Warning! Timeout $T$ Cannot Protect You From Losing Coins

pipeSwap: Forcing the Early Release of a Secret for Atomic Swaps Across All Blockchains

Peifang Ni, Anqi Tian, Jing Xu

ABSTRACT

Atomic cross-chain swap, which allows users to exchange coins securely, is critical functionality to facilitate inter-currency exchange and trading. Although most classic atomic swap protocols based on Hash Timelock Contracts have been applied and deployed in practice, they are substantially far from universality due to the inherent dependence of rich scripting language supported by the underlying blockchains. The recently proposed Universal Atomic Swaps protocol [IEEE S&P’22] takes a novel path to scriptless cross-chain swap, and it ingeniously delegates scripting functionality to cryptographic lock mechanisms, particularly the adaptor signature and timed commitment schemes designed to guarantee atomicity. However, in this work, we discover a new form of attack called double-claiming attack, such that the honest user would lose coins with overwhelming probability and atomicity is directly broken. Moreover, this attack is easy to carry out and can be naturally generalized to other cross-chain swap protocols as well as the payment channel networks, highlighting a general difficulty in designing universal atomic swap.

We present pipeSwap, a cross-chain swap protocol that satisfies both security and practical universality. To avoid transactions of the same frozen coins being double-claimed to violate the atomicity property, pipeSwap proposes a novelly designed paradigm of pipelined coins flow by using two-hop swap and two-hop refund techniques. pipeSwap achieves universality by not relying on any specific script language, aside from the basic ability to verify signatures. Furthermore, we analyze why existing ideal functionality falls short in capturing the atomicity property of Universal Atomic Swaps, and define for the first time ideal functionality to guarantee atomicity. In addition to a detailed security analysis in the Universal Composability framework, we develop a proof-of-concept implementation of pipeSwap with Schnorr/ECDSA signatures, and conduct extensive experiments to evaluate the overhead. The experimental results show that pipeSwap can be performed in less than 1.7 seconds and requires less than 7 kb of communication overhead on commodity machines, which demonstrates its high efficiency.

KEYWORDS


1 INTRODUCTION

With numerous and diverse blockchain systems coexisting today, it is impossible to envision each one evolving in isolation, especially given the explosive development of cryptocurrencies such as Bitcoin [29], Ethereum [46], and Ripple [39]. This proposes an extremely urgent demand of deploying robust currency payments across any blockchain based cryptocurrency. The atomic cross-chain swap protocol [32] is built on top of the underlying blockchains and introduced for securely exchanging coins between two distrusting users, who respectively hold coins in two distinct blockchains. The fundamental security property atomicity guarantees that the honest user cannot lose coins. Conventionally, the timeout parameter $T$, which is predefined specifically for each user, serves as the crux of describing atomicity. In slightly more detail, either the honest user obtains its desired coins before its timeout $T$, or its frozen coins are refunded after timeout $T$.

A Desideratum for Achieving Atomic Cross-Chain Swap in the Absence of Custom Scripts. Most classic efforts focus on studying the Hash Timelock Contracts (HTLC)-style solutions [10, 11, 19], which use the rich scripting languages supported by underlying blockchains to describe when and how the user can claim the counterparty’s locked coins or refund its own locked coins. Unsurprisingly, this design form is incompatible with most existing cryptocurrencies (e.g., Bitcoin [29], Monero [25], Mimblewimble [34], Ripple [39] and Zcash [38]) and far from the universal solution. In addition, it raises a privacy concern due to the use of timelock functionality, which makes the transactions easier to be distinguished from the general ones [1], especially for blockchains having already achieved privacy [38]. Indeed, when introducing a new cryptocurrency or even among the existing ones, relying on specific scripting functionality would lead to fatal obstacles for communications.

Therefore, it is not only of practically relevant, but also theoretically interesting to investigate what are the minimum scripting functionalities necessary to design secure cross-chain swaps. Noticeably, instead of relying on traditional on-chain scripting to denote locked coins and their corresponding unlock conditions, Universal Atomic Swaps [42] make the best use of adaptor signature scheme [3] and verifiable timed discrete logarithm (VTD) [43] to become the closest solution for a universal proposal. Specifically, the witness extractability of adaptor signatures facilitates a successful swap, where once the user holding witness $y$ posts a valid swap transaction, the witness $y$ is subsequently released to the other user to complete its swap operation. Additionally, VTD ensures that in the event of a failed swap, the locked coins are refunded to their original owner after a predefined timeout $T$. Universal Atomic Swaps take a novel path to scriptless cross-chain swap and thus become arguably the best candidate for implementing cryptocurrency exchange.

Is Timeout $T$ Really Secure for Honest User? Nevertheless, the timeout $T$ can potentially violate the atomicity property. Herein we present a new attack termed the double-claiming attack. It is noteworthy that in the context of Universal Atomic Swaps [42], the predefined timeout $T_i$
is intentionally designed for user $P_1$ to securely refund its frozen coins (i.e., user $P_1$ obtains the full secret key of its frozen address at timeout $T_1$), but timeout $T_1$ cannot deprive the right of user $P_0$ to post its swap transaction (i.e., the witness $y$ is still valid). As a result, the timeout $T_1$ will become a flashpoint for security issues. For instance, when user $P_1$ posts a refund transaction after timeout $T_1$, the malicious $P_0$ can still release its swap transaction to make $P_1$’s frozen coins double-claiming. Unfortunately, if user $P_0$’s swap transaction is accepted and finally confirmed by the underlying blockchain, honest user $P_1$ will neither successfully enter into Swap Complete Phase nor prevail in its Swap Timeout Phase, ultimately resulting in the loss of coins. Even worse, the double-claiming attack is extremely easy to carry out in all the network models (cf. Section 3 for detailed discussions).

The above issues bring the fundamental challenge in designing atomic cross-chain swaps: the HTLC-style proposals put severe obstacles to achieving universality, whereas Universal Atomic Swaps protocol is susceptible to the double-claiming attack mentioned above. This naturally raises a question:

“Can we design a cross-chain swap protocol to achieve the best of both security and universality?”

1.1 Our Contributions

In this work, we contribute to the rigorous understanding of atomic cross-chain swaps and answer the aforementioned question affirmatively by presenting a new protocol called pipeSwap. The contributions are outlined below:

- **Double-claiming attack.** We analyze the security of Universal Atomic Swaps [42], and discover a new form of attack called double-claiming attack, which can directly break atomicity with overwhelming probability, i.e., the protocol will end with the adversarial user both obtaining its counterparty’s frozen coins and refunding its own frozen coins. We argue that this attack is easy to carry out and naturally generalizes to other cross-chain swap protocols as well as the payment channel networks [4, 20, 26, 44], since the attack exploits the fact that the frozen coin can be claimed by both its original owner and the intended receiver after its timeout $T$.

- **The stronger atomicity.** We analyze why the existing security model falls short in capturing the security of Universal Atomic Swaps, and observe that the ideal functionality $\mathcal{F}_\text{swap}^\beta$ (Fig.3 in [42]) cannot cope with the condition that both users simultaneously initiate their respective $buy$ and $abort$ requirements. This motivates the definition of ideal functionality to guarantee stronger atomicity. Informally, such an ideal functionality states that each frozen coin can only be claimed by a swap transaction if the valid swap transaction can be generated before the timeout; otherwise, it can only be refunded.

- **pipeSwap.** Inspired by a novel paradigm of pipelined coins flow, we present a new atomic cross-chain swap protocol called pipeSwap. As depicted in Fig.1, each frozen coin is viewed as a drop of water and flows along the one-way arrows. Informally, for each frozen coin, it can only flow to its intended receiver via the green pipe if the swap completes successfully, otherwise, once the procedure enters into Timeout Phase, it definitely flows to its original owner via the red pipe. By this way, if the frozen coin has been claimed by a valid swap transaction, it will no longer continue to flow forward. Instead, after timeout $T$, the frozen coin will never rewind to its intended receiver. Thus pipeSwap satisfies the definition of stronger atomicity. Moreover, pipeSwap is universal, that is, the protocol does not require any specific script language, apart from the basic capability to verify signatures. Additionally, pipeSwap preserves fungibility, which means that an observer cannot distinguish a swap transaction from a standard one, owning to the fact that the on-chain transactions in pipeSwap are identical to standard one-to-one transactions. As a byproduct of our approach, the core idea of pipelined coins flow can be leveraged to design secure multi-hop swaps (including the multi-hop payments). The comparisons with prior approaches are shown in Table 1.

