECDSA-based Adaptor Signature. We discuss this signature scheme since ECDSA-based signature is used in bitcoin script. A pre-signature (r, s) is constructed for statement X, embedding X in the r-component. The signer must get a zero knowledge proof of a valid witness for statement X. The four algorithms $\mathtt{pSign}, \mathtt{pVrfy}, \mathtt{Adapt}, \mathtt{Ext}$ are defined as follows [5]:

$\underline{\mathtt{pSign}(sk, m, X)}$
$\overline{x} := sk$
$k \xleftarrow{\$} \mathbb{Z}_q$
$\overline{K} := kG, K := kX$
$(r_x, r_y) := K$
$s' := k^{-1}(\mathcal{H}(m) + r_x.\overline{x})$
$\pi \leftarrow \mathsf{P}_{\mathsf{DLEQ}}((G, \overline{K}), (X, K), \overline{x})$
return $\hat{\sigma} = (r_x, s', K, \pi)$

$\underline{\mathtt{Adapt}(\hat{\sigma}, x)}$
$(r_x, s', K, \pi) := \hat{\sigma}$
$s := s'x^{-1}$
return $\sigma = (r_x, s)$

$\underline{\mathtt{pVrfy}(pk, m, X, \hat{\sigma})}$
$(r_x, s', K, \pi) := \hat{\sigma}$
$u := \mathcal{H}(m)s'$
$v := rs'^{-1}$
$K' = uG + vpk$
$b_1 := ((r_x, r_y) \stackrel{?}{=} K)$
$b_2 := \mathsf{V}_{\mathsf{DLEQ}}((G, K'), (X, K), \pi)$
return $b_1 \wedge b_2$

$\underline{\mathtt{Ext}(\sigma, \hat{\sigma}, X)}$
$(r_x, s) := \sigma$
$(r_x, s', K, \pi) := \hat{\sigma}$
$x' := s^{-1}s'$
if $(x', X) \in R$
  then return x'
else
  return $\perp$

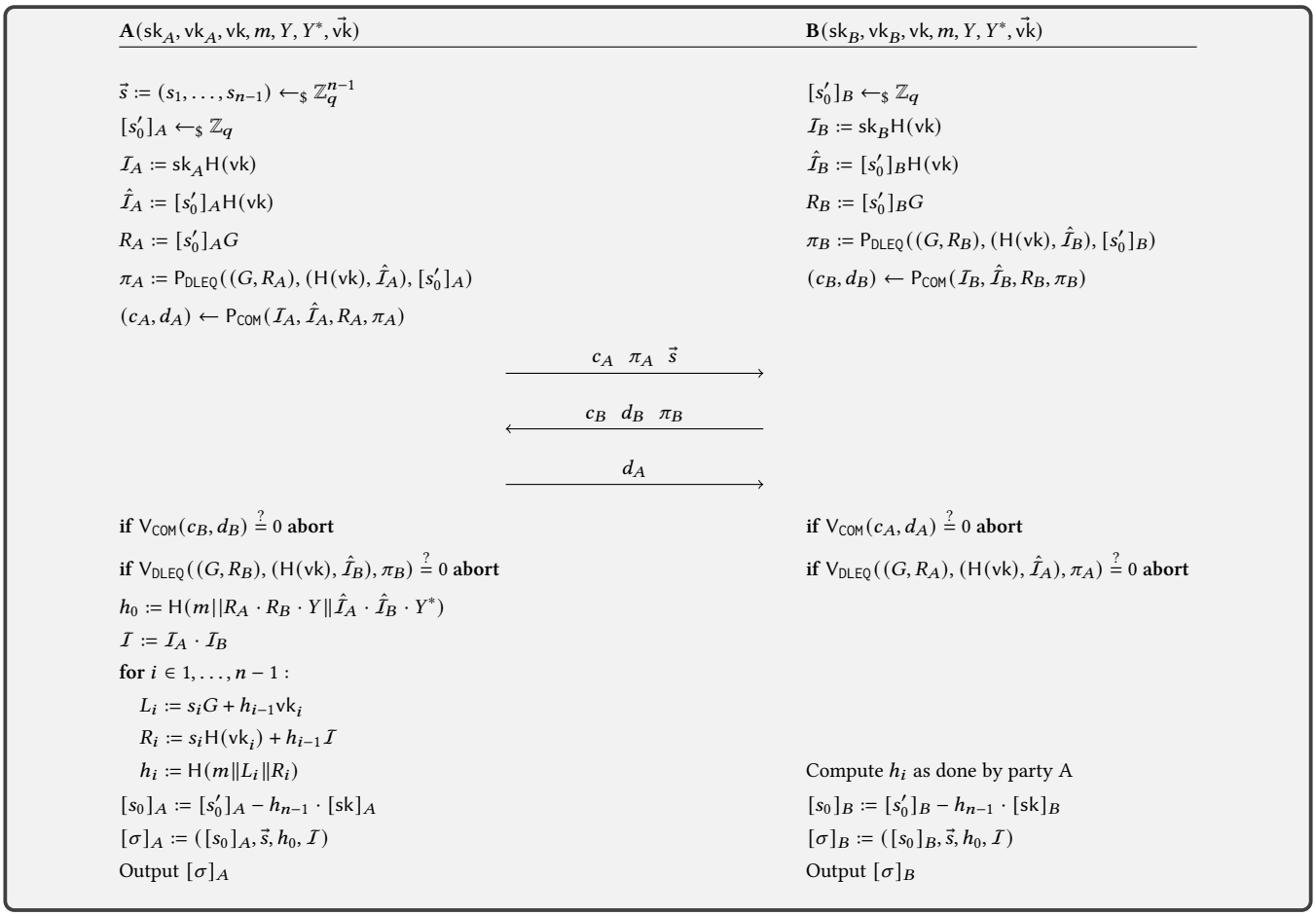

Figure 3: Threshold Adaptor Ring signature: Pre-ring signature generation

Threshold Adaptor Signatures. Given public keys pk_1 and pk_2, with corresponding secret keys sk_1 and sk_2, the two-party computation of threshold adaptor signature [2] comprises two sub-protocols $pSign2$ and $pVryf2$ defined as follows:

- $\mathtt{pSign2}(sk_1, sk_2, m, X)$: Upon input of secret keys sk_1, sk_2, message $m \in \{0,1\}^*$ and statement $X \in L_R$, the protocol outputs an a pre-signature $\hat{\sigma}$.
- $\mathtt{pVrfy2}(pk_1, pk_2, m, X, \hat{\sigma})$: Given the public keys pk_1, pk_2, the message $m \in \{0,1\}^*$, the statement $X \in L_R$, and pre-signature $\hat{\sigma}$ as input, the protocol outputs a bit b. If $b = 1$, $\hat{\sigma}$ is a valid pre-signature on message m.

We discuss the two-party threshold adaptor ring signature [32]. **A** samples a secret y and generates the statements Y and Y^*. A pre-signature is constructed for a statement Y and both the parties embed their secret share into the pre-signature. We consider here that **A** and **B** are involved in generating the pre-signature and later, **B** completes the signature using witness for Y. Given the secret share of **A** as sk_A and secret share of party **B** as sk_B, the steps in signature generation are as follows:

- `Key generation`: $((\mathsf{sk}_A, \mathsf{vk}_A, \mathsf{vk}), (\mathsf{sk}_B, \mathsf{vk}_B, \mathsf{vk})) \leftarrow \mathsf{KeyGen}\ \langle(\lambda), (\lambda)\rangle$. **A** generates $\mathsf{vk}_A := \mathsf{sk}_A G$ with proof $\pi_A = \mathsf{P}_{\mathsf{DL}}((G, \mathsf{vk}_A), \mathsf{sk}_A)$, **B** generates $\mathsf{vk}_B := \mathsf{sk}_B G$ with proof $\pi_B = \mathsf{P}_{\mathsf{DL}}((G, \mathsf{vk}_B), \mathsf{sk}_B)$. Both of them exchange their public keys along with the proof of knowledge of secret keys to construct $\mathsf{vk} = \mathsf{vk}_A + \mathsf{vk}_B$.
- `Relation generation`: The hard relation is abstracted away with the following algorithm $(y, Y, Y^*) \leftarrow \mathsf{RelGen}(\lambda, \mathsf{vk})$, where $Y := yG$ and $Y^* := yH(\mathsf{vk})$. The proof of cross-chain discrete logarithm equality $\pi_Y = \mathsf{P}_{\mathsf{DLEQ}}((G, Y), (H(\mathsf{vk}), Y^*), y)$ is generated by **A** and (Y, Y^*, π_Y) is shared with **B**. Given the statement Y and Y^*, the latter verifies whether both the statement shares the same witness.
- `Pre-ring signature generation`: Given n is the size of the ring, the set $\vec{\mathsf{vk}}$ is a set of n public keys where any $n-1$ public keys $vk_1, vk_2, \ldots, vk_{n-1}$ is chosen from the participants in the Monero blockchain and the n^{th} public key is vk. The pre-signature is a two party protocol, denoted as $\mathtt{pSign2}(sk_A, sk_B, m, Y)$, used to generate a partial signature on message m. As output, **A** generates $[\sigma]_A$ and **B** generates $[\sigma]_B$. The details have been provided in Fig. 3.

Functionality $\mathcal{L}(\texttt{Valid})$

Data: A list L that stores tuples of the form (tx_{id}, tx), where tx is the transaction identified by tx_{id}.

Operations:

- `(SendTx,id,tx)` On input a transaction `tx`, compute $b \leftarrow$ `Valid(tx)`. If $b = 0$ return $\perp$, otherwise compute a fresh transaction id tx_{id}, append (tx_{id}, tx) to L and return tx_{id}.
- `(ReadTx,id,`$\texttt{tx}_{\texttt{id}}$`)` On input a transaction identifier $\texttt{tx}_{\texttt{id}}$, check if there exists an entry in L of the form $((tx^*_{id}, tx))$ such that $tx^*_{id} = \texttt{tx}_{\texttt{id}}$ and return tx. Otherwise, return $\perp$.

Figure 4: Ledger functionality $\mathcal{L}$ using as parameter a predicate `Valid`.

- `Verification of pre-ring signature and Adapt phase`: **B** sends $[\sigma]_B$ to **A**. The latter verifies that $[\sigma]_B$ is valid "half-signature". **A** sends $[\sigma]_A$ to **B**. When **A** learns the witness y, it can compute the full signature $\sigma := [\sigma]_B + [\sigma]_A + y$.
- `Extraction of witness`: From σ, **B** can get the value y as $y := \sigma - ([\sigma_B] + [\sigma]_A)$.

5.4 Formal Description of the protocol

The protocol $\Pi_{LightSwaps}$ defined in the $(\mathcal{G}_{clock}, \mathcal{F}_{smt}, \mathcal{L})$-hybrid world, is shown in Figure 5. In the next section, we define the global ledger functionality $\mathcal{L}$ before providing a detailed description of the protocol.

5.4.1 Global Ledger Functionality. We define the interfaces for global ledger $\mathcal{L}$, as shown in Figure 4. It is parameterized by the predicate `Valid` and has a list L which maintains the set of all valid transactions. The list L is publicly accessible. `Valid` checks whether a given transaction spends an unspent output or the terms mentioned in the redeem script is not violated. The functions defined in the interfaces are:

- `(SendTx,id,tx)`: This function enables mining of valid transaction into the ledger $\mathcal{L}$. It is first checked whether the transaction denoted as `tx` is valid or not, by running $b \leftarrow$`Valid(tx)`. If $b = 1$, then the transaction gets mined and the required amount mentioned in `tx` is send from one address to another address and return transaction id tx_{id} to the function caller from session id id.
- `(ReadTx,id,`$\texttt{tx}_{\texttt{id}}$`)`: This function checks whether the transaction denoted by tx_{id} exists in the ledger $\mathcal{L}$. If so, then it returns to the transaction tx to the function caller from session id id.

5.4.2 Detailed Description. The protocol interacts with two instances of the global ledger $\mathcal{L}$: $\mathcal{L}_{\texttt{BTC}}$ depicting the Bitcoin ledger and $\mathcal{L}_{\texttt{XMR}}$ depicting the Monero ledger. Each ledger uses different instance of the predicate `Valid`, suitable for Bitcoin and Monero transaction. We define two functions: `TxXMR` and `TxBTC`, which creates Monero and Bitcoin transactions respectively. Any off-chain communication between **A** and **B** occurs using ideal functionality for secure message transmission, $\mathcal{F}_{smt}$ [11]. All honest parties follow the global clock $\mathcal{G}_{clock}$ [7] in order to proceed to the next round and keep track of the elapsed time.

The protocol is divided into the following operations:
(i) Lock operation - **A** locks x_A coins into ledger $\mathcal{L}_{\texttt{XMR}}$. **B** checks whether **A** has locked her coins, then he locks y_B coins in $\mathcal{L}_{\texttt{BTC}}$.

(ii) Swap Initiate Operation - **A** uses her share of secret key s_A to generate a valid signature for $\texttt{BTC}_\texttt{r}$. It is then published in $\mathcal{L}_{\texttt{BTC}}$ but **A** cannot claim x_A coins before elapse of time t_2.

(iii) Redeem Operation - **B** extracts the secret s_A from the signature $\sigma^{redeem}_{btc,\texttt{pk}_B}$, uses it to create secret $s_A + s_B$ for generating ring signature σ^{redeem}_{xmr}. Using this signature, it publishes the transaction $\texttt{XMR}_\texttt{r}$ in $\mathcal{L}_{\texttt{XMR}}$. After t_2 units elapses from the point of publishing $\texttt{BTC}_\texttt{r}$, **A** checks if **B** has already applied for an emergency refund. If not, then it generates the signature and publishes $\texttt{BTC}_\texttt{t}$ to claim y_B.

(iv) Emergency Refund Operation - If **A** has refunded x_A back to her account, **B** extracts the secret r_A from the signature σ^{refund}_{xmr} of $\texttt{XMR}_\texttt{c}$ published in ledger $\mathcal{L}_{\texttt{XMR}}$. He uses this secret to create signature $\sigma^{emergency\_refund}_{btc,\texttt{pk}_B}$ for $\texttt{BTC}_\texttt{e}$ and publishes it to refund the y_B to his account.

(v) Refund Operation - In order to cancel the swap, **A** publishes transaction $\texttt{XMR}_\texttt{c}$ in $\mathcal{L}_{\texttt{XMR}}$. **B** waits till time t_1 after $\texttt{BTC}_\texttt{l}$ has been published and claims y_B coins by publishing $\texttt{BTC}_\texttt{c}$ in $\mathcal{L}_{\texttt{BTC}}$.

6 SECURITY ANALYSIS

6.1 Security Model

We model the security of LightSwaps in the global universal composability or GUC framework [12], an extension of UC model using global setup. By global setup we mean that any resource which is considered as global is assumed to be accessed parallely by several other protocol instances. Our protocol in the real world involves set of two parties A and B which executes the protocol in presence of an adversary $\mathcal{A}$. We assume a static corruption model. Before the execution of protocol, $\mathcal{A}$ can corrupt either of the two parties and gets access to its internal states. Later, in the protocol execution, $\mathcal{A}$ controls the input and outputs of the corrupted parties. An environment, denoted as $\mathcal{Z}$, interacts with the honest parties as well as the adversary. It observes output send out by honest parties as well as the leakage by $\mathcal{A}$.

Ideal Functionality for Transaction Creation. The ideal functionality for transaction creation, $\mathcal{F}_{\texttt{tx}}$, is shown in Figure 6. The functions defined in the interface are as follows:

- `(CreateTxBTC,id,`α`,`$\texttt{pk}_s$`,`$\texttt{pk}_r$`,`t`)`: A function caller from session id sends a transaction creation request to be mined in the bitcoin ledger. The transaction after being mined must transfer α from sender address $\texttt{pk}_s$ to recipient address $\texttt{pk}_r$. The additional condition is that the transaction becomes valid only if $current\_time > t$. $\mathcal{F}_{\texttt{tx}}$ forms a valid bitcoin transaction tx taking into consideration all the factors and returns tx to the caller.
- `(CreateTxXMR,id,`α`,`$\texttt{pk}_s$`,`$\texttt{pk}_r$`)`: A function caller from session id sends a transaction creation request to be mined in the monero ledger. The transaction after being mined must

Initialization:

- Two instances of global ledger functionality $\mathcal{L}$:
  (i) $\mathcal{L}_{BTC}$ modeling the Bitcoin blockchain;
  (ii) $\mathcal{L}_{XMR}$ modeling the Monero blockchain.
- $\texttt{TxXMR}(\mathsf{pk}_s, \mathsf{pk}_r, val)$: Creates a Monero transaction whereby a sender pk_s sends an amount val to receiver pk_r.
- $\texttt{TxBTC}(\mathsf{pk}_s, \mathsf{pk}_r, val, t)$: Creates a Bitcoin transaction whereby a sender pk_s sends an amount val to receiver pk_r after elapse of time t.

Lock Operations

- **A** upon receiving input $(\texttt{lockXMR}, \texttt{id}, x, B)$:
  - Pre-signing Phase
    - **A** uses $\mathsf{vk}_{xmr,A}$ as the funding address for x XMR, generates 2-of-2 secret shared address $\mathsf{pk} = \mathcal{H}((v_A + v_B)D)G + (s_A + s_B)G$ with **B** using their shares of private spend keys s_A and s_B, and shares of private view key v_A and v_B. **A**'s share of secret key is $s'_A = \mathcal{H}((v_A + v_B)D) + s_A$ and B' share of secret key is s_B. Both **A** and **B** exchange their view keys v_A and v_B with each other.
    - **A** samples a secret r_A and generates $R_A = r_A G$ and $R^*_A = r_A H$, where G and H are two different groups. She generates $S_A = s_A G$ and $S^*_A = s_A H$.
    - **A** generates a refund address $\mathsf{vk}_{xmr,A,refund}$, creates the transaction $\texttt{XMR}_1 \leftarrow \texttt{TxXMR}(\mathsf{vk}_{xmr,A}, \mathsf{pk}, x)$ and $\texttt{XMR}_c \leftarrow \texttt{TxXMR}(\mathsf{pk}, \mathsf{vk}_{xmr,A,refund}, x)$, generates $\sigma^{lock}_{xmr} \leftarrow LSAG(\mathsf{sk}_{lock}, \texttt{XMR}_1)$.
    - **B** verifies the signature σ^{lock}_{xmr} and, generates the signature $\hat{\sigma}^{refund}_{xmr} \leftarrow \texttt{pSign2}(s'_A, s_B, \texttt{XMR}_c, R_A)$, in **A**'s collaboration, for $\texttt{XMR}_c$.
  - **A** forms $tx_{xmr,lock} = (\sigma^{lock}_{xmr}, \texttt{XMR}_1)$ and sends $(\texttt{SendTx}, id, tx_{xmr,lock})$ to ideal functionality $\mathcal{L}_{XMR}(\texttt{Valid\_Monero})$. It returns $tx_{id,xmr,lock}$ to **A**. The latter shares the transaction id $tx_{id,xmr,lock}$ with **B** and outputs $(\texttt{lockedXMR}, \texttt{id}, success)$.
- **B** receives input $(\texttt{lockBTC}, \texttt{id}, y, t_1, t_2, A)$:
  - Pre-signing Phase
    - **B** first checks whether x XMR got locked in the ledger $\mathcal{L}_{XMR}(\texttt{Valid\_Monero})$. It sends instruction $(\texttt{ReadTx}, id, tx_{id,xmr,lock})$ to the ledger.
      - Upon getting the corresponding transaction $tx_{xmr,lock}$, **B** parses the transaction, gets the lock address pk, uses the secret view key $v_A + v_B$ to check the amount monero locked in pk.
      - If x XMR is locked, **B** continues, else it aborts.
    - **A** generates pair of private and public key (a, pk_A) and **B** generates pair of private and public key (b, pk_B). Both pk_A and pk_B is used for signing Bitcoin transactions.
    - **B** shares the Bitcoin funding address tx_{fund} with **A**. Both of them generate transaction $\texttt{BTC}_1 \leftarrow \texttt{TxBTC}(tx_{fund}, \mathsf{pk}_A + \mathsf{pk}_B, y, .)$, which sends y BTC to *2-of-2* multi-sig address.
    - Transactions $\texttt{BTC}_r$, $\texttt{BTC}_c$, $\texttt{BTC}_e$ and $\texttt{BTC}_t$ are created as well, where $\texttt{BTC}_r \leftarrow \texttt{TxBTC}(\texttt{BTC}_1\texttt{\_output}, \mathsf{pk}_A + \mathsf{pk}_B, y, .)$, y BTC is send to a redeem script.
      $\texttt{BTC}_r\texttt{\_output}$ can be spend in either of the two ways:
      (i) $\texttt{BTC}_e \leftarrow \texttt{TxBTC}(\texttt{BTC}_r\texttt{\_output}, addr_{emergency\_refund}, y, .)$ spends the output of the redeem script, refunding the money to $B's$ address $addr_{emergency\_refund}$
      (ii) $\texttt{BTC}_t \leftarrow \texttt{TxBTC}(\texttt{BTC}_r\texttt{\_output}, addr_{claim}, y, t_2)$ allows **A** to claim the money after elapse of time t_2, sends it to address $addr_{claim}$.
      $\texttt{BTC}_c \leftarrow \texttt{TxBTC}(\texttt{BTC}_1\texttt{\_output}, addr_{refund}, y, t_1)$ refunds y BTC to **B** by sending it to address $addr_{refund}$ after elapse of time t_1.
    - **B** generates signature $\sigma^{take}_{btc,\mathsf{pk}_B} \leftarrow \texttt{Sign}(b, \texttt{BTC}_t)$ and sends it **A**. The latter verifies the signature, generates $\sigma^{refund}_{btc,\mathsf{pk}_A} \leftarrow \texttt{Sign}(a, \texttt{BTC}_c)$ and sends it to **B**. He generates adaptor signatures $\hat{\sigma}^{redeem}_{btc,\mathsf{pk}_B} \leftarrow \texttt{pSign}(b, \texttt{BTC}_r, S^*_A)$, $\hat{\sigma}^{emergency\_refund}_{btc,\mathsf{pk}_B} \leftarrow \texttt{pSign}(b, \texttt{BTC}_e, R^*_A)$, and sends it to **A**.
    - **A** verifies both the signature, generates $\sigma^{lock}_{btc,pk_A} \leftarrow \texttt{Sign}(a, \texttt{BTC}_1)$ and sends it to **B**. The latter verifies the signature and generates $\sigma^{lock}_{btc,pk_B} \leftarrow \texttt{Sign}(b, \texttt{BTC}_1)$.
  - **B** forms $tx_{btc,lock} = (\sigma^{lock}_{btc,pk_A}, \sigma^{lock}_{btc,pk_B}, \texttt{BTC}_1)$ and sends $(\texttt{SendTx}, id, tx_{btc,lock})$ to $\mathcal{L}_{BTC}(\texttt{Valid\_Bitcoin})$. It returns $tx_{id,btc,lock}$ to **B**. The latter in turn shares the transaction id with **A** and outputs $(\texttt{lockedBTC}, \texttt{id}, success)$.

Figure 5: Formal Protocol $\Pi_{LightSwaps}$

transfer α from sender address pk_s to recipient address pk_r. $\mathcal{F}_{\texttt{tx}}$ forms a valid monero transaction tx which hides the sender address pk_s amongst set of other address and provides a commitment of transaction amount α, only the recipient address is publicly visible. It returns tx to the caller.