- **Implementation.** We develop a proof-of-concept implementation of pipeSwap for Schnorr and ECDSA, and conduct extensive experiments to evaluate the overhead. The results demonstrate the high efficiency and the best suitability of our design. In particular, pipeSwap has the running time of less than 1.7 seconds and the communication costs of less than 7 kb. Remarkably, even though pipeSwap provides stronger security protection against double-claiming attack, compared to Universal Atomic Swaps [42], it only takes a few more milliseconds for completing the second hop of swap/refund operation.

![Figure 1: The pipelined life-cycle of swapped coins](image)

1.2 Technique Overview

Recall that in a classic $\alpha$-to-$\beta$ swaps protocol [42], the user $P_0$ is given priority to post its swap transaction of coins $\beta$ before timeout $T_1$, simultaneously releases a witness $y$ w.r.t. hard relation $R$ to $P_1$ for completing its swap operation of coins $\alpha$. To avoid double-claiming of the same frozen coins...
to violate the *atomicity*, we resort to different techniques of forcing the earlier release of witness \( y \), i.e., releasing witness \( y \) is viewed as a prerequisite for posting swap transaction of coins \( \beta \). We elaborate on technical contributions as detailed below.

A new freezing structure better prepared for atomicity. To instantiate the pipelined coins flow, our critical step is to correctly foresee the flow direction of the frozen coin before its timeout. Different from Universal Atomic Swaps, we propose a new freezing structure in which the frozen coins \( \beta \) are stored in two distinct frozen addresses and the smaller part with value \( \varepsilon \to 0 \) is used to compete for the final flow direction. Notably, a pre-transaction (i.e., pre-swap or pre-refund transaction) is designed for spending frozen coins \( \varepsilon \), and the real spending transaction (i.e., swap or refund transaction) of coins \( \beta \) takes the corresponding pre-transaction as one of its inputs. Such a method forces the users to actively post a pre-transaction instead of waiting until the timeout.

**two-hop swap.** More importantly, the new freezing structure inspires us to novelly design a two-hop swap method of claiming frozen coins \( \beta \), while frozen coins \( \alpha \) can be directly unlocked with witness \( y \). Formally, the puzzle \( (Y, y) \in \mathbb{R} \) is inserted in the signature of pre-swap transaction of coins \( \varepsilon \) instead of the final swap transaction of frozen coins \( \beta \).

**two-hop swap** can prevent user \( P_1 \) from losing coins \( \alpha \) even if the adversary \( P_0 \) posts its swap transaction after timeout \( T_1 \), because the witness \( y \) has been released before timeout \( T_1 \).

**two-hop refund.** Obviously, it is not enough to guarantee the pipelined coins flow solely with the two-hop swap design, if the frozen coins \( \beta \) can be directly refunded by user \( P_1 \) after timeout \( T_1 \). We further propose the corresponding two-hop refund of frozen coins \( \beta \), where frozen coins \( \varepsilon \) are firstly refunded by a pre-refund transaction after time \( T_1 \) (\( T_1 - T_1 > \varphi \), where \( \varphi \) is theconfirmation latency of underlying blockchain) and then frozen coins \( \beta - \varepsilon \) are spent by the final refund transaction after timeout \( T_1 \). Notice that the design of two-hop refund can effectively force user \( P_0 \) to post its pre-swap transaction before time \( T_1 \), otherwise the malicious delay would lead to frozen coins \( \varepsilon \) being claimed by a pre-refund transaction and finally coins \( \beta \) being refunded by user \( P_1 \) after timeout \( T_1 \).

2 **BLOCKCHAIN AND CROSS-CHAIN ATOMIC SWAP**

We first recall the formal definition of Unspent Transaction Output (UTXO) model \([3]\), which is adopted by the majority of current blockchains (e.g., Bitcoin \([29]\) ), and then take a brief overview of Universal Atomic Swaps \([42]\).

2.1 **The UTXO-based Blockchain**

**Transactions.** Under the UTXO model, a transaction is a tuple of the form \((\text{input}, \text{output}, V, \Omega)\), which transfers coins from \( m \geq 1 \) inputs \( \text{input} := \{i_1, \ldots, i_m\} \) to \( l \geq 1 \) outputs \( \text{output} := \{o_1, \ldots, o_l\} \). In particular, \( V := \{v_1, \ldots, v_m\} \) denotes the value of each input and \( \Omega := \{\sigma_1, \ldots, \sigma_l\} \) is the witness of spending each input. Usually, we use public key \( pk \) to denote the input/output address, for example, \( tx := (pk_1, pk_2, v, \sigma) \) means transferring coins \( v \) in address \( pk_1 \) to address \( pk_2 \), and \( \sigma \) is the signature of \( tx \) that verifies w.r.t. \( pk_1 \) and the coins in address \( pk_2 \) can only be further spent with signature under \( pk_2 \). Additionally, the conditions of spending coins can be some scripts supported by scripting language of the underlying blockchain (i.e., TimeLock and HTLC), but in this paper we focus on the scriptless ones.

We use transaction chart to visualize the coins flow between addresses. As depicted in Fig. 2(a), the rounded rectangle represents transaction \( tx \) with the incoming arrow as \textit{input} and the blue box with value \( v \) represents the amount of coins, whose spending condition is written above the outgoing arrow.

**Blockchain.** A blockchain can be used as an append-only bulletin board \( T \) to record the posted transactions and also be viewed as a trusted ledger \( L \) to store all the unspent coins associated with each address \( pk \). In essence, a blockchain is built and maintained by the parties who compete to be elected as the next leader to propose a candidate block, which contains a sequence of transactions. It is extremely important to notice that, in a secure real-world blockchain execution, the leaders prioritize packaging the transactions received first into blocks and, for two conflicting transactions (i.e., spending the same coins) received simultaneously, they randomly select one of the two transactions as a valid transaction. What’s more, we strictly separate the *parties* who are responsible for the secure execution of the underlying blockchain from the *users* who only participate in the cross-chain swap protocol supported by the underlying blockchains. Therefore, the (honest) leaders do not care about the story behind each transaction and, in their views, all the valid transactions are treated equally. Unsurprisingly, it is reasonable that, the valid transaction \( tx' \) in a pair of conflicting transactions \( \{tx, tx'\} \) is finally confirmed even if it is maliciously generated.

![Figure 2: The transaction flow.](image-url)

(a) **Transaction** \( tx \) signed w.r.t. \( pk_1 \) transfers coins of value \( v \) from address \( pk_1 \) to \( pk_2 \), and \( tx \) can be further spent by a transaction signed w.r.t. \( pk_2 \).

(b) **Transaction** \( tx_3 \) is finally confirmed by the underlying blockchain and further spent by a valid transaction \( tx_2 \), which has been recorded in \( T \) and not confirmed yet.
valid transactions, especially for the conflicting transactions, it only selects the one that arrives at earlier or randomly selects one in the case of simultaneous arrival. Obviously, a valid transaction $tx \in T$ can be confirmed finally (i.e., $tx \in L$) if it has been in $T$ for time $\varphi$.

For presentation simplicity, we use doubled blue edge rectangle and single edge rectangle to present the confirmed transaction and valid transaction respectively (see Fig.2(b)). After a clear understanding of transaction processing mechanism of the underlying blockchain, we have to pay more attention to the following simple but practical scenario (see Fig.3). Since the one who holds secret key $sk_i$ w.r.t. public key $pk_i$ can sign any transactions at its will, thus we cannot prevent the adversarial payer $U_1$ from generating two valid transactions $tx_1, tx_2$, where $tx_2$ pays for the intended receiver $U_2$ and $tx_1$ pays back to itself (i.e., users $U_1$ and $U_2$ own addresses $pk_1, pk'_1$ and $pk_2, pk'_2$, respectively). Fortunately, if $t_2 < t_1$ then $tx_2$ can be accepted by $T$ and as a result the receiver $U_2$ obtains its deserved coins after time $\varphi$. However, if $t_1 = t_2$ then, with probability 50%, the receiver $U_2$ will lose coins as $tx_1$ being accepted by $T$. Even worse, when the network delay is under adversarial control (e.g., the $\Delta$-synchronous communication network [22]), the malicious transaction $tx_1$ would compete against $tx_2$ with a landslide.

Leveraging the above observation, we should be wary of some time points that can cause the same coins to be doubly claimed by different users. Especially for realizing atomic coins transfer/swap in a decentralized manner, only relying on a valid transaction rather than a confirmed one would lead to fatal vulnerabilities (see Section 3).

### 2.2 Cross-Chain Atomic Swap

Generally, a cross-chain swap protocol enables two distrustful users $P_0$ and $P_1$, who respectively own coins $\alpha$ and $\beta$ in two distinct blockchains $B_0$ and $B_1$, to exchange coins securely. The fundamental security property of cross-chain swap protocol is atomicity: the honest users cannot lose coins. Specifically, honest user $P_i$ definitely gets coins $\beta$ in $B_1$ if swap succeeds. Otherwise, $P_0$ enters into its Timeout Phase and successfully unlocks frozen coins $\alpha$.

Now we recall the design of Universal Atomic Swaps [42] (see Fig.4). We use hereunder notations: (1) item with subscript $\in \{frz, swp, rfd\}$ respectively refers to the freeze, swap and refund operation. It consists of four phases described as follows:

**Swap Setup Phase-Freezing coins:** Users $P_0$ and $P_1$ jointly generate the frozen addresses $pk_i^{(0)}$, where the corresponding secret keys $sk_i^{(0)} := s_i^{(0)} + sk_i^{(1)}$ and $sk_i^{(10)} := sk_i^{(10)} + sk_i^{(1)}$ are shared between them, and respectively compute the timed commitments (Def.4) $VDT_{frz} := (C^{(0)}, \pi^{(1)})$ and $VDT_{frz} := (C^{(0)}, \pi^{(0)})$ of shares $s_i^{(0)}$ and $s_i^{(1)}$ (Note that, after timeout $T_1$, user $P_i$ can get secret key $sk_i^{(1-i)}$). After the above is successful, user $P_i$ transfers coins from address $pk_i^{(0)}$ to $pk_i^{(1-i)}$ via frozen transaction $tx_i^{frz}$.

**Swap Lock Phase:** Using adaptor signature w.r.t. hard relation $(Y, y) \in R$ (Def.1) selected by user $P_0$, users $P_0$ and $P_1$ jointly generate pre-signatures $\sigma_{swp}^{(1)}$ of swap transaction $tx_i^{swp}$ in sequence. Notice that, from now on, user $P_0$ with witness $y$ can generate valid swap transaction $tx_i^{swp}$ at any time.

**Swap Complete Phase:** If user $P_0$ actively posts swap transaction $tx_i^{swp}$ before timeout $T_1$, $P_1$ can extract $y$ (i.e., the purple dotted arrow) to generate signature $\sigma_{swp}^{(1)}$ of swap transaction $tx_i^{swp}$. Thus the users successfully swap coins;

**Swap Timeout Phase:** If Swap Complete Phase fails, user $P_1$ enters into its Swap Timeout Phase after timeout $T_1$ and posts refund transaction $tx_i^{rfd}$ with secret key $sk_i^{(10)}$. Correspondingly, after timeout $T_0$, user $P_0$ posts refund transaction $tx_i^{rfd}$. Therefore, the frozen coins are respectively refunded to their original owners.

**Security Analysis.** We summarize security analysis of Universal Atomic Swaps and defer detailed proofs to [42];

- **Successful Swap:** If user $P_0$ honestly posts swap transaction $tx_i^{swp}$ before timeout $T_1$, user $P_1$ can extract $y$ to generate its swap transaction $tx_i^{swp}$ successfully;
- **Failed Swap:** If user $P_0$ fails to post swap transaction $tx_i^{swp}$ before timeout $T_1$, user $P_1$ fails in Swap Complete Phase and enters into Swap Timeout Phase to refund its frozen coins via posting transaction $tx_i^{swp}$. Similarly, user $P_0$ can refund its frozen coins after timeout $T_0$.
3 THE DOUBLE-CLAIMING ATTACK

What Double-Claiming Attack Is. In a cross-chain swap protocol, the malicious user (it is user $P_0$ or $P_1$) can deviate arbitrarily from the swap protocol. In addition, the underlying blockchain network may be under the adversarial control to delay or reorder messages, and it prioritizes transactions that arrive first (cf. Fig. 7). The double-claiming attack refers to a situation where a malicious user attempts to create a double-claimed state for the frozen coins (i.e., the frozen coins are spent simultaneously by their original owner and the intended receiver), thereby obtaining the offered coins from its counterparty while refusing to transfer its own coins. This directly violates the atomicity property of the cross-chain swap protocol.

Why There Exists This Attack. Essentially, the core reasons that lead to double-claiming attack are:

Reason 1: The balance security of transaction. Independent from the inner workings in cross-chain swap protocols, total balances of all the addresses in underlying blockchain are unchanged. Specifically, no new coins are generated causelessly and any coins can only be equivalently transferred to some new addresses. Furthermore, the scriptless nature of cross-chain swap determines that the verification of a transaction relies only on the balance and signature. Thus, in the view of underlying blockchain, the conflicting transactions (e.g., the swap and refund transactions of the same frozen coins) are separately valid and the balance security determines that only one of these two transactions can be finally confirmed.

Reason 2: After timeout $T$, the frozen coins can be spent by both its original owner and intended receiver. We recall that the frozen coins can only be refunded after the predefined timeout $T$, while the intended receiver is able to claim these coins both before and after timeout $T$. At first glance, this seems reasonable that the timeout $T$ provides enough time for the intended receiver to swap and guarantees the coins can be refunded in the case of failure. However, there is no mechanism to cancel the intended receiver’s ability to claim frozen coins after timeout $T$ (i.e., signing its swap transaction), which is the source of double-claiming attack.

How Easy It Is to Conduct This Attack. Note that the double-claiming attack is completely different from the double-spending attack [21], where the former enables the malicious user to prevent an honest transaction from being confirmed, while the latter enables the malicious miner who is involved in the maintenance of underlying blockchain to confirm the both spending transactions of one coin. Additionally, the double-spending attack requires the attacker (i.e., the malicious miner) to hold and consume enough resources (e.g., computational power [29] or stakes [6]), while the double-claiming attack is crazy-cheap and only requires the attacker (i.e., the malicious user) to have the ability of signing transactions (i.e., holding the signing secret key). In particular, the double-claiming attack can work in all the network models of the underlying blockchains (e.g., the strong synchrony [17], $\Delta$-synchrony [39], partial synchrony [15] and asynchrony [2]). Even worse, with the weakening of the underlying blockchain network (i.e., from strong synchrony to asynchrony), the double-claiming attack can succeed with a higher probability, which can be detailed as follows.

As depicted in Fig. 5, double-claiming attack can work in Universal Atomic Swaps [42], where the underlying blockchain network is strong synchrony (i.e., there is no message propagation delay and $\Delta = 0$), in two manners.

User $P_0$ is malicious (Fig. 5(a)). First, users $P_0$ and $P_1$ initiate an $\alpha$-to-$\beta$ swap via successfully freezing their respective coins, and then the adversary $P_0$ gets off-line until timeout $T_1$. In the Swap Timeout Phase, user $P_1$ posts refund transaction $tx_{rfd}^{(1)}$ and simultaneously the adversary $P_0$ goes re-online and posts swap transaction $tx_{swp}^{(0)}$. At timeout $T_0$, the adversary $P_0$ enters into Swap Timeout Phase to post refund transaction $tx_{rfd}^{(0)}$. Due to the fact that only one of transactions $tx_{swp}^{(0)}$ and $tx_{rfd}^{(1)}$ can be finally confirmed by the corresponding underlying blockchain. Accordingly, with probability close to 1/2, the malicious transactions $tx_{swp}^{(0)}$ and $tx_{rfd}^{(0)}$ are both confirmed. As a result, the adversary $P_0$ harvests double assets and the atomicity property is broken.