Swap Initiate Operations

A upon receiving input (initiateSwapBTC, id):

- **A** uses the secret key s_A corresponding to condition S_A^* and generates a valid signature $\sigma_{btc,pk_B}^{redeem} \leftarrow \mathtt{Adapt}(\sigma_{btc,pk_B}^{\hat{redeem}}, s_A)$.
- **A** generates $\sigma_{btc,pk_A}^{redeem} \leftarrow \mathtt{Sign}(a, \mathtt{BTC_r})$, forms $tx_{btc,templock} = (\sigma_{btc,pk_A}^{redeem}, \sigma_{btc,pk_B}^{redeem}, \mathtt{BTC_r})$ and sends $(\mathtt{SendTx}, id, tx_{btc,templock})$ to $\mathcal{L}_{BTC}(\mathtt{Valid\_Bitcoin})$. It returns $tx_{id,btc,templock}$ to **A**.

Redeem Operations

A upon receiving input (redeemBTC, id):

- If the elapsed time exceeds t_2 since publishing of transaction $\mathtt{BTC_r}$ and **B** has not claimed an emergency refund, then **A** generates $\sigma_{\mathtt{pk}_A}^{take} \leftarrow \mathtt{Sign}(a, \mathtt{BTC_t})$, forms $tx_{btc,redeem} = (\sigma_{\mathtt{pk}_A}^{take}, \sigma_{\mathtt{pk}_B}^{take}, \mathtt{BTC_t})$.
- It sends $(\mathtt{SendTx}, id, tx_{btc,redeem})$ to $\mathcal{L}_{BTC}(\mathtt{Valid\_Bitcoin})$. The latter returns $tx_{id,btc,redeem}$ to **A**.

B receives input (redeemXMR, id):

- **B** sends $(\mathtt{ReadTx}, \mathtt{id}, tx_{id,btc,templock})$ to $\mathcal{L}_{BTC}(\mathtt{Valid\_Bitcoin})$. If it returns $tx_{btc,templock}$, it continues with the next step else aborts.
- **B** parses the transaction $tx_{btc,templock}$, extracts $\sigma_{btc,pk_B}^{redeem}$, gets witness $s_A \leftarrow \mathtt{Ext}(\sigma_{btc,pk_B}^{redeem}, \sigma_{btc,pk_B}^{\hat{redeem}}, S_A^*)$.
- It create the transaction $\mathtt{XMR_r} \leftarrow \mathtt{TxXMR}(\mathsf{pk}, \mathsf{vk}_{xmr,B,redeem}, x)$, where x XMR is spend from the address pk to an address of **B**, $\mathsf{vk}_{xmr,B,redeem}$. Next, it generates the signature $\sigma_{xmr}^{redeem} \leftarrow LSAG(s_A + s_B, \mathtt{XMR_r})$, forms $tx_{xmr,redeem} \leftarrow (\sigma_{xmr}^{redeem}, \mathtt{XMR_r})$ and sends $(\mathtt{SendTx}, id, tx_{xmr,redeem})$ to ideal functionality $\mathcal{L}_{XMR}(\mathtt{Valid\_Monero})$. It returns $tx_{id,xmr,redeem}$ to **B**, if the transaction is added to the ledger.

Emergency Refund Operations

B receives input (emergencyRefundBTC, id):

- If **A** has generated a refund for monero, **B** sends $(\mathtt{ReadTx}, \mathtt{id}, tx_{id,xmr,refund})$ to $\mathcal{L}_{XMR}(\mathtt{Valid\_Monero})$. If it returns a valid transaction $tx_{xmr,refund}$, **B** continues else it aborts.
- **B** parses the transaction $tx_{xmr,refund}$, extracts the signature σ_{xmr}^{refund}, gets witness $r_A \leftarrow \mathtt{Ext}(\sigma_{xmr}^{refund}, \hat{\sigma_{xmr}^{refund}}, R_A)$.
- It generates signature $\sigma_{btc,\mathtt{pk}_B}^{emergency\_refund} \leftarrow \mathtt{Adapt}(\hat{\sigma_{btc,\mathtt{pk}_B}^{emergency\_refund}}, r_A)$, forms $tx_{btc,emergency\_refund} = (\sigma_{btc,\mathtt{pk}_B}^{emergency\_refund}, \mathtt{BTC_e})$ and sends $(\mathtt{SendTx}, \mathtt{id}, tx_{btc,emergency\_refund})$ to ideal functionality $\mathcal{L}_{\mathtt{BTC}}(\mathtt{Valid\_Bitcoin})$. It returns $tx_{id,btc,emergency\_refund}$ to **B**.

Refund Operations

- **A** upon receiving (refundXMR, id):
  - **A** uses the secret r_A to create a valid signature $\sigma_{xmr}^{refund} \leftarrow \mathtt{Adapt}(\hat{\sigma}_{xmr}^{refund}, r_A)$.
  - **A** forms $tx_{xmr,refund} = (\sigma_{xmr}^{refund}, \mathtt{XMR_c})$ and sends $(\mathtt{SendTx}, id, tx_{xmr,refund})$ to ideal functionality $\mathcal{L}_{XMR}(\mathtt{Valid\_Monero})$. It returns $tx_{id,xmr,refund}$ to **A**.
- **B** upon receiving (refundBTC, id):
  - **B** checks it the elapsed time is greater than t_1 since publishing of $\mathtt{BTC}_1$ and **A** has not initiated the swap.
  - It generates $\sigma_{btc,pk_B}^{refund} \leftarrow \mathtt{Sign}(b, \mathtt{BTC_c})$, forms $tx_{btc,refund} = (\sigma_{btc,pk_A}^{refund}, \sigma_{btc,pk_B}^{refund}, \mathtt{BTC_c})$ and sends $(\mathtt{SendTx}, id, tx_{btc,refund})$ to ideal functionality $\mathcal{L}_{BTC}(\mathtt{Valid\_Bitocin})$.
  - If the transaction is valid, the ledger returns $tx_{id,btc,refund}$ to **B**.

Figure 5: Formal Protocol $\Pi_{LightSwaps}$ (Continued)

Communication Model. It is assumed that the communication between parties happen in a synchronized fashion, with protocol execution taking place in rounds. All honest parties are assumed to follow an ideal global clock $\mathcal{G}_{clock}$ which keep tracks of the time of each round. Offline communication between honest parties is assumed to occur via ideal functionality $\mathcal{F}_{smt}$, which ensures secure message transmission. Message send by party P to Q at round t reaches party Q at $t + 1$. An adversary gets to know when the message is being sent out but doesn't get to know the content of the message. Messages exchanged between parties and environment $\mathcal{Z}$ or parties and the adversary is assumed to take 0 rounds for transmission.

Ideal Functionality for Atomic Swap. We define the ideal functionality for atomic swap, $\mathcal{F}_{\mathtt{atomic\_swap}}$, as shown in Figure 7, where party A wants to exchange x_A coins for y_B coins of party B. $\mathcal{F}_{\mathtt{atomic\_swap}}$ interacts with the ideal global clock $\mathcal{G}_{clock}$, which returns the current time and keeps track of each round of the protocol. It also interacts with two instances of ideal global ledger, one for bitcoin, termed as $\mathcal{L}_{\mathtt{BTC}}$, and one for monero, termed as $\mathcal{L}_{\mathtt{XMR}}$. Each instance of ledger is parameterized with the instance of the predicate Valid

Functionality $\mathcal{F}_{tx}$

Operations:

- $(\texttt{CreateTxBTC}, \texttt{id}, \alpha, \texttt{pk}_s, \texttt{pk}_r, t)$ On input a transaction amount α, a sender's public key $\texttt{pk}_s$, a receiver's public key $\texttt{pk}_r$ and a timeperiod t, return a Bitcoin transaction tx where $\texttt{pk}_s$ sends α coins to $\texttt{pk}_r$.
- $(\texttt{CreateTxXMR}, \texttt{id}, \alpha, \texttt{pk}_s, \texttt{pk}_r)$ On input a transaction amount α, a sender's public key $\texttt{pk}_s$ and a receiver's public key $\texttt{pk}_r$, return a Monero transaction tx where $\texttt{pk}_s$ sends α to $\texttt{pk}_r$.

Figure 6: Auxiliary functionality $\mathcal{F}_{tx}$

which checks the correctness of transaction as per the conditions defined for the ledger of that particular cryptocurrency. Here, we have two instances - `Valid_Bitcoin` and `Valid_Monero`.

- `Valid_Bitcoin` checks whether a particular Bitcoin transaction is valid or not by checking for double spends and whether the transaction is has been broadcasted after the elapse of the elapse of relative locktime mentioned in the transaction.
- `Valid_Monero` checks whether a particular Monero transaction is valid or not by checking for duplicate key image for double spend, correctness of range proofs and commitments of the transaction amount.

$\mathcal{F}_{\texttt{atomic\_swap}}$ interacts with auxiliary ideal functionality $\mathcal{F}_{tx}$ which enables creation of transactions. The former has local variable t_{refund} and $t_{emergency\_refund}$. Each variable keeps track of the relative timeout period within which bitcoins must be redeemed or in case, monero gets refunded, bitcoins can be refunded as well. The other local variables maintained are *stateBTC* and *stateXMR* which keeps track of the status of bitcoin and monero respectively. If the status is *locked*, it denotes that funds have been frozen in the ledger, if the status is *redeemed*, that means the counterparty has claimed the amount after swap. If the status is $refunded$, the party which had frozen its money has withdrawn it from the ledger. *stateBTC* additionally maintains two other states: *redeemEnabled* and *emergencyRefunded*. The first one denotes that party A has initiated the process of redeeming bitcoins but cannot spend it immediately. The latter denotes that party B has withdrawn the bitcoin since A has withdrawn the monero thereby canceling the swap. $\mathcal{F}_{\texttt{atomic\_swap}}$ is initialized with two parameters $\texttt{pk}_{xmr,fund}$ and $\texttt{pk}_{btc,fund}$. These signify monero unspent address and bitcoin unspent address respectively. It is assumed that unspent bitcoins present in $\texttt{pk}_{btc,fund}$ is greater than y_B coins and unspent monero present in $\texttt{pk}_{xmr,fund}$ is greater than x_A. The ideal functionality supports the following operations:

(i) **Lock Operations**: Party A sends an instruction for locking x_A coins, ideal functionality $\mathcal{F}_{\texttt{atomic\_swap}}$ uses the funding address $\texttt{pk}_{xmr,fund}$ and creates a monero transaction, which transfers x_A coins from the funding address to another address $\texttt{pk}_{xmr,lock}$, by sending a request to $\mathcal{F}_{tx}$. Once the transaction is returned by $\mathcal{F}_{tx}$, it is send to the ideal adversary *Sim*. The latter sends the transaction to the ledger $\mathcal{L}_{\texttt{XMR}}$ which is parameterized by the predicate `Valid_Monero`. If the transaction is valid, it returns a transaction id to *Sim*, which is forwarded to the ideal functionality. The variable *statusXMR* is set to *locked*.
After the required amount of monero is locked, party B sends an instruction for locking y_B coins along with two relative locktime t_1 and t_2. $\mathcal{F}_{\texttt{atomic\_swap}}$ leaks the transaction id for locking monero, $tx_{id,xmr,lock}$, to *Sim*. The latter verifies whether the monero has been locked by checking the existence of the transaction in ledger $\mathcal{L}_{\texttt{XMR}}$, using the id $tx_{id,xmr,lock}$. Once a confirmation is send to $\mathcal{F}_{\texttt{atomic\_swap}}$, it uses the funding address $\texttt{pk}_{btc,fund}$ and creates a bitcoin transaction, which transfers y_B coins from the funding address to another address $\texttt{pk}_{btc,lock}$, by sending a request to $\mathcal{F}_{tx}$. As was done for locking phase of monero, the transaction returned by $\mathcal{F}_{tx}$ is send to the ideal adversary *Sim*. The latter sends the transaction to the ledger $\mathcal{L}_{\texttt{BTC}}$ which is parameterized by the predicate `Valid_Bitcoin`. If the transaction is valid, it returns a transaction id to *Sim*, which is forwarded to the ideal functionality. t_1 is assigned to t_{refund} and t_2 is assigned to $t_{emergency\_refund}$. The variable *statusBTC* is set to *locked*.

(ii) **Swap Initiate Operations**: If party A wants to initiate the swap, it will sends an instruction for initiating swap to $\mathcal{F}_{\texttt{atomic\_swap}}$. However, the latter creates a transaction that sends y_B coins from $\texttt{pk}_{btc,lock}$ to another address $\texttt{pk}_{btc,templock}$. The transaction is mined by sending it to ledger $\mathcal{L}_{\texttt{BTC}}$. Once locked, the money can be spend only if A makes such a request after elapse of relative locktime $t_{emergency\_refund}$ or by B, if in the meantime A has refunded x_A coins. The variable *stateBTC* is set to *redeemEnabled*.

(iii) **Redeem Operations**: If party A wants to claim y_B coins, it sends the required instruction to $\mathcal{F}_{\texttt{atomic\_swap}}$. The ideal functionality checks whether the current time is more than $t_{emergency\_refund}$ and the status of *stateBTC* is *redeemEnabled*. If both the criteria holds, then a transaction is created that sends y_B coins from $\texttt{pk}_{btc,templock}$ to an address $\texttt{pk}_{btc,redeem}$, enabling A to spend it. Once the transaction gets validated by $\mathcal{L}_{\texttt{BTC}}$, the variable *stateBTC* is set to *redeemed* which signals claiming of bitcoin by party A.
Party B can claim x_A coins by sending the required instruction to $\mathcal{F}_{\texttt{atomic\_swap}}$. The ideal functionality checks whether the current time is more than $t_{emergency\_refund}$ and the status of *stateXMR* is not $refunded$. At the same time, it must be ensured that *stateBTC* must not be *locked* or $refunded$. Claiming monero must be enabled only if party A had initiated the swap by enabling redeeming of bitcoins. If all the criteria holds, then a transaction is created which sends x_A coins from $\texttt{pk}_{xmr,lock}$ to an address $\texttt{pk}_{xmr,redeem}$, enabling B to spend it. Once the transaction gets validated by $\mathcal{L}_{\texttt{XMR}}$, the variable *stateXMR* is set to *redeemed* which signals claiming of monero by party B.

(iv) **Emergency Refund Operations**: If party A has initiated the swap but refunded the monero as well, then B must refund the bitcoin as well preventing party A from claiming both bitcoin and monero. It sends an emergency refund instruction to $\mathcal{F}_{\texttt{atomic\_swap}}$. The ideal functionality checks whether the status of *stateBTC* = *redeemEnabled* and *stateXMR* = $refunded$. If both the criteria holds, then a bitcoin transaction is created which sends y_B coins locked in $\texttt{pk}_{btc,templock}$ to a refund address of

Functionality $\mathcal{F}_{\texttt{atomic\_swap}}$

Setup: The ideal functionality $\mathcal{F}_{\texttt{atomic\_swap}}$ interacts with two parties, A and B, and the ideal adversary *Sim*. The ideal functionality for global clock, $\mathcal{G}_{clock}$, returns the current time. $\mathcal{F}_{\texttt{atomic\_swap}}$ has access to the auxiliary ideal functionality $\mathcal{F}_{tx}$. Finally, $\mathcal{F}_{\texttt{atomic\_swap}}$ also interacts with two instances of $\mathcal{L}$: (i) $\mathcal{L}_{BTC}$ modeling the Bitcoin blockchain; and (ii) $\mathcal{L}_{XMR}$ modeling the Monero blockchain.

Parameters: $\mathsf{pk}_{xmr,fund}$: Funding address for Monero.
$\mathsf{pk}_{btc,fund}$: Funding address for Bitcoin.

Local Variables: t_{refund}: A variable denoting when the refund of Bitcoin can happen
$t_{emergencyRefund}$: A variable denoting when the emergency refund of Bitcoin can happen
stateBTC: A variable denoting the state of the swap in BTC.
stateXMR: A variable denoting the state of the swap in XMR.

Lock Operations:

- Upon receiving $(\texttt{lockXMR}, \texttt{id}, x_A, B)$ from A:
  - Generate a lock address $\mathsf{pk}_{xmr,lock}$ and send $(\texttt{CreateTxXMR}, \texttt{id}, x_A, \mathsf{pk}_{xmr,fund}, \mathsf{pk}_{xmr,lock})$ to $\mathcal{F}_{tx}$ and receive transaction $tx_{xmr,lock}$.
  - Leak ($tx_{xmr,lock}$,XMR) to *Sim*. The latter sends $(\texttt{SendTx}, \texttt{id}, \texttt{tx}_{\texttt{xmr,lock}})$ to $\mathcal{L}_{XMR}(\texttt{Valid\_Monero})$
  - If $\mathcal{L}_{XMR}$ responds with $(\texttt{Confirmed}, \texttt{id}, \texttt{tx}_{\texttt{id,xmr,lock}})$, then *Sim* sends $tx_{id,xmr,lock}$ to $\mathcal{F}_{\texttt{atomic\_swap}}$.
    - $\mathcal{F}_{\texttt{atomic\_swap}}$ sets $stateXMR = locked$ and outputs $(\texttt{lockedXMR}, \texttt{id}, success)$ to A and B.
  - If $\mathcal{L}_{XMR}$ sends $\perp$ to *Sim*, $\mathcal{F}_{\texttt{atomic\_swap}}$ outputs $(\texttt{lockedXMR}, \texttt{id}, failed)$ to A and B, then abort.
- Upon receiving $(\texttt{lockBTC}, \texttt{id}, y_B, t_1, t_2, A)$ from B:
  - $\mathcal{F}_{\texttt{atomic\_swap}}$ sends transaction id for locking monero, $tx_{id,xmr,lock}$ to *Sim*. The latter sends $(\texttt{ReadTx}, \texttt{id}, \texttt{tx}_{\texttt{id,xmr,lock}})$ to $\mathcal{L}_{XMR}(\texttt{Valid\_Monero})$. If it responds with $tx_{xmr,lock}$ then proceed else sends $\perp$ to $\mathcal{F}_{\texttt{atomic\_swap}}$. In that case, the latter outputs $(\texttt{lockedBTC}, \texttt{id}, failed)$ to A and B.
  - Generate $\mathsf{pk}_{btc,lock}$ as the address for locking bitcoin. and send $(\texttt{CreateTxBTC}, \texttt{id}, y_B, \mathsf{pk}_B, \mathsf{pk}_{btc,lock}, \phi)$ to $\mathcal{F}_{tx}$ and receive transaction $tx_{btc,lock}$. Leak $(tx_{btc,lock}, BTC, t_1, t_2)$ to *Sim*.
  - If *Sim* gets a valid respond, send $(\texttt{SendTx}, \texttt{id}, \texttt{tx}_{\texttt{btc,lock}})$ to $\mathcal{L}_{BTC}(\texttt{Valid\_Bitcoin})$.
  - If $\mathcal{L}_{BTC}$ responds with $(\texttt{Confirmed}, \texttt{id}, \texttt{tx}_{\texttt{id,btc,lock}})$, then *Sim* sends $tx_{id,btc,lock}$ to $\mathcal{F}_{\texttt{atomic\_swap}}$.
    - $\mathcal{F}_{\texttt{atomic\_swap}}$ stores $t_{refund} = t_1$ and $t_{emergency\_refund} = t_2$, set $stateBTC = locked$. Output $(\texttt{lockedBTC}, \texttt{id}, success)$ to A and B.
  - If $\mathcal{L}_{BTC}$ sends $\perp$ to *Sim* then $\mathcal{F}_{\texttt{atomic\_swap}}$ outputs $(\texttt{lockedBTC}, \texttt{id}, failed)$ to A and B.

Swap Initiate Operations:

- Upon receiving $(\texttt{initiateSwapBTC}, \texttt{id})$ from A:
  - If $stateBTC = refunded$ then return $(\texttt{initiatedSwapBTC}, \texttt{id}, failed)$ to A.
  - Generate $\mathsf{pk}_{btc,templock}$ as an address for temporary locking bitcoin. and send $(\texttt{CreateTxBTC}, \texttt{id}, y_B, \mathsf{pk}_{btc,lock}, \mathsf{pk}_{btc,templock}, \phi)$ to $\mathcal{F}_{tx}$ and receive transaction $tx_{btc,templock}$. Leak ($tx_{btc,templock}$,BTC) to *Sim*. The latter sends $(\texttt{SendTx}, \texttt{id}, \texttt{tx}_{\texttt{btc,templock}})$ to $\mathcal{L}_{BTC}(\texttt{Valid\_Bitcoin})$.
  - If $\mathcal{L}_{BTC}$ responds with $(\texttt{Confirmed}, \texttt{id}, \texttt{tx}_{\texttt{id,btc,templock}})$ then *Sim* sends $tx_{id,btc,templock}$ to $\mathcal{F}_{\texttt{atomic\_swap}}$. The latter sets $stateBTC = redeemEnabled$. Output $(\texttt{initiatedSwapBTC}, \texttt{id}, success)$ to A.
  - If $\mathcal{L}_{BTC}$ sends $\perp$ to *Sim* then $\mathcal{F}_{\texttt{atomic\_swap}}$ outputs $(\texttt{initiatedSwapBTC}, \texttt{id}, failed)$ to A.

Redeem Operations:

- Upon receiving $(\texttt{redeemXMR}, \texttt{id})$ from B :
  - If $stateXMR = refunded$ and ($stateBTC = refunded$ or $stateBTC = locked$), then return $(\texttt{claimedXMR}, \texttt{id}, failed)$ to B.
  - $\mathcal{F}_{\texttt{atomic\_swap}}$ sends transaction id for redeeming Bitcoin, $tx_{id,btc,templock}$ to *Sim*. The latter sends $(\texttt{ReadTx}, \texttt{id}, \texttt{tx}_{\texttt{id,btc,templock}})$ to $\mathcal{L}_{BTC}(\texttt{Valid\_Bitcoin})$. If it responds with $tx_{btc,templock}$ then proceed else sends $\perp$ to $\mathcal{F}_{\texttt{atomic\_swap}}$. In that case, the latter outputs $(\texttt{claimedXMR}, \texttt{id}, failed)$ to B.
  - If $stateXMR = refunded$ and ($stateBTC = refunded$ or $stateBTC = locked$), then return $(\texttt{claimedXMR}, \texttt{id}, failed)$ to B.
  - $\mathcal{F}_{\texttt{atomic\_swap}}$ sends transaction id for redeeming Bitcoin, $tx_{id,btc,templock}$ to *Sim*. The latter sends $(\texttt{ReadTx}, \texttt{id}, \texttt{tx}_{\texttt{id,btc,templock}})$ to $\mathcal{L}_{BTC}(\texttt{Valid\_Bitcoin})$. If it responds with $tx_{btc,templock}$ then proceed else sends $\perp$ to $\mathcal{F}_{\texttt{atomic\_swap}}$. In that case, the latter outputs $(\texttt{claimedXMR}, \texttt{id}, failed)$ to B.