User $P_1$ is malicious (Fig. 5(b)). Let us continue to recall some details of Universal Atomic Swaps. If user $P_0$ honestly posts swap transaction $tx_{swp}^{(0)}$ before timeout $T_1$, this swap protocol should be successful such that swap transactions $tx_{swp}^{(0)}$ and $tx_{swp}^{(1)}$ are finally confirmed by the underlying...
blockchain. However, if user $P_0$ honestly posts swap transaction $tx^{(0)}_{swp}$ almost near timeout $T_1$, the adversary $P_1$ can issue its swap transaction $tx^{(1)}_{swp}$ immediately and try to enter into its Swap Timeout Phase by posting refund transaction $tx^{(1)}_{rfd}$. Similarly, due to the fact that only one of transactions $tx^{(0)}_{swp}$ and $tx^{(1)}_{rfd}$ can be finally confirmed by the corresponding underlying blockchain. Accordingly, with probability close to 1/2, the malicious transactions $tx^{(0)}_{swp}$ and $tx^{(1)}_{rfd}$ are both confirmed. As a result, the malicious user $P_1$ harvests double assets and the atomicity property is broken.

Furthermore, the strong network synchrony assumption is impractical for a large-scale distributed system such as Bitcoin [37]. When considering the $\Delta$-synchronous blockchain network [33] that the network delay is under adversarial control and message is delivered within a known delay $\Delta$, we note that the double-claiming attack can work easier. Generally, the attack strategies are the same as above. For malicious user $P_0$ (Fig.5(a)), as long as it releases swap transaction $tx^{(0)}_{swp}$ within time $\Delta$ after refund transaction $tx^{(1)}_{rfd}$ being posted, then transaction $tx^{(0)}_{swp}$ can defeat $tx^{(1)}_{rfd}$ and user $P_0$ harvests double assets (i.e., by confirming transactions $tx^{(0)}_{swp}$ and $tx^{(1)}_{rfd}$) with an absolute advantage. Correspondingly, for malicious user $P_1$ (Fig.5(b)), once user $P_0$ honestly posts swap transaction $tx^{(0)}_{swp}$ between time $T_1 - \Delta$ and $T_1$, the adversary $P_1$ can post its swap transaction $tx^{(1)}_{swp}$ immediately and then posts a competitive refund transaction $tx^{(1)}_{rfd}$ after timeout $T_1$ via delaying the transaction $tx^{(0)}_{swp}$. As a result, the malicious user $P_1$ harvests double assets (i.e., by confirming transactions $tx^{(0)}_{swp}$ and $tx^{(1)}_{rfd}$) with an absolute advantage and the atomicity property is broken. Therefore, the weaker but realistic blockchain network can be exploited to improve the successful probability of double-claiming attack.

The Generality of This Attack. It should be emphasized that double-claiming attack is not specifically tailored to Universal Atomic Swaps, but generally applies to the scriptless (or timelock scripts only) cross-chain swaps and to Universal Atomic Swaps, but generally applies to the multi-hop payments [37]. Here, our attack is not specifically tailored under a $\Delta$-synchronous blockchain. Accordingly, with probability close to 1/2, the malicious user $P_1$ harvests double assets and the atomicity property is broken.

4 OUR SOLUTION IN A NUTSHELL

Recall the core root of double-claiming attack is that the frozen coins $\beta$ can be claimed by both users after timeout $T_1$. Therefore, our straightforward solution is to realize the pipelined coins flow for frozen coins $\beta$, that is, the corresponding swap transaction $tx^{(0)}_{swp}$ and refund transaction $tx^{(1)}_{rfd}$ cannot be both valid. As depicted in Fig.6, the key idea is how to force the early release of witness $y$, such that a valid transaction (i.e., either $tx^{(0)}_{swp}$ or $tx^{(1)}_{rfd}$) can be pre-determined at timeout $T_1$. Specifically, we introduce a “two-hop completion” method, called two-hop swap and two-hop refund, where two-hop swap forces user $P_0$ to post the pre-swap transaction $tx^{(0)}_{swp}$ $\varphi$ time before posting a valid swap transaction $tx^{(0)}_{swp}$, and the corresponding two-hop refund forces user $P_1$ to post the pre-refund transaction $tx^{(1)}_{rfd}$ (it is locked until time $T_1 - \varphi$) before refunding its frozen coins by transaction $tx^{(1)}_{rfd}$. Further, we use user $P_0$ actively posting $tx^{(0)}_{swp}$ before time $T_1$, otherwise it cannot generate a valid swap transaction $tx^{(0)}_{swp}$ before timeout $T_1$. As a result, if pre-swap transaction $tx^{(0)}_{swp}$ is finally confirmed before timeout $T_1$, coins $\beta$ can only be swapped by user $P_0$ otherwise, coins $\beta$ can only be refunded by user $P_1$ after timeout $T_1$. Now we walk through how we realize the “two-hop completion”.

First ingredient: splitting frozen coins $\beta$ into two parts. To ensure the flow direction of frozen coins $\beta$ and prevent potential risk of malicious user $P_0$ or $P_1$, the frozen coins $\beta$ are stored in two distinct outputs with values $\varepsilon$ ($\varepsilon > 0$ is arbitrarily small, i.e., $\varepsilon \rightarrow 0$) and $\beta - \varepsilon$ respectively. Specifically, coins $\varepsilon$, $\beta - \varepsilon$ can only be further spent with respective secret keys $sk^{(10)}$ and $sk^{(10)}$, which are shared between users $P_0$ and $P_1$.

Second ingredient: two-hop swap. To prevent the adversary $P_0$ from suddenly releasing its swap transaction $tx^{(0)}_{swp}$ near timeout $T_1$, a new swap method called two-hop swap is proposed. Specifically, user $P_0$ can generate a valid swap transaction $tx^{(0)}_{swp}$ only when its pre-swap transaction $tx^{(0)}_{swp}$ has been finally confirmed by the underlying blockchain. In particular, the transaction $tx^{(0)}_{swp}$ is jointly pre-signed by both users with respective key shares ($i.e., sk^{(10)}_0, sk^{(10)}_1$) and puzzle $Y$, while swap transaction $tx^{(0)}_{swp}$ takes $tx^{(0)}_{swp}$ and its inputs is jointly signed by both users with respective key shares $sk^{(10)}_0$ and $sk^{(10)}_1$, where statement-witness pair $(Y, y) \in R$ is selected by user $P_0$. That is, swap transaction $tx^{(0)}_{swp}$ is valid implying that user $P_0$ has posted the pre-swap
transaction $\overline{tx}_{\text{swap}}^{(0)}$ at least $\varphi$ time ago, where $\varphi$ is the confirmation latency of underlying blockchain. Since the witness $y$ has been released by the posted pre-swap transaction $\overline{tx}_{\text{swap}}^{(0)}$, user $P_1$ can generate its valid swap transaction $tx_{\text{swap}}^{(1)}$ at least $\varphi$ time earlier than user $P_0$.

**Third ingredient: two-hop refund.** To pre-determine the valid transaction among swap transaction $tx_{\text{swap}}^{(0)}$ and refund transaction $tx_{\text{refd}}^{(1)}$ at timeout $T_1$, we require witness $y$ to be released before time $T_1 := T_0 - \varphi$, that is, user $P_0$ should post pre-swap transaction $\overline{tx}_{\text{swap}}^{(0)}$ before time $T_1$. Accordingly, a new refund method called two-hop refund is proposed, where user $P_1$ can refund its frozen coins by transaction $tx_{\text{refd}}^{(1)}$ after timeout $T_1$ only when its pre-swap transaction $\overline{tx}_{\text{swap}}^{(0)}$ has been finally confirmed by the underlying blockchain, and the pre-refund transaction $\overline{tx}_{\text{refd}}^{(1)}$ is locked until time $T_1$. This means if pre-swap transaction $\overline{tx}_{\text{swap}}^{(0)}$ is confirmed before timeout $T_1$, it is impossible to generate a valid refund transaction $tx_{\text{refd}}^{(1)}$ even after timeout $T_1$; otherwise, if pre-refund transaction $\overline{tx}_{\text{refd}}^{(1)}$ is confirmed before timeout $T_1$, there is no valid swap transaction $tx_{\text{swap}}^{(0)}$ at any time. Nevertheless, there is a subtle issue remained. In particular, if user $P_0$ posts pre-swap transaction $\overline{tx}_{\text{swap}}^{(0)}$ almost near time $T_1$, then the adversary $P_1$ can initiate double-claiming attack via posting pre-refund transaction $\overline{tx}_{\text{refd}}^{(1)}$ immediately and making transactions $\overline{tx}_{\text{swap}}^{(0)}$ and $\overline{tx}_{\text{refd}}^{(1)}$ both valid, due to the fact that the adversary can always delay the delivery of honest messages within time $\Delta$.

To solve the issue, we just let the pre-swap transaction $\overline{tx}_{\text{swap}}^{(0)}$ be posted at least $\Delta$ time before time $T_1$.

Clearly, the final flow direction of frozen coins $\beta$ can be determined at timeout $T_1$. We now give an intuition that.pipeSwap satisfies atomicity:

- **Successful Swap:** If user $P_0$ is honest, pre-swap transaction $\overline{tx}_{\text{swap}}^{(0)}$ posted before time $T_1 - \Delta$ will be finally confirmed, and then swap transactions $tx_{\text{swap}}^{(0)}$ and $tx_{\text{swap}}^{(1)}$ could be validated before timeout $T_1$. Additionally, no valid refund transaction $tx_{\text{refd}}^{(1)}$ exists after timeout $T_1$. As a result, each user obtains its desired exchanged coins!
- **Failed Swap:** If user $P_0$ is malicious and user $P_1$ enters into its Swap Timeout Phase after timeout $T_1$, i.e., user $P_1$ cannot generate valid swap transaction $tx_{\text{swap}}^{(1)}$ before timeout $T_1$, then refund transaction $tx_{\text{refd}}^{(1)}$ will be valid after timeout $T_1$ and no valid swap transaction $tx_{\text{swap}}^{(0)}$ exists at any time. As a result, user $P_1$ can successfully refund its frozen coins.

5 FORMAL DEFINITION OF PIPESWAP

**Notations.** We denote by $\lambda$ the security parameter and by $A(x; \tau) \rightarrow z$ or $z \leftarrow A(x; \tau)$ the output $z$ of algorithm $A$ with inputs $x$ and randomness $\tau \in \{0, 1\}^\lambda$ (it is only mentioned explicitly when required). We write the events that “send message $m$ to $P$ at time $t''$ as “$m \rightarrow P''$” and “receive message $m$ from $P$ at time $t''$ as “$m \leftarrow P''$”, where $P$ could be a user or ideal functionality.

5.1 Modeling the System and Threats

We model security of cross-chain swaps in the Universal Composability (UC) model [12] and deploy the version with a global setup (GUC) [13]. We define the cross-chain swap model over users $P_0$ and $P_1$, and take the underlying blockchains $\mathbb{B} := \{\mathbb{B}_0, \mathbb{B}_1\}$ as global ideal functionalities $\mathcal{F}_\mathbb{B} := \{\mathcal{F}_{\mathbb{B}_0}, \mathcal{F}_{\mathbb{B}_1}\}$ with the confirmation delay time $\varphi$ (Fig.7).

**The User.** There are two designated users $P_0$ and $P_1$ to participate in the swap protocol. The malicious user (it is $P_0$ or $P_1$) is determined before the protocol starts, who can deviate arbitrarily from the swap protocol (e.g., delaying the posting of its transaction). There is a secure message transmission channel between users modeled by the ideal functionality $\mathcal{F}_{\text{int}}$ [12].

**Ideal Functionality $\mathcal{F}_\mathbb{B}(\Delta, \varphi)$**

1. **Initialization:** upon receiving the address-balance pair $(pk, v) \leftarrow \mathbb{Z}$, set $L := \{(pk_1, v_1), \ldots, (pk_\ell, v_\ell)\} \in \mathbb{R}_{\geq 0}^\ell$ store and send $L \rightarrow S$.

2. **Posting transaction:** upon receiving $\text{Post}(tx, t) \leftarrow \mathbb{Z}$, send $(post, tx, t) \rightarrow S$ if $\text{Valid}(tx) = 1$.
   - Upon receiving $(post, tx, t') \leftarrow S$, if $t' < t \leq \Delta$, then set $t := t'$; otherwise, set $t := t + \Delta$. Update list $\text{list} := \text{list} \cup (tx, t)$;
   - For conflicting transactions $(tx_0, t_0), (tx_1, t_1) \in \text{list}$, if $t_0 < t_1$, then remove $(tx_1, t_1)$ from list; else, if $t_0 = t_1$, then randomly select $b \in \{0, 1\}$ and remove $(tx_b, t_0)$ from list; otherwise, remove $(tx_0, t_0)$ from list;
   - For $(tx, t) \in \text{list}$, update $T := T \cup (tx, t)$ at time $t$.

3. **Confirming transaction:** upon receiving $\text{Confirm}(tx, t'' \leftarrow \mathbb{Z}$, if $(tx, t) \in T$ and $t'' - t \geq \varphi$, then update $\mathbb{L}$ as $tx.pk_{\text{in}}, \text{balance} := tx.pk_{\text{in}}, \text{balance} - v$ and $tx.pk_{\text{out}}, \text{balance} := tx.pk_{\text{out}}, \text{balance} + v$; otherwise, abort.

![Figure 7: The blockchain ideal functionality $\mathcal{F}_\mathbb{B}(\Delta, \varphi)$](image)

---

**\Delta-Synchronous Network.** We assume the network of underlying blockchains is \(\Delta\)-synchronous [33], i.e., the network delay is under adversarial control, up to a known delay upper bound $\Delta$. The adversary $A$ of the underlying blockchain can see the posted honest message but cannot modify or drop it.

**The Blockchain Ideal Functionality.** We take the underlying blockchain $\mathbb{B} (\mathbb{B}_0$ or $\mathbb{B}_1$) involved in the cross-chain swap protocol as a global ideal functionality (just as in [3, 42]) with confirmation delay time $\varphi$ that records the balance of each address (i.e., ledger $L$) and maintains a trusted append-only bulletin board $\mathcal{T}$, denoted as $\mathcal{F}_\mathcal{T}(\Delta, \varphi)$. More precisely, functionality $\mathcal{F}_\mathbb{B}$ offers interface $\text{Valid}(tx)$ to determine the validity of a transaction $tx$ (i.e., checking $\text{inputs} \in L$ have enough balance and they are signed correctly), uses interface $\text{Post}(tx, t)$ to add valid transaction $tx$ to bulletin board $\mathcal{T}$ at time $t' \leq t + \Delta$, where time $t'$ is determined by the simulator $\mathcal{S}$. We emphasize that $\mathcal{T}$ always prefers to accept the earlier arrived transaction, i.e., upon receiving $\text{Post}(tx_0, t_0)$ and $\text{Post}(tx_1, t_1)$ at time $t_1$, it will accept the $tx_0$ and update $\mathcal{T}$ accordingly.
Post\((tx_1, t_1)\) for posting conflicting transactions \(tx_0, tx_1, T\) will accept \(tx_0\) if \(t_0' < t_1'\) or randomly select one of \(\{tx_0, tx_1\}\) if \(t_0' = t_1'\). Additionally, functionality \(\mathcal{F}_b\) confirms transaction \(tx\) via interface \(\text{Confirm}(tx)\) (e.g., if \(tx\) has been recorded in \(T\) for time \(\varphi\), it updates \(L\) via removing coins from input address \(pk_{in}\) to output address \(pk_{out}\)). See Fig.7 for the details.

5.2 Ideal Functionality \(\mathcal{F}\) of Cross-Chain Swap

When taking a careful analysis of the ideal functionality \(\mathcal{F}_{\text{swap}}\) for fair swap of coins (Fig.3 in [42]), we have a fatal observation that \(\mathcal{F}_{\text{swap}}\) cannot cope with the condition of the users \(U_0\) and \(U_1\) simultaneously initiating their respective \textit{buy} and \textit{abort} requirements, which further confirms the \textit{double-claiming} attack in Universal Atomic Swaps.