Figure 7: Interface of Ideal Functionality $\mathcal{F}_{\texttt{atomic\_swap}}$

- – Generate $\mathsf{pk}_{xmr,redeem}$ as the refund address for monero. and send $(\mathtt{CreateTxXMR}, \mathtt{id}, x_A, \mathsf{pk}_{xmr,lock}, \mathsf{pk}_{xmr,redeem})$ to $\mathcal{F}_{tx}$ and receive transaction $tx_{B,redeem}$. It leaks ($tx_{xmr,redeem}$,XMR) to *Sim*. The latter sends $(\mathtt{SendTx}, \mathtt{id}, \mathtt{tx}_{\mathtt{xmr,redeem}})$ to $\mathcal{L}_{XMR}(\mathtt{Valid\_Monero})$
  - If $\mathcal{L}_{XMR}$ responds with $(\mathtt{Confirmed}, \mathtt{id}, \mathtt{tx}_{\mathtt{id,xmr,redeem}})$ then *Sim* sends $tx_{id,xmr,redeem}$ to $\mathcal{F}_{\mathtt{atomic\_swap}}$. The latter set $stateXMR = redeemed$, sends $(\mathtt{claimedXMR}, \mathtt{id}, success)$ to A.
  - If *Sim* receives $\perp$ then $\mathcal{F}_{\mathtt{atomic\_swap}}$ outputs $(\mathtt{claimedXMR}, \mathtt{id}, failed)$ to A.
- Upon receiving $(\mathtt{redeemBTC}, \mathtt{id})$ from B
  - If $current\_time < t_{emergency\_refund}$ or $stateBTC \neq redeemEnabled$ then return $(\mathtt{claimedBTC}, \mathtt{id}, failed)$ to A.
  - Generate $\mathsf{pk}_{btc,claim}$ as the address for claiming bitcoins after timeout and send $(\mathtt{CreateTxBTC}, \mathtt{id}, y_B, \mathsf{pk}_{btc,redeem}, \mathsf{pk}_{btc,claim}, t_{emergency\_refund})$ to $\mathcal{F}_{tx}$ and receive transaction $tx_{btc,redeem}$. Leak ($tx_{btc,redeem}$,BTC) to *Sim*. The latter sends $(\mathtt{SendTx}, \mathtt{id}, \mathtt{tx}_{\mathtt{btc,redeem}})$ to $\mathcal{L}_{BTC}(\mathtt{Valid\_Bitcoin})$
  - If $\mathcal{L}_{BTC}$ responds with $(\mathtt{Confirmed}, \mathtt{id}, \mathtt{tx}_{\mathtt{id,btc,redeem}})$ then *Sim* sends $tx_{id,btc,redeem}$ to $\mathcal{F}_{\mathtt{atomic\_swap}}$. The latter sends $(\mathtt{claimedBTC}, \mathtt{id}, success)$ to A and sets $stateBTC = redeemed$.
  - If $\mathcal{F}_{\mathtt{atomic\_swap}}$ receives $\perp$ from *Sim* then output $(\mathtt{claimedBTC}, \mathtt{id}, failed)$ to A.

Emergency Refund Operations:

- Upon receiving $(\mathtt{emergencyRefundBTC}, \mathtt{id})$ from B:
  - If $stateBTC \neq redeemEnabled$ and $stateXMR \neq refunded$, then return $(\mathtt{emergencyRefundedBTC}, \mathtt{id}, failed)$ to B.
  - $\mathcal{F}_{\mathtt{atomic\_swap}}$ sends transaction id for refunding Monero, $tx_{id,xmr,refund}$ to *Sim*. The latter sends $(\mathtt{ReadTx}, \mathtt{id}, \mathtt{tx}_{\mathtt{id,xmr,refund}})$ to $\mathcal{L}_{XMR}(\mathtt{Valid\_Monero})$. If it responds with $tx_{xmr,refund}$ then proceed else sends $\perp$ to $\mathcal{F}_{\mathtt{atomic\_swap}}$. In that case, the latter outputs $(\mathtt{emergencyRefundedBTC}, \mathtt{id}, failed)$ to A.
  - Generate $\mathsf{pk}_{btc,emergency\_refund}$ as the emergency refund address for bitcoins.
  - Send $(\mathtt{CreateTx}, \mathtt{id}, x_A, \mathsf{pk}_{btc,redeem}, \mathsf{pk}_{btc,emergency\_refund}, \phi)$ to $\mathcal{F}_{tx}$ and receive transaction $tx_{btc,emergency\_refund}$. Leak ($tx_{btc,emergency\_refund}$,BTC) to *Sim*. The latter sends $(\mathtt{SendTx}, \mathtt{id}, \mathtt{tx}_{\mathtt{btc,emergency\_refund}})$ to $\mathcal{L}_{BTC}(\mathtt{Valid\_Bitcoin})$.
  - If $\mathcal{L}_{BTC}$ responds with $(\mathtt{Confirmed}, \mathtt{id}, \mathtt{tx}_{\mathtt{id,btc,emergency\_refund}})$, then *Sim* sends $tx_{btc,emergency\_refund}$ to $\mathcal{F}_{\mathtt{atomic\_swap}}$. The latter sets $stateBTC = emergencyRefunded$, sends $(\mathtt{emergencyRefundedBTC}, \mathtt{id}, success)$ to B.
  - If $\mathcal{F}_{\mathtt{atomic\_swap}}$ receives $\perp$ from *Sim* then output $(\mathtt{emergencyRefundedBTC}, \mathtt{id}, failed)$ to B.
  - Generate $\mathsf{pk}_{btc,emergency\_refund}$ as the emergency refund address for bitcoins.
  - Send $(\mathtt{CreateTx}, \mathtt{id}, x_A, \mathsf{pk}_{btc,redeem}, \mathsf{pk}_{btc,emergency\_refund}, \phi)$ to $\mathcal{F}_{tx}$ and receive transaction $tx_{btc,emergency\_refund}$. Leak ($tx_{btc,emergency\_refund}$,BTC) to *Sim*. The latter sends $(\mathtt{SendTx}, \mathtt{id}, \mathtt{tx}_{\mathtt{btc,emergency\_refund}})$ to $\mathcal{L}_{BTC}(\mathtt{Valid\_Bitcoin})$.
  - If $\mathcal{L}_{BTC}$ responds with $(\mathtt{Confirmed}, \mathtt{id}, \mathtt{tx}_{\mathtt{id,btc,emergency\_refund}})$, then *Sim* sends $tx_{btc,emergency\_refund}$ to $\mathcal{F}_{\mathtt{atomic\_swap}}$. The latter sets $stateBTC = emergencyRefunded$, sends $(\mathtt{emergencyRefundedBTC}, \mathtt{id}, success)$ to B.
  - If $\mathcal{F}_{\mathtt{atomic\_swap}}$ receives $\perp$ from *Sim* then output $(\mathtt{emergencyRefundedBTC}, \mathtt{id}, failed)$ to B.

Refund Operations:

- Upon receiving $(\mathtt{refundXMR}, \mathtt{id})$ from A :
  - If $stateXMR = redeemed$, then return $(\mathtt{refundedXMR}, \mathtt{id}, failed)$ to A.
  - Generate $\mathsf{pk}_{xmr,refund}$ as the refund address for monero.
  - Send $(\mathtt{CreateTxXMR}, \mathtt{id}, x_A, \mathsf{pk}_{xmr,lock}, \mathsf{pk}_{xmr,refund})$ to $\mathcal{F}_{tx}$ and receive transaction $tx_{xmr,refund}$. Leak ($tx_{xmr,refund}$,XMR) to *Sim*. The latter sends $(\mathtt{SendTx}, \mathtt{id}, \mathtt{tx}_{\mathtt{xmr,refund}})$ to $\mathcal{L}_{XMR}(\mathtt{Valid\_Monero})$
  - If $\mathcal{L}_{XMR}$ responds with $(\mathtt{Confirmed}, \mathtt{id}, \mathtt{tx}_{\mathtt{id,xmr,refund}})$, then *Sim* sends $tx_{id,xmr,refund}$ to $\mathcal{F}_{\mathtt{atomic\_swap}}$. The latter send $(\mathtt{refundedXMR}, \mathtt{id}, success)$ to A and sets $stateXMR = refunded$.
  - If $\mathcal{F}_{\mathtt{atomic\_swap}}$ receives $\perp$ from *Sim*, then output $(\mathtt{refundedXMR}, \mathtt{id}, failed)$ to A.
- Upon receiving $(\mathtt{refundBTC}, \mathtt{id})$ from B:
  - If $stateBTC \neq locked$ or $current\_time < t_{refund}$ then return $(\mathtt{refundedBTC}, \mathtt{id}, failed)$ to B.
  - Generate $\mathsf{pk}_{btc,refund}$ as the refund address for bitcoins.
  - Send $(\mathtt{CreateTxBTC}, \mathtt{id}, y_B, \mathsf{pk}_{btc,lock}, \mathsf{pk}_{btc,refund}, t_{refund})$ to $\mathcal{F}_{tx}$ and receive transaction $tx_{btc,refund}$. It leaks ($tx_{btc,refund}$,BTC) to *Sim*. The latter sends $(\mathtt{SendTx}, \mathtt{id}, \mathtt{tx}_{\mathtt{btc,refund}})$ to $\mathcal{L}_{BTC}(\mathtt{Valid\_Bitcoin})$.
  - If $\mathcal{L}_{BTC}$ responds with $(\mathtt{Confirmed}, \mathtt{id}, \mathtt{tx}_{\mathtt{id,btc,refund}})$ then *Sim* sends $tx_{id,btc,refund}$ to $\mathcal{F}_{\mathtt{atomic\_swap}}$. The latter sets $stateBTC = refunded$ and outputs $(\mathtt{refundedBTC}, \mathtt{id}, success)$ to B.
  - If *Sim* receives $\perp$ then $\mathcal{F}_{\mathtt{atomic\_swap}}$ outputs $(\mathtt{refundedBTC}, \mathtt{id}, failed)$ to B.

Figure 7: Interface of Ideal Functionality $\mathcal{F}_{\mathtt{atomic\_swap}}$ (Continued)

B. The transaction is send to $\mathcal{L}_{\mathtt{BTC}}$ for validation. Once it gets validated, set $stateBTC = emergencyRefunded$.

(v) **Refund Operations**: If party A wants to refund x_A coins, it requests for refund to $\mathcal{F}_{\mathtt{atomic\_swap}}$. The ideal functionality checks whether the status of $stateXMR$ is $locked$. If the criteria holds,

then a monero transaction is created which sends x_A coins from $\mathsf{pk}_{xmr,lock}$ to an address $\mathsf{pk}_{xmr,refund}$. Once the transaction gets validated by $\mathcal{L}_{\mathtt{XMR}}$, the variable $stateXMR$ is set to $refunded$ which signals refund of x_A coins to A.
Party B can refund its y_B coins by requesting $\mathcal{F}_{\mathtt{atomic\_swap}}$ for a refund. The ideal functionality checks whether the current time is more than t_{refund} and the status of $stateBTC = locked$. If both the criterias holds, then a bitcoin transaction is created which sends y_B coins from $\mathsf{pk}_{btc,lock}$ to a refund address $\mathsf{pk}_{btc,refund}$. If the transaction gets validated by $\mathcal{L}_{\mathtt{BTC}}$, the latter returns t the variable $stateXMR$ is set to $redeemed$ which signals claiming of monero by party B.

6.2 Security Discussions

We discuss how the ideal functionality $\mathcal{F}_{\mathtt{atomic\_swap}}$ guarantees the security properties discussed in Section 5.1.

- **Correctness**: If both parties A and B are honest, then the atomic swap protocol ensures that if party A is able to redeem y_B coins then party B can redeem x_A coins as well within a bounded timeperiod.
- **Soundness**: If party A initiates the swap but refunds monero before party B can claim it, then a relative locktime of t_2, between initiating swap and claiming of bitcoins, allows party B to opt for an emergency refund and refund its bitcoin.
- **Linkability**: If all the parties are honest, and the amount in the Monero transaction remains hidden, any malicious party observing the ledger $\mathcal{L}$ cannot link between a pair of bitcoin and monero transaction involved in the swap.
- **Fungibility**: There is no structural difference between a normal Monero transaction and a Monero transaction involved in the swap. Any malicious party observing the ledger $\mathcal{L}$ can distinguish between such transactions with negligible probability.

We show that any attack that can be performed on $\Pi_{LightSwaps}$ can also be simulated on $\mathcal{F}_{atomic\_swap}$, or in other words that $\Pi_{LightSwaps}$ is at least as secure as $\mathcal{F}_{atomic\_swap}$. To prove this, we design a simulator *Sim*, that acts like an ideal attacker for the ideal functionality. We show that no PPT environment can distinguish between interacting with the real world and interacting with the ideal world. In the real world, the environment $\mathcal{Z}$ sends instructions to a real attacker $\mathcal{A}$ and interacts with $\Pi_{LightSwaps}$. In the ideal world, $\mathcal{Z}$ sends attack instructions to *Sim* and interacts with $\mathcal{F}_{atomic\_swap}$. We provide the design of simulator *Sim* in Figure 9 - when A is dishonest and B is honest, and in Figure 10 - when A is honest and B is dishonest. We prove formally that the protocol Π GUC-realizes the protocol $\mathcal{F}_{\mathtt{atomic\_swap}}$ in the $(\mathcal{G}_{clock}, \mathcal{F}_{smt}, \mathcal{L})$-hybrid world.

7 DISCUSSION

7.1 Building Monero transactions

Pre-signing transactions involve signing a transaction where the outputs that need to be spent as input in this transaction have not been added to the blockchain. Since the private spend key and private view key for spending the output of $\mathtt{XMR}_1$ is generated using 2-of-2 secret sharing, it requires both parties to co-operate and generate a valid signature for spending this output. However, if Bob stops responding, Alice will never get back her coins. Pre-signing $\mathtt{XMR}_\mathsf{c}$ will allow her to go for refund anytime she wants prior to signing of $\mathtt{XMR}_1$ [28]. Unfortunately, it is not possible to implement the pre-signing of Monero transaction in its present form. We specify the key components for building a Monero transaction - (i) a transaction has a ring signature per input to hide exactly which output is being spent, (ii) a unique key image for an input being spent to avoid double-spending, (iii) Pedersen commitments [35] for every input and output, retaining the confidentiality of the transaction, and lastly, (iv) to show that input and output balance out and the output is non-negative, bulletproofs [10] are used for the output.

The input of a Monero transaction, denoted as `vin`, consists of the amount, key offsets, and key image. Since the amount is confidential, it is set to 0. The key offset allows verifiers to find ring member keys and commitments in the blockchain. It consists of the real output public key along with 10 other decoy outputs. The first offset value is the absolute height of the block where the first member is present. Rest are assigned values relative to the absolute value. For example, if the set of 11 public keys forming ring members have real offsets $\{h, h+4, h+6, h+10, h+20, h+33, h+45, h+50, h+67, h+77, h+98\}$, then it is recorded as $\{h, 4, 2, 4, 10, 13, 12, 5, 17, 10, 21\}$ where h is the height of the block where the first public key can be found and each subsequent offset is relative to the previous. This set is termed as *"ring"* and is stored in the transaction. To ensure that a particular output can only be used once as an input, Monero includes a key image of the output's public key. The key image is constructed using the public key of the output that will be spent. This avoids double-spending attacks in Monero blockchain. Next, we discuss how the input *"ring"* is used for constructing the ring signature CLSAG.

For computing the signature hash $c_{i+1}, \forall i \in \{0, 1, \ldots, 10\}$ where $c_{11} = c_0$, *"ring"* is taken as input along with other parameters and concatenated with L_i and R_i. To generate the signature, the offsets must be known. Offsets are not known until and unless all the outputs in set *ring* have been added to the blockchain. Lack of offsets violates the policy of pre-signing where the transaction must be signed before the output that needs to be spent gets added to the blockchain. To avoid this problem, instead of using the key offsets as input for generating a signature hash, the set of public keys can be used as input. However, this would require changing Monero's codebase but the change is necessary for realizing Layer 2 protocols in Monero blockchain.

7.2 Building Bitcoin transactions

We created the necessary Bitcoin transactions for LightSwap and deployed these transactions on the Bitcoin testnet. We observed and recorded the size of transactions in bytes, where $\mathtt{BTC}_1$ and $\mathtt{BTC}_\mathsf{r}$ is 360 B each, $\mathtt{BTC}_\mathsf{c}$ is 230 B, $\mathtt{BTC}_\mathsf{e}$ is 231 B, and $\mathtt{BTC}_\mathsf{t}$ is 229 B. Our result demonstrates the compatibility of the protocol with the current Bitcoin network. The code is available in https://anonymous.4open.science/r/btc_xmr_swap-A7B1, forked from https://github.com/generalized-channels/gc.

8 RELATED WORK

There have been efforts to design time locks on Monero. DLSAG [32] mentions that Monero is locked in a 2-of-2 joint address comprising

two different public keys. Any one of the public keys can be used to spend Monero from the address based on certain conditions, for example, pre-defined block height. However, Monero needs to undergo a hard fork to implement DLSAG. Thyagarajan et al. [42] proposed the first payment channel for Monero, PayMo, without requiring any system-wide modifications. Additionally, the authors have also proposed a secure atomic cross-chain swap using PayMo. The payment channel uses a new cryptographic primitive called *Verifiable Timed Linkable Ring Signature (VTLRS)*. The signature scheme uses the timed commitment of a linkable ring signature on a given Monero transaction. However, timed commitment requires a huge computation overhead, making it unsuitable for designing lightweight protocols.

Threshold ring multi-signature proposed by Goodell and Noether [18] was used for spender-ambiguous cross-chain atomic swaps. Their construction doesn't involve any timelock mechanism, it is based on sharing of secret keys - whenever one party goes on-chain for claiming the amount, the other party can reconstruct the secret key completely. However, the paper doesn't formally define the refund method in case one of the parties acts maliciously.

Gugger proposed a protocol which tries to solve the problem [19]. **A** locks its monero in a 2-of-2 secret shared address, where a part of the private spend key remains with **B**. On the other hand, **B** locks Bitcoin in a multisig address having two outputs, one is redeem and one is for refunding. The redeem script uses hash lock where the preimage of the hash must be used for claiming Bitcoins. First, **B** locks Bitcoin and upon confirmation, **A** locks its Monero. When the former confirms that the required amount of XMR has been locked, it sends the preimage of the hash defined in the redeem script. Using it, **B** publishes the redeem transaction and releases his part of private spend key to **A**. The latter uses it to construct the private spend key and claim XMR. Note that in our case, we require **A** to lock XMR before **B** does. If we change the order then **A** is at a risk of losing its deposit forever if **B** refuses to collaborate i.e. *unhappy path (4)*. There is no way **A** can go for refund at time t_5. Even *unhappy path (2)* is not realizable using this protocol, since **A** can claim BTC but cannot initiate a refund since it is dependent on **B** for that. The schematic diagram of the protocol is shown in Figure 8. Hoenisch and Pino [22] provide a high-level sketch of a protocol that mitigates the limitations of Gugger's protocol. However, it avoids any detailed description of the construction of the adaptor ring signature on Monero.

To address these problems, we propose a protocol which allows **A** to refund instead of depending on **B**. With this guarantee, **A** can always make the first move by locking XMR before **B** locks BTC.

9 CONCLUSIONS

We propose LightSwap, a lightweight two-party atomic swap facilitating the exchange of Bitcoin and Monero. LightSwap does not require any type of timeout at one of the two blockchains, without additional trust assumptions. Our protocol is thus efficient, fungible, scalable, and can be used for any cryptocurrency whose script does not support timelock. Either the party can initiate a refund, even if the counterparty does not cooperate. We provide steps for implementing LightSwap that demonstrate the ability to seamlessly deploy the protocol if Monero's codebase is changed to enable Layer 2 protocols. In the future, we are interested to study if a protocol can be designed without using timelock even at the Bitcoin side and what additional trust assumptions would be needed.

REFERENCES

[1] 2013. *TierNolan.* Technical Report. https://github.com/TierNolan.

[2] 2019. *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society. https://www.ndss-symposium.org/ndss2019/

[3] anonymous. 2022. LightSwap: An Atomic Swap Does Not Require Timeouts At Both Blockchains (Full version). https://anonymous.4open.science/r/Lightswap-B982/Final-LongversionXMR_lock_then_BTC.pdf.

[4] Team Ark. 2019. ARK Ecosystem Whitepaper. https://ark.io/Whitepaper.pdf.

[5] Lukas Aumayr, Oguzhan Ersoy, Andreas Erwig, Sebastian Faust, Kristina Hostakova, Matteo Maffei, Pedro Moreno-Sanchez, and Siavash Riahi. 2020. Generalized Bitcoin-Compatible Channels. *IACR Cryptol. ePrint Arch.* 2020 (2020), 476.

[6] Lukas Aumayr, Pedro Moreno-Sanchez, Aniket Kate, and Matteo Maffei. 2021. Blitz: Secure Multi-Hop Payments Without Two-Phase Commits. In *USENIX Security 21*.

[7] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. 2017. Bitcoin as a transaction ledger: A composable treatment. In *Annual international cryptology conference*. Springer, 324–356.

[8] Iddo Bentov, Yan Ji, Fan Zhang, Lorenz Breidenbach, Philip Daian, and Ari Juels. 2019. Tesseract: Real-Time Cryptocurrency Exchange Using Trusted Hardware. In *CCS '19, London, UK, November 11-15, 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM, 1521–1538. https://doi.org/10.1145/3319535.3363221

[9] Michael Borkowski, Marten Sigwart, Philipp Frauenthaler, Taneli Hukkinen, and Stefan Schulte. 2019. DeXTT: Deterministic cross-blockchain token transfers. *IEEE Access* 7 (2019), 111030–111042.

[10] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. 2018. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE symposium on security and privacy (SP)*. IEEE, 315–334.

[11] Ran Canetti. 2001. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. IEEE, 136–145.

[12] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. 2007. Universally composable security with global setup. In *Theory of Cryptography Conference*. Springer, 61–85.

[13] Peter Chvojka, Tibor Jager, Daniel Slamanig, and Christoph Striecks. 2021. Versatile and sustainable timed-release encryption and sequential time-lock puzzles. In *ESORICS '21*. Springer, 64–85.

[14] Amazon Elastic Compute Cloud. 2011. Amazon web services. *Retrieved November* 9, 2011 (2011), 2011.

[15] Bingrong Dai, Shengming Jiang, Menglu Zhu, Ming Lu, Dunwei Li, and Chao Li. 2020. Research and implementation of cross-chain transaction model based on improved hash-locking. In *International Conference on Blockchain and Trustworthy Systems*. Springer, 218–230.

[16] Apoorvaa Deshpande and Maurice Herlihy. 2020. Privacy-preserving cross-chain atomic swaps. In *FC '20*. Springer, 540–549.

[17] Oded Goldreich. 2007. *Foundations of cryptography: volume 1, basic tools*. Cambridge university press.

[18] Brandon Goodell and Sarang Noether. 2018. Thring Signatures and their Applications to Spender-Ambiguous Digital Currencies. *IACR Cryptol. ePrint Arch.* 2018 (2018), 774.

[19] Joël Gugger. 2020. Bitcoin-Monero Cross-chain Atomic Swap. Cryptology ePrint Archive, Report 2020/1126. https://eprint.iacr.org/2020/1126.

[20] Runchao Han, Haoyu Lin, and Jiangshan Yu. 2019. On the optionality and fairness of Atomic Swaps. In *ACM AFT '19*. 62–75.

[21] Maurice Herlihy. 2018. Atomic Cross-Chain Swaps. In *PODC '18, Egham, United Kingdom, July 23-27, 2018*, Calvin Newport and Idit Keidar (Eds.). ACM, 245–254. https://doi.org/10.1145/3212734

[22] Philipp Hoenisch and Lucas Soriano del Pino. 2021. Atomic Swaps between Bitcoin and Monero. *CoRR* abs/2101.12332 (2021). arXiv:2101.12332 https://arxiv.org/abs/2101.12332

[23] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. [n.d.]. Zcash protocol specification. ([n. d.]).

[24] Aggelos Kiayias and Dionysis Zindros. 2019. Proof-of-work sidechains. In *FC '19*. Springer, 21–34.

[25] Komodo. 2018. Komodo (Advanced Blockchain Technology, Focused On Freedom). https://cryptorating.eu/whitepapers/Komodo/2018-02-14-Komodo-White-Paper-Full.pdf.

[26] Jae Kwon and Ethan Buchman. 2019. Cosmos whitepaper. *A Netw. Distrib. Ledgers* (2019).