\(\text{(A) Swap Setup Phase - Freezing Coins}\)

1. Upon receiving \((frz, id, pk_0^{(0)}, sk_0^{(0)}) \rightarrow P_0\), invoke subroutine \(\text{Freeze}(id, pk_0^{(0)}, sk_0^{(0)}, \alpha, pk_F^{(0)}, T_0)\); upon receiving \((\text{Confirmed}, id, ok) t_1^{\leq t_1 + \Delta} \rightarrow \mathcal{F}_{B_0}\), send \((frz, id, ok) t_1 \rightarrow P_0\);
2. Upon receiving \((frz, id, pk_1^{(1)}, sk_1^{(1)}) \rightarrow P_1\), invoke subroutine \(\text{Freeze}(id, pk_1^{(1)}, sk_1^{(1)}, \beta, pk_F^{(1)}, T_1)\); upon receiving \((\text{Confirmed}, id, ok) t_1^{\leq t_1 + \Delta} \rightarrow \mathcal{F}_{B_1}\), send \((frz, id, ok) t_1 \rightarrow P_1\);
3. After the above steps are successful, send \((\text{Setup}, id, ok) t_1 \rightarrow P_i\) \((i \in \{0, 1\})\) and proceed to procedure \(B\); otherwise, proceed to procedure \(C\).

\(\text{(B) Swap Complete Phase}\)

1. Upon receiving \((\text{sup}, id, pk_0^{(0)}_{\text{sup}}) t' \rightarrow P_0\), do the following:
   (1) if \(t' < T_1\), set \(b^{(0)} = 0\) and invoke subroutine \(\text{Transfer}(id, pk_F^{(0)}, sk_F^{(0)}, \beta, pk_{\text{swap}}^{(0)}, t')\); upon receiving \((\text{Post}, id, ok) t_1^{\leq t_1 + \Delta} \rightarrow \mathcal{F}_{B_0}\), send \((\text{sup}, id, ok) t_1 \rightarrow P_0\);
   (2) if \(t' = T_1\), set \(b^{(0)} \in \{0, 1\}\) and if \(b^{(0)} = 0\), invoke subroutine \(\text{Transfer}(id, pk_F^{(0)}, sk_F^{(0)}, \beta, pk_{\text{swap}}^{(0)}, t')\); upon receiving \((\text{Post}, id, ok) t_1^{\leq t_1 + \Delta} \rightarrow \mathcal{F}_{B_1}\), send \((\text{sup}, id, ok) t_1 \rightarrow P_0\);
2. Otherwise, abort.

2. Upon receiving \((\text{sup}, id, pk_1^{(1)}_{\text{sup}}) t' \rightarrow P_1\), do the following:
   (1) if \(b^{(0)} \in \{0, 1\}\), set \(b^{(1)} = 1\) and invoke subroutine \(\text{Transfer}(id, pk_F^{(1)}, sk_F^{(1)}, \alpha, pk_{\text{swap}}^{(1)}, t')\); upon \((\text{Post}, id, ok) t_1^{\leq t_1 + \Delta} \rightarrow \mathcal{F}_{B_0}\), send \((\text{sup}, id, ok) t_1 \rightarrow P_0\);
   (2) Otherwise, abort.

\(\text{(C) Swap Timeout Phase}\)

1. Upon receiving \((\text{rfd}, id, pk_1^{(1)}_{\text{rfd}}) t'' \rightarrow P_i\) \((i \in \{0, 1\})\), do the following:
   (1) if \(t'' = T_1\) \& \(b^{(1-i)} = 1 - i\), invoke subroutine \(\text{Unfreeze}(id, pk_F^{(1)}, sk_F^{(1)}, \alpha, pk_{\text{swap}}^{(1)}, t'')\); upon receiving \((\text{Post}, id, ok) t_1^{\leq t_1 + \Delta} \rightarrow \mathcal{F}_{B_1}\), send \((\text{rfd}, id, ok) t_1 \rightarrow P_i\);
   (2) Otherwise, abort.

To model security in our setting more comprehensively, we require that the ideal functionality \(\mathcal{F}\) guarantees stronger atomicity: either both users interested in the swap successfully swap their coins, or the swap fails and honest user refunds its coins and the malicious user may lose coins for its malicious manners. Accordingly, we allow ideal functionality \(\mathcal{F}\) to take actions according to the timeout \(T\) of each participating user.

Formally, ideal functionality \(\mathcal{F}\) (see Fig.8 and Fig.9) communicates with users \(P_0, P_1\), the environment \(S\), the simulator \(\mathcal{S}\), and the underlying blockchain functionalities \(\mathcal{F}_{B_0}\) and \(\mathcal{F}_{B_1}\). It consists of three procedures and each is triggered by one message sent by user \(P_i\) \((i \in \{0, 1\})\), including its request and the session \(id\).

(A) Swap Setup Phase-Freezing Coins: Users \(P_0\) and \(P_1\) initiate an \(\alpha\)-to-\(\beta\) swap with their respective \textit{freeze} messages \((frz, id, pk_0^{(0)}, sk_0^{(0)})\) and \((frz, id, pk_1^{(1)}, sk_1^{(1)})\), which specifies that the coins in addresses \(pk_0^{(0)}\) (owned by user \(P_0\)) and can be spent with secret key \(sk_0^{(0)}\) and \(pk_1^{(1)}\) (owned by user \(P_1\)) and can be spent with secret key \(sk_1^{(1)}\) are to be swapped. Ideal functionality \(\mathcal{F}\) calls subroutine \(\text{Freeze}(id, pk_0^{(0)}, sk_0^{(0)}, pk_F^{(0)}, T)\) to transfer coins in address \(pk_0^{(0)}\) to a specific address \(pk_F^{(0)}\) controlled by \(\mathcal{F}\) until timeout \(T_0\), where \(T_0 > T_1\).

(B) Swap Complete Phase: User \(P_0\) sends its swap message \((\text{sup}, id, pk_0^{(0)}_{\text{sup}})\) at time \(t'\). If \(t' < T_1\), \(F\) transfers coins from address \(pk_1^{(1)}\) to \(pk_0^{(0)}\) (controlled by user \(P_0\)) and sets \(b^{(0)} = 0\) to indicate that user \(P_0\) has successfully finished swap operation. If \(t' = T_1\) (it implies that user \(P_0\) is trying to initiate \textit{double-claiming} attack), \(\mathcal{F}\) randomly determines whether user \(P_0\) completes swap operation (i.e., \(b^{(0)} \in \{0, 1\}\)). Otherwise, if \(t' > T_1\), user \(P_0\) fails in swap phase (i.e., \(b^{(0)} = \perp\)). While user \(P_1\) can request \textit{swap} only when user \(P_0\) has initiated its swap operation (i.e., \(b^{(0)} = 0\)).

(C) Swap Timeout Phase: The frozen coins are released to the original owners after the respective timeout \(T_1\) and \(T_0\).

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{Fig8.png}
\caption{The ideal functionality \(\mathcal{F}\)}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{Fig9.png}
\caption{The subroutines}
\end{figure}
Remark 1. Careful readers might notice that the functionality $F$ finishes Swap Setup Phase only when the frozen transactions have been finally confirmed by the underlying blockchains, while, in Swap Complete Phase and Swap Time-out Phase, $F$ responds with ok as long as the corresponding swap transaction and refund transaction have been added in the respective bulletin board. The correctness lies in the facts that only the finally confirmed transaction can be further spent by a new transaction (where the frozen transaction will be spent by a swap or refund transaction) and, as defined in blockchain functionality $F_{\beta}$ (Fig.7), transactions in bulletin board can certainly be finally confirmed in time $\varphi$.

Security Analysis. Now we analyze that the ideal functionality $\mathcal{F}$ satisfies the stronger atomicity:

- **Successful Swap**: If user $P_0$ honestly initiates its swap operation before timeout $T_1$, then $\mathcal{F}$ will enable both users $P_0$ and $P_1$ to complete the swap via respectively transferring the frozen coins (controlled by $\mathcal{F}$) to their corresponding addresses $pk_{\text{swap}}^{(0)}$ and $pk_{\text{swap}}^{(1)}$ (i.e., $b^{(0)} = 0$ and $b^{(1)} = 1$). Especially, if the adversary $P_0$ tries to delay its swap operation until timeout $T_1$, then $\mathcal{F}$ will enable $P_0$ to complete swap (i.e., $b^{(0)} = 0$) with probability $\frac{1}{2}$, instead, $\mathcal{F}$ is certainly to transfer user $P_0$’s frozen coins to address $pk_{\text{swap}}^{(1)}$ (i.e., $b^{(1)} = 1$);
- **Failed Swap**: If the adversary $P_0$ fails to initiate its swap operation until timeout $T_1$, $\mathcal{F}$ will enable user $P_1$ to complete both swap and refund operations for $b^{(0)} = 1$, or just refund its own frozen coins for $b^{(0)} \neq 1$.