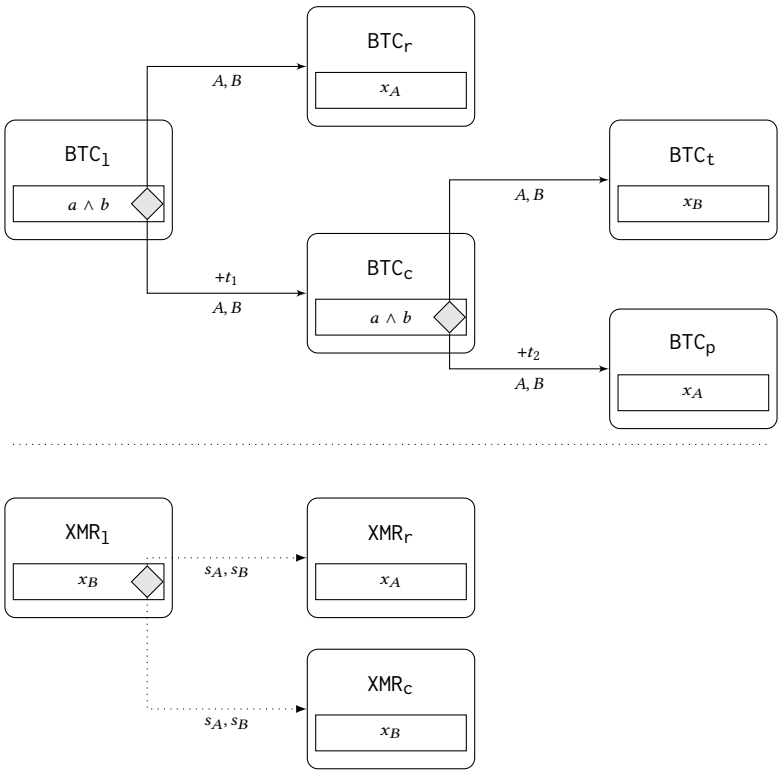

Figure 8: Transaction schema for BTC to XMR atomic swaps from Gugger et al [19]. *Top*: Transaction schema for Bitcoin. *Bottom*: Transaction schema for Monero. *Note: Monero view keys are omitted for clarity.*

[27] Rongjian Lan, Ganesha Upadhyaya, Stephen Tse, and Mahdi Zamani. 2021. Horizon: A Gas-Efficient, Trustless Bridge for Cross-Chain Transactions. *arXiv preprint arXiv:2101.06000* (2021).

[28] Lucas. 2021. How to build a Monero transaction. https://comit.network/blog/2021/05/19/monero-transaction/.

[29] Léonard Lys, Arthur Micoulet, and Maria Potop-Butucaru. 2021. *R-SWAP: Relay based atomic cross-chain swap protocol.* Ph.D. Dissertation. Sorbonne Université.

[30] Giulio Malavolta, Pedro Moreno-Sanchez, Clara Schneidewind, Aniket Kate, and Matteo Maffei. 2019. Anonymous Multi-Hop Locks for Blockchain Scalability and Interoperability, See [2]. https://www.ndss-symposium.org/ndss-paper/anonymous-multi-hop-locks-for-blockchain-scalability-and-interoperability/

[31] Mahdi H Miraz and David C Donald. 2019. Atomic cross-chain swaps: development, trajectory and potential of non-monetary digital token swap facilities. *Annals of Emerging Technologies in Computing (AETiC) Vol* 3 (2019).

[32] Pedro Moreno-Sanchez, Arthur Blue, Duc Viet Le, Sarang Noether, Brandon Goodell, and Aniket Kate. 2020. DLSAG: Non-interactive Refund Transactions for Interoperable Payment Channels in Monero. In *FC '20, Kota Kinabalu, Malaysia, February 10-14, 2020 Revised Selected Papers (Lecture Notes in Computer Science, Vol. 12059)*, Joseph Bonneau and Nadia Heninger (Eds.). Springer, 325–345. https://doi.org/10.1007/978-3-030-51280-4_18

[33] Krishnasuri Narayanam, Venkatraman Ramakrishna, Dhinakaran Vinayagamurthy, and Sandeep Nishad. 2022. Generalized HTLC for Cross-Chain Swapping of Multiple Assets with Co-Ownerships. *arXiv preprint arXiv:2202.12855* (2022).

[34] Shen Noether. 2015. Ring Signature Confidential Transactions for Monero. Cryptology ePrint Archive, Report 2015/1098. https://eprint.iacr.org/2015/1098.

[35] Torben Pryds Pedersen. 1991. Non-interactive and information-theoretic secure verifiable secret sharing. In *Annual international cryptology conference*. Springer, 129–140.

[36] Ronald L. Rivest, Adi Shamir, and David A. Wagner. 1996. *Time-lock puzzles and timed-release crypto.* Technical Report.

[37] Drew Stone. 2021. Trustless, privacy-preserving blockchain bridges. *arXiv preprint arXiv:2102.04660* (2021).

[38] Erkan Tairi, Pedro Moreno-Sanchez, and Matteo Maffei. 2019. A^{2}L: Anonymous Atomic Locks for Scalability and Interoperability in Payment Channel Hubs. *IACR Cryptol. ePrint Arch.* 2019 (2019), 589. https://eprint.iacr.org/2019/589

[39] Stefan Thomas and Evan Schwartz. 2015. A protocol for interledger payments. *URL https://interledger. org/interledger. pdf* (2015).

[40] Sri AravindaKrishnan Thyagarajan, Giulio Malavolta, and Pedro Moreno-Sánchez. 2021. Universal Atomic Swaps: Secure Exchange of Coins Across All Blockchains. *Cryptology ePrint Archive* (2021).

[41] Sri Aravinda Krishnan Thyagarajan, Adithya Bhat, Giulio Malavolta, Nico Döttling, Aniket Kate, and Dominique Schröder. 2020. Verifiable Timed Signatures Made Practical. In *CCS '20, USA, November 9-13, 2020*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM, 1733–1750. https://doi.org/10.1145/3372297.3417263

[42] Sri Aravinda Krishnan Thyagarajan, Giulio Malavolta, Fritz Schmidt, and Dominique Schröder. 2020. PayMo: Payment Channels For Monero. *IACR Cryptol. ePrint Arch.* 2020 (2020), 1441. https://eprint.iacr.org/2020/1441

[43] Hangyu Tian, Kaiping Xue, Xinyi Luo, Shaohua Li, Jie Xu, Jianqing Liu, Jun Zhao, and David SL Wei. 2021. Enabling cross-chain transactions: A decentralized cryptocurrency exchange protocol. *IEEE Transactions on Information Forensics and Security* 16 (2021), 3928–3941.

[44] Nicolas Van Saberhagen. 2013. CryptoNote v 2.0.

[45] Gilbert Verdian, Paolo Tasca, Colin Paterson, and Gaetano Mondelli. 2018. Quant Overledger Whitepaper. https://uploads-ssl.webflow.com/6006946fee85fda61f666256/60211c93f1cc59419c779c42_Quant_Overledger_Whitepaper_Sep_2019.pdf.

[46] Gang Wang. [n.d.]. SoK: Exploring Blockchains Interoperability. ([n. d.]).

[47] Gavin Wood. 2016. Polkadot: Vision for a heterogeneous multi-chain framework. *White Paper* 21 (2016), 2327–4662.

[48] Victor Zakhary, Divyakant Agrawal, and Amr El Abbadi. 2019. Atomic commitment across blockchains. *arXiv preprint arXiv:1905.02847* (2019).

[49] Alexei Zamyatin, Dominik Harz, Joshua Lind, Panayiotis Panayiotou, Arthur Gervais, and William J. Knottenbelt. 2019. XCLAIM: Trustless, Interoperable, Cryptocurrency-Backed Assets. In *IEEE S & P '19, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 193–210. https://doi.org/10.1109/SP.2019.00085

Lock Operations

- For locking monero:
  - *Sim* gets funding address $\mathsf{vk}_{xmr,A}$ from A^*. It samples pairs of private key and secret key $(\mathsf{sk}_A, \mathsf{pk}_A)$ and $(\mathsf{sk}_B, \mathsf{pk}_B)$ to sign bitcoin transactions on behalf of A and B respectively.
  - It creates a locking address for monero denoted as vk^*, refund address vk_A.
  - It creates $\mathtt{XMR}_1 \leftarrow \mathit{SendXMR}(\mathsf{vk}_{xmr,A}, \mathsf{vk}^*)$ and signs the transaction $\mathtt{XMR}_1$. The signature on $\mathtt{XMR}_1$ is σ^{lock}_{xmr}. It also creates transaction $\mathtt{XMR}_\mathsf{c} \leftarrow \mathit{SendXMR}(\mathsf{vk}^*, \mathsf{vk}_A)$.
  - It creates $tx_{xmr,lock} = (\sigma^{lock}_{xmr}, \mathtt{XMR}_1)$ and sends $(\mathtt{SendTx}, id, tx_{xmr,lock})$ to $\mathcal{L}_{\mathtt{XMR}}(\mathtt{Valid\_XMR})$. If it returns $tx_{id,xmr,lock}$ then *Sim* sends $(\mathtt{lockXMR}, \mathtt{id}, tx_{id,xmr,lock})$ to $\mathcal{F}_{\mathtt{atomic\_swap}}$.
- For locking bitcoins:
  - If *Sim* gets $tx_{id,xmr,lock}$ from $\mathcal{F}_{atomic\_swap}$, it sends $(\mathtt{ReadTx}, \mathtt{id}, tx_{id,xmr,lock})$ to $\mathcal{L}_{\mathtt{XMR}}(\mathtt{Valid\_Monero})$. If the latter responds with $tx_{xmr,refund}$, then *Sim* forwards it to the ideal functionality.
  - If the $tx_{xmr,refund}$ is correct, $\mathcal{F}_{atomic\_swap}$ leaks $(tx_{btc,lock}, t_1, t_2)$ to *Sim*. It parses $tx_{B,lock}$ to get $\mathsf{pk}_{btc,funding}$, $\mathsf{pk}_{btc,lock}$ and the amount y_B.
  - *Sim* sends $(\mathtt{SendTx}, id, tx_{btc,lock})$ to $\mathcal{L}_{\mathtt{BTC}}(\mathtt{Valid\_BTC})$. If it returns $tx_{id,btc,lock}$ then *Sim* sends $tx_{id,btc,lock}$ to $\mathcal{F}_{\mathtt{atomic\_swap}}$.

Swap Initiate Operations

- *Sim* internally executing the protocol between A^* and B.
- If A^* knows the secret s_A for condition S^*_A and wants to initiate swap:
  - *Sim* checks if $\mathit{current\_time} < t_{refund}$, generates a 2-of-2 multisig address $\mathsf{pk}_{btc,templock}$, creates transaction $\mathtt{BTC}_\mathsf{r}$, sending y_B coins from $\mathsf{pk}_{btc,lock}$ to $\mathsf{pk}_{btc,templock}$, uses s_A and creates the signature $\sigma^{redeem}_{btc,\mathsf{pk}_B}$.
  - It forms $tx_{btc,templock} = (\sigma^{redeem}_{btc,\mathsf{pk}_A}, \sigma^{redeem}_{btc,\mathsf{pk}_B}, \mathtt{BTC}_\mathsf{r})$ and sends $(\mathtt{SendTx}, id, tx_{btc,templock})$ to $\mathcal{L}_{\mathtt{BTC}}(\mathtt{Valid\_BTC})$. If it returns $tx_{id,btc,templock}$ then send $(\mathtt{initiateSwapBTC}, \mathtt{id}, tx_{id,btc,templock})$ to $\mathcal{F}_{atomic\_swap}$.
- Else *Sim* goes idle.

Redeem Operations

- For redeeming bitcoins:
  - If A^* wants to redeem y_B coins by publishing $\mathtt{BTC}_\mathsf{t}$, *Sim* does the following:
    - *Sim* checks if $\mathit{current\_time} \geq t_{emergency\_refund}$, generates a claim address $\mathsf{pk}_{btc,redeem}$, creates transaction $\mathtt{BTC}_\mathsf{t}$, sending y_B from $\mathsf{pk}_{btc,redeem}$ to $\mathsf{pk}_{btc,redeem}$.
    - It signs the transaction on behalf of both A^* and B, forms $tx_{btc,redeem} = (\sigma^{take}_{btc,\mathsf{pk}_A}, \sigma^{take}_{btc,\mathsf{pk}_B}, \mathtt{BTC}_\mathsf{t})$ and sends $(\mathtt{SendTx}, id, tx_{btc,redeem})$ to $\mathcal{L}_{\mathtt{BTC}}(\mathtt{Valid\_BTC})$. If it returns $tx_{id,btc,redeem}$ then send $(\mathtt{claimBTC}, \mathtt{id}, tx_{id,btc,redeem})$ to $\mathcal{F}_{atomic\_swap}$.
  - Else *Sim* goes idle.
- For redeeming monero:
  - If *Sim* gets $tx_{id,btc,templock}$ from $\mathcal{F}_{atomic\_swap}$, it sends $(\mathtt{ReadTx}, \mathtt{id}, tx_{id,btc,templock})$ to $\mathcal{L}_{\mathtt{BTC}}(\mathtt{Valid\_Bitcoin})$. If the latter returns a valid transaction $tx_{btc,templock}$, it forwards it to $\mathcal{F}_{atomic\_swap}$.
  - $\mathcal{F}_{atomic\_swap}$ leaks $tx_{xmr,redeem}$, *Sim* parses the transaction to get $\mathsf{pk}_{xmr,redeem}$.
  - It sends $(\mathtt{SendTx}, id, tx_{xmr,redeem})$ to $\mathcal{L}_{\mathtt{XMR}}(\mathtt{Valid\_XMR})$. If it returns $tx_{id,xmr,redeem}$ then send $tx_{id,xmr,redeem}$ to $\mathcal{F}_{atomic\_swap}$.

EmergencyRefund Operations

- If *Sim* gets $tx_{id,xmr,refund}$ from $\mathcal{F}_{atomic\_swap}$, it sends $(\mathtt{ReadTx}, \mathtt{id}, tx_{id,xmr,refund})$ to $\mathcal{L}_{\mathtt{XMR}}(\mathtt{Valid\_Monero})$. If the latter returns a valid monero transaction $tx_{xmr,refund}$, it forwards it to $\mathcal{F}_{atomic\_swap}$.
- $\mathcal{F}_{atomic\_swap}$ leaks $tx_{btc,emergency\_refund}$ to *Sim*, it parses the transaction to get $\mathsf{pk}_{btc,emergency\_refund}$.
- It sends $(\mathtt{SendTx}, id, tx_{btc,emergency\_refund})$ to $\mathcal{L}_{\mathtt{BTC}}(\mathtt{Valid\_Bitcoin})$. If it returns $tx_{id,btc,emergency\_refund}$ then send $tx_{id,btc,emergency\_refund}$ to $\mathcal{F}_{atomic\_swap}$.

Refund Operations

- For refunding monero:
  - If A^* has the secret r_A, generates the signature for $\mathtt{XMR}_\mathsf{c}$ and publishes the transaction, then *Sim* does the following:
    - *Sim* generates $tx_{xmr,refund} = \{\sigma^{refund}_{xmr}, \mathtt{XMR}_\mathsf{c}\}$,
    - It sends $(\mathtt{SendTx}, id, tx_{xmr,refund})$ to $\mathcal{L}_{\mathtt{XMR}}(\mathtt{Valid\_Monero})$. If it returns $tx_{id,xmr,refund}$ then *Sim* sends $(\mathtt{refundXMR}, \mathtt{id}, tx_{id,xmr,refund})$ to $\mathcal{F}_{\mathtt{atomic\_swap}}$.
  - Else *Sim* goes idle.
- For refunding bitcoin:
  - If *Sim* gets $tx_{btc,refund}$ from $\mathcal{F}_{\mathtt{atomic\_swap}}$, it parses the transaction to get $\mathsf{pk}_{btc,refund}$.
  - It sends $(\mathtt{SendTx}, id, tx_{btc,refund})$ to $\mathcal{L}_{\mathtt{BTC}}(\mathtt{Valid\_Bitcoin})$. If it returns $tx_{id,btc,refund}$ then send $tx_{id,btc,refund}$ to $\mathcal{F}_{atomic\_swap}$.

Figure 9: Simulator design when A^* is dishonest and B is honest

Lock Operations

- For locking monero:
  - *Sim* receives $tx_{xmr,lock}$ from $\mathcal{F}_{atomic\_swap}$. It will parse $tx_{xmr,lock}$ to get the lock address vk.
  - It sends $(\mathtt{SendTx}, id, tx_{xmr,lock})$ to $\mathcal{L}_{\mathtt{XMR}}(\mathtt{Valid\_Monero})$. If it returns $tx_{id,xmr,lock}$ then send $tx_{id,xmr,lock}$ to $\mathcal{F}_{atomic\_swap}$.
- For locking bitcoins:
  - *Sim* gets $(tx_{funding}, y, t_1, t_2)$ and funding address $tx_{funding}$ for y_B coins from B^*. If no message is received, *Sim* goes idle.
  - It sends $(\mathtt{ReadTx}, \mathtt{id}, tx_{id,xmr,lock})$ to $\mathcal{L}_{\mathtt{XMR}}(\mathtt{Valid\_Monero})$. If $\mathcal{L}_{\mathtt{XMR}}$ replies with $\perp$, *Sim* goes idle, else it continues with the next step.
  - It samples pairs of private and public key to sign bitcoin transactions on behalf of parties A and B, denoted as $(\mathsf{sk}_A, \mathsf{pk}_A)$ and $(\mathsf{sk}_B, \mathsf{pk}_B)$.
  - *Sim* generates the transaction $\mathtt{BTC}_1$ by sending y_B coins from $tx_{funding}$ to a multisig address denoted as $\mathsf{pk}_{btc,lock}$.
  - It signs the transaction $\mathtt{BTC}_1$ on behalf of both A and B, generating $\sigma^{lock}_{btc,\mathsf{pk}_A}$ and $\sigma^{lock}_{btc,\mathsf{pk}_B}$.
  - It creates $tx_{btc,lock} = (\sigma^{lock}_{btc,\mathsf{pk}_A}, \sigma^{lock}_{btc,\mathsf{pk}_B}, \mathtt{BTC}_1)$ and $(\mathtt{SendTx}, id, tx_{btc,lock})$ to $\mathcal{L}_{\mathtt{BTC}}(\mathtt{Valid\_Bitcoin})$. If it returns $tx_{id,btc,lock}$ then *Sim* sends $(\mathtt{lockBTC}, \mathtt{id}, tx_{id,btc,lock})$ to $\mathcal{F}_{atomic\_swap}$.

Swap Initiate Operations

- If *Sim* gets $tx_{btc,templock}$ from $\mathcal{F}_{atomic\_swap}$, it parses the transaction to get the redeem address $\mathsf{pk}_{btc,templock}$.
- It sends $(\mathtt{SendTx}, id, tx_{btc,templock})$ to $\mathcal{L}_{\mathtt{BTC}}(\mathtt{Valid\_BTC})$. If it returns $tx_{id,btc,templock}$ then send $tx_{id,btc,templock}$) to $\mathcal{F}_{atomic\_swap}$.

Redeem Operations

- For redeeming bitcoins:
  - *Sim* receives $tx_{btc,redeem}$ from $\mathcal{F}_{atomic\_swap}$. It will parse $tx_{btc,redeem}$ to get the redeem address $\mathsf{pk}_{btc,redeem}$.
  - It sends $(\mathtt{SendTx}, id, tx_{btc,rdeem})$ to $\mathcal{L}_{\mathtt{BTC}}(\mathtt{Valid\_Bitcoin})$. If it returns $tx_{id,btc,redeem}$ then send $tx_{id,btc,redeem}$ to $\mathcal{F}_{atomic\_swap}$.
- For redeeming monero:
  - If B^* has the secret s_A and publishes the transaction $\mathtt{XMR}_\mathsf{r}$, *Sim* checks if A has initiated the swap by querying the ledger $\mathcal{L}_{\mathtt{BTC}}$. It sends $(\mathtt{ReadTx}, \mathtt{id}, tx_{id,btc,templock})$ to the ledger. If it responds with a valid transaction, then *Sim* continues with the next step, else it remains idle.
  - It creates the transaction $\mathtt{XMR}_\mathsf{r}$, sending x_A coins from $\mathsf{pk}_{xmr,lock}$ to $\mathsf{pk}_{xmr,redeem}$.
  - It generates the signature $\sigma^{redeem}_{xmr} \leftarrow LSAG(s_A + s_B, \mathtt{XMR}_\mathsf{r})$.
  - It creates $tx_{xmr,redeem} = (\sigma^{redeem}_{xmr}, \mathtt{XMR}_\mathsf{r})$ and $(\mathtt{SendTx}, id, tx_{xmr,redeem})$ to $\mathcal{L}_{\mathtt{XMR}}(\mathtt{Valid\_Monero})$. If it returns $tx_{id,xmr,redeem}$ then *Sim* sends $(\mathtt{claimedXMR}, \mathtt{id}, tx_{id,xmr,redeem})$ to $\mathcal{F}_{atomic\_swap}$.

EmergencyRefund Operations

- If B^* has the secret r_A and publishes the transaction $\mathtt{BTC}_\mathsf{e}$, *Sim* checks if A has refunded monero by querying the ledger $\mathcal{L}_{\mathtt{XMR}}$. It sends $(\mathtt{ReadTx}, \mathtt{id}, tx_{id,xmr,refund})$ to the ledger. If it responds with a valid transaction, then *Sim* continues with the next step, else it remains idle.
- It creates the transaction $\mathtt{BTC}_\mathsf{e}$, sending y_B coins from $\mathsf{pk}_{btc,redeem}$ to $\mathsf{pk}_{btc,emergency\_refund}$.
- It generates the signature for the transaction $\mathtt{BTC}_\mathsf{e}$ using witness for statement R^*_A, denoted as $\sigma^{emergency\_refund}_{btc,\mathsf{pk}_B}$.
- It creates $tx_{btc,emergency\_refund} = (\sigma^{emergency\_refund}_{btc,\mathsf{pk}_A}, \sigma^{emergency\_refund}_{btc,\mathsf{pk}_B}, \mathtt{BTC}_\mathsf{e})$ and $(\mathtt{SendTx}, id, tx_{btc,emergency\_refund})$ to $\mathcal{L}_{\mathtt{BTC}}(\mathtt{Valid\_Bitcoin})$. If it returns $tx_{id,btc,emergency\_refund}$ then *Sim* sends $(\mathtt{emergencyRefundBTC}, \mathtt{id}, tx_{id,btc,emergency\_refund})$ to $\mathcal{F}_{atomic\_swap}$.

Refund Operations

- For refunding monero:
  - If *Sim* gets $tx_{xmr,refund}$ from $\mathcal{F}_{atomic\_swap}$, it parses the transaction to get the refund address $\mathsf{pk}_{xmr,refund}$.
  - It sends $(\mathtt{SendTx}, id, tx_{xmr,refund})$ to $\mathcal{L}_{\mathtt{XMR}}(\mathtt{Valid\_XMR})$. If it returns $tx_{id,xmr,refund}$ then send $tx_{id,xmr,refund}$ to $\mathcal{F}_{atomic\_swap}$.
- For refunding bitcoin:
  - If B^* wants to refund y_B coins by publishing $\mathtt{BTC}_\mathsf{c}$, *Sim* does the following:
    - *Sim* checks if $current\_time \geq t_{refund}$, generates a claim address $\mathsf{pk}_{btc,refund}$, creates transaction $\mathtt{BTC}_\mathsf{c}$, sending y_B coins from $\mathsf{pk}_{btc,lock}$ to $\mathsf{pk}_{btc,refund}$.
    - It signs the transaction on behalf of both A^* and B, forms $tx_{btc,refund} = (\sigma^{refund}_{btc,\mathsf{pk}_A}, \sigma^{refund}_{btc,\mathsf{pk}_B}, \mathtt{BTC}_\mathsf{c})$ and sends $(\mathtt{SendTx}, id, tx_{btc,refund})$ to $\mathcal{L}_{\mathtt{BTC}}(\mathtt{Valid\_BTC})$. If it returns $tx_{id,btc,refund}$ then send $(\mathtt{refundBTC}, \mathtt{id}, tx_{id,btc,refund})$ to $\mathcal{F}_{atomic\_swap}$.
  - Else *Sim* goes idle.

Figure 10: Simulator design when A is honest and B^* is dishonest