Obviously, the ideal functionality $\mathcal{F}$ is secure against the double-claiming attack, ensuring that regardless of how the malicious user behaves, the honest user never loses coins.

6 PIPESWAP: PROTOCOL DESCRIPTION

6.1 Cryptographic Building Blocks

To guarantee universality, we insist on the fundamental building blocks from [42]: adaptor signature [3] and verifiable timed discrete logarithm (VTD) [43].

**Adaptor Signature.** The adaptor signature (cf. Def.1 in Appendix B) allows users to insert a puzzle $Y$ (e.g., statement-witness pair $(Y, y) \in R$, cf. Def.3 in Appendix B) into the generation of a signature on message $m \in \{0, 1\}^k$. The user with secret key first computes a pre-signature $\sigma$ of message $m$ which by itself is not a valid signature, but can later be adapted into a valid signature $\sigma$ (i.e., $\sigma \in \{\text{SIG.V}_f(pk, m, \sigma) = 1\}$, cf. Def.2 in Appendix B) with witness $y$. In addition, witness $y$ can be further extracted by $\sigma$ and $\sigma$.

Here, the digital signature scheme satisfies the standard notion of unforgeability [5] and, to show the universality of our construction, we assume $\sigma \in \{\text{Schnorr, ECDSA}\}$ to capture most existing cryptocurrencies, e.g., Bitcoin, Ethereum and Ripple. Adaptor signature is required to satisfy security properties of unforgeability, witness extractability and pre-signature adaptability (cf. [42] for the detailed definitions).

**Verifiable Timed Dlog (VTD).** The VTD enables the committer to generate a timed commitment $C$ of value $x$ with timing hardness $T$, which can be verified publicly and forcibly opened in time $T$ (cf. Def.4 in Appendix B). VTD is required to satisfy the security properties of soundness and privacy (cf. [42] for the detailed definitions).

In this work, we use adaptor signature scheme and VTD in a black-box manner and refer the readers to [3, 36, 42, 43] for efficient constructions. In slightly more detail, as it is in [42], we adopt the construction of adaptor signature in [3], where the underlying signature scheme is Schnorr or ECDSA, and the hard relation $R$ is the discrete log (dlog) relation (i.e., the language is defined as $L_{\text{dlog}} := \{Y | \exists y \in \mathbb{Z}_p^*, s.t. Y = g^y \in G\}$). For the construction of VTD, the committer embeds the dlog.value $x$ inside a time-lock puzzle $H$, uses a non-interactive zero-knowledge proof (NIZK) to prove that $H$ can be solved in time $T$ and the value $x$ satisfies equation $H = g^x$, where such an efficient construction of NIZK [43] can be from the cut-and-choose techniques, Shamir secret sharing [40] and homomorphic time-lock puzzles [27].

Additionally, since the frozen address $pk$ is jointly controlled by users $P_0$ and $P_1$ (i.e., the corresponding secret key $sk$ is shared between them), it is inevitable that we rely on the interactive protocols (denoted as $\Gamma_{\text{AdpSig}}$ and $\Gamma_{\text{Sig}}$) to realize the jointly (pre-)signing of a message $m$ under public key $pk$, which can be efficiently instantiated w.r.t. $\sigma \in \{\text{Schnorr, ECDSA}\}$ with the protocols in [26].

6.2 Procedures of pipeSwap

Recall that in the classic setting, users $P_0$ (owns coins $\alpha$ on blockchain $B_0$) and $P_1$ (owns coins $\beta$ on blockchain $B_1$) wish to complete the $\alpha$-to-$\beta$ cross-chain swap. As we have briefly mentioned before (cf. Section 4), the pipelined coins flow of the frozen coins definitely guarantees atomicity. Forcing the earlier release of witness $y$ is the crux of making pipeSwap secure, and this is achieved by three critical ingredients, i.e., splitting frozen coins $\beta$ into $(\varepsilon, \beta - \varepsilon)$, two-hop swap and two-hop refund.

**Protocol details.** For ease of understanding, we illustrate the coin flow of pipeSwap in Fig.10 and describe pipeSwap in Fig.11, where the key point of each phase is presented below:

(A) Swap Setup Phase-Freezing Coins: This phase allows

![Figure 10: The coins flow of pipeSwap](image-url)
addresses \( \widetilde{pk}^{(10)} \) and \( pk^{(10)} \) with respective values \( \varepsilon \) and \( \beta - \varepsilon \). Importantly, to guarantee the frozen coins \( \beta \) are refunded as it is hoped, pre-refund transaction \( \mathcal{T}_{\text{refd}}^{(1)} := (\widetilde{pk}^{(10)}, \widetilde{sk}_{\text{refd}}^{(1)}) \) is locked until time \( T_1 := T_1 - \varphi \) (i.e., user \( P_0 \) makes a timed commitment of secret key share \( \widetilde{sk}_0^{(10)} \) with timing hardness \( \mathcal{T}_1 \)), while the refund transaction \( \mathcal{T}_{\text{rfd}}^{(1)} := ((pk^{(10)}, \widetilde{sk}_{\text{rfd}}^{(1)}, \widetilde{pk}_{\text{rfd}}^{(1)}, \beta) \) is generated with secret keys \( sk^{(10)} \) (jointly held by users \( P_0 \) and \( P_1 \) ) and \( \widetilde{sk}_{\text{rfd}}^{(1)} \) (held by user \( P_1 \) ), and will be valid at timeout \( T_1 \) (i.e., the time of confirming \( \mathcal{T}_{\text{rfd}}^{(1)} \)). Meanwhile, user \( P_1 \) makes a timed commitment of secret key share \( \widetilde{sk}_1^{(10)} \) with timing hardness \( T_0 > T_1 \) for refunding frozen coins \( \alpha \) (i.e., user \( P_0 \) can generate the refund transaction \( \mathcal{T}_{\text{tx}}^{(0)} := (pk^{(01)}, \widetilde{pk}_{\text{tx}}^{(0)}, \alpha) \) after timeout \( T_0 \));

(B1) Swap Lock Phase: This phase prepares for atomic swaps. Both users jointly pre-sign swap transactions \( \mathcal{T}_{\text{swp}}^{(0)} := (pk^{(01)}, \widetilde{pk}_{\text{swp}}^{(0)}, \alpha) \) and \( \mathcal{T}_{\text{swp}}^{(0)} := (pk^{(00)}, \widetilde{pk}_{\text{swp}}^{(0)}, \varepsilon) \) in sequence, where statement-witness pair \( (Y, y) \in R_{\text{swp}} \) is selected by user \( P_0 \). Similarly, to guarantee the coins \( \beta \) are swapped as it is hoped, the swap transaction \( \mathcal{T}_{\text{swp}}^{(0)} := ((pk^{(0)}, \widetilde{sk}_{\text{swp}}^{(0)}, \varepsilon), \beta) \) is well generated with secret keys \( sk^{(10)} \) (jointly held by users \( P_0 \) and \( P_1 \) ) and \( \widetilde{sk}_{\text{swp}}^{(0)} \) (held by user \( P_0 \)), and the final confirmation of transaction \( \mathcal{T}_{\text{swp}}^{(0)} \) is an essential prerequisite for the validity of \( \mathcal{T}_{\text{swp}}^{(0)} \). Thus, in order to generate a valid swap transaction, user \( P_0 \) is forced to release witness \( y \) at least \( \varphi \) time earlier (i.e., posting pre-swap transaction \( \mathcal{T}_{\text{swp}}^{(0)} \));

(B2) Swap Complete Phase: If user \( P_0 \) can generate a valid swap transaction \( \mathcal{T}_{\text{swp}}^{(0)} \) before timeout \( T_1 \) (i.e., it honestly posts pre-swap transaction \( \mathcal{T}_{\text{swp}}^{(0)} \) before time \( T_1 \)), then both swap transactions \( \mathcal{T}_{\text{swp}}^{(0)} \) and \( \mathcal{T}_{\text{swp}}^{(1)} \) must be finally confirmed by the underlying blockchains;

(C) Swap Timeout Phase: If user \( P_0 \) fails to generate a swap transaction \( \mathcal{T}_{\text{swp}}^{(0)} \) before timeout \( T_1 \) (i.e., pre-swap transaction \( \mathcal{T}_{\text{swp}}^{(0)} \) is not confirmed before timeout \( T_1 \) ), and pre-refund transaction \( \mathcal{T}_{\text{rfd}}^{(1)} \) is finally confirmed), then user \( P_1 \) posts refund transaction \( \mathcal{T}_{\text{swp}}^{(1)} \). Similarly, if user \( P_1 \) fails to post swap transaction \( \mathcal{T}_{\text{swp}}^{(1)} \) before timeout \( T_0 \), then user \( P_0 \) posts refund transaction \( \mathcal{T}_{\text{swp}}^{(0)} \). Note that above analysis includes the case that the adversary \( P_0 \) successfully posts a swap transaction \( \mathcal{T}_{\text{swp}}^{(1)} \) at timeout \( T_1 \) (i.e., it initiates a double-claiming attack), but this cannot prevent user \( P_1 \) from completing its Swap Complete Phase before timeout \( T_1 \).

Failed Swap. We consider the following possible cases: • User \( P_0 \) does not initiate its swap operation before timeout \( T_1 \) (i.e., it does not post transaction \( \mathcal{T}_{\text{swp}}^{(0)} \) before timeout \( T_1 \)), then both users can successfully refund their frozen coins; • User \( P_0 \) fails to generate a valid swap transaction \( \mathcal{T}_{\text{swp}}^{(0)} \) before/at timeout \( T_1 \) (i.e., pre-refund transaction \( \mathcal{T}_{\text{rfd}}^{(1)} \) is finally confirmed before timeout \( T_1 \)), user \( P_1 \) can successfully refund its frozen coins. Moreover, since user \( P_0 \) has posted pre-swap transaction \( \mathcal{T}_{\text{swp}}^{(0)} \), user \( P_1 \) also can extract witness \( y \) to complete Swap Complete Phase by posting swap transaction \( \mathcal{T}_{\text{swp}}^{(1)} \) before timeout \( T_1 \);

• The adversary \( P_0 \) can never generate a valid swap transaction \( \mathcal{T}_{\text{swp}}^{(0)} \) after timeout \( T_1 \), thus user \( P_1 \) can successfully refund its frozen coins after timeout \( T_1 \) and even swap \( P_0 \)'s frozen coins.

Therefore, pipeSwap runs as expected, and satisfies atomicity in that the honest user never lose coins.

6.3 Evaluation and Comparison

Implementation Details. We develop a prototypical C implementation to demonstrate the feasibility of our construction and evaluate its performance. We conduct experiments on the PC with the following configuration: CPU(Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz with 4 cores), RAM(16.0 GB) and OS(x64-based Windows). Basically, the signatures of Schnorr and ECDSA are instantiated over secp256k1 curve, and the transaction size is set to 250 bytes approximating the basic Bitcoin transaction. We implement the two-party computation protocol for digital signature \( \Gamma^{\Sigma} \) and use the implementations of adaptor signatures \( \Gamma^{\text{Adp}}_{\Sigma} \) and VTD respectively in [41] and [43]. Our code used in evaluations is available at https://github.com/Anqi333/pipeSwap.

Computation Time. We first measure the time of basic operations required in pipeSwap, and the results are shown in Table 2. Then we measure the computation time required by both users together in Table 3. We observe that (1) each instance of pipeSwap requires only 1.605 seconds for Schnorr and 1.624 seconds for ECDSA; (2) the computation time of Swap Setup-Freezing Phase accounts for more than 99%, because both users jointly complete two VTD computations.

<table>
<thead>
<tr>
<th>Table 3: The computation time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schnorr</td>
</tr>
<tr>
<td>UAS*</td>
</tr>
<tr>
<td>ECDSA</td>
</tr>
</tbody>
</table>

Communication Overhead. We measure the communication overhead as the amount of messages that users exchange during the execution of interactive algorithms in the Swap Setup Phase and Swap Lock Phase (cf. Table 4).
Assume the swapped coins $\alpha$ and $\beta$ are respectively stored in addresses $pk(0)$ and $pk(1)$ on the corresponding blockchains $B_0$ and $B_1$. Global parameters are $(G, q, g), \varDelta, \varphi, T_1 - T_1 \geq \varphi$ and $T_0 > T_1$; $+:=+ if SIG = Schnorr and $\oplus := \ast if SIG = ECDSA$.

(A) Swap Setup Phase - Freezing Coins

1. Users $P_0$ and $P_1$ respectively completes Setup:
   1) User $P_0$ runs Setup process (Fig.12) and sends $(pk(0), pk(10), pk(10), (C(1), \pi(1))) \rightarrow P_1$;
   2) User $P_1$ runs Setup process (Fig.12) and sends $(pk(0), pk(10), pk(10), (C(0), \pi(0))) \rightarrow P_0$.

2. Users $P_0$ and $P_1$ generate their frozen addresses:
   1) User $P_0$ does the following:
      - It checks if $\mathcal{D}_C(P_1)$, $(\mathcal{C}(0), \pi(0)) = 1$, and stops otherwise;
      - It generates frozen address $pk(0) = pk(0) \oplus pk(1)$.
   2) User $P_1$ does the following:
      - It checks if $\mathcal{D}_C(P_1)$, $(\mathcal{C}(1), \pi(1)) = 1$, and stops otherwise;
      - It generates frozen addresses $pk(10) = pk(10) \oplus pk(10)$.

3. Users $P_0$ and $P_1$ transfer swapped coins to the corresponding frozen addresses.
   1) User $P_0$ does the following:
      - It generates frozen transaction $tx_{frz}(0) := (pk(0), pk(0), \alpha)$ and signature $\sigma_{frz}^{0} \leftarrow \mathcal{SIG}.\mathcal{S}ig(sk(0), tx_{frz})$;
      - It posts $(tx_{frz}^{0}, \sigma_{frz}^{0})$ on blockchain $B_0$ and starts solving $\mathcal{VTD}.\mathcal{F}orceOp(C(0))$.
   2) Users $P_0$ and $P_1$ jointly do the following:
      - $P_1$ generates frozen transaction $tx_{frz}^{1} := (pk(1), (pk(0), pk(10)), (\epsilon, \beta - \epsilon))$, pre-refund transaction $tx_{frz}^{0} := (pk(10), pk(10), \epsilon)$ and refund transaction $tx_{frz}^{1} := ((pk(10), pk(10)), pk(0))$. It sends $(tx_{frz}^{0}, \sigma_{frz}^{0}, tx_{frz}^{1}) \rightarrow P_0$;
      - $P_0$ checks that transactions $(tx_{frz}^{0}, \sigma_{frz}^{0}, tx_{frz}^{1})$ are well formed (i.e., satisfy the two-hop refund framework), and stops otherwise;
      - $P_0$ and $P_1$ run a 2PC protocol $\Gamma_{\mathcal{S}ig}$ with input $(sk(0), sk(1), tx_{frz})$ (Fig.12) and obtain signature $\sigma(1)$.
      - $P_1$ computes signature $\sigma_{frz}^{1} \leftarrow \mathcal{SIG}.\mathcal{S}ig(sk(1), tx_{frz})$;
      - $P_1$ posts $(tx_{frz}^{1}, \sigma_{frz}^{1})$ on blockchain $B_1$ and starts solving $\mathcal{VTD}.\mathcal{F}orceOp(C(1))$.

(B1) Swap Lock Phase

1. User $P_0$ runs $(Y, y) \leftarrow \mathcal{R}Gen(1^3)$ and sends $Y \rightarrow P_1$.

2. Users $P_0$ and $P_1$ generate their swap transactions:
   1) User $P_0$ does the following:
      - It generates pre-swap transaction $\mathcal{E}_{swap}^{0} := (pk(10), pk(0), \epsilon)$ and swap transaction $tx_{swap}^{0} := (pk(10), pk(0), \epsilon)$;
      - It sends $(\mathcal{E}_{swap}, tx_{swap}) \rightarrow P_1$.
   2) User $P_1$ does the following:
      - It checks that transactions $(\mathcal{E}_{swap}, tx_{swap})$ are well formed (i.e., satisfy the two-hop swap framework), and stops otherwise;
      - It generates swap transaction $tx_{swap}^{1} := (pk(0), pk(10), \alpha)$;
      - It sends $tx_{swap}^{1} \rightarrow P_0$.

3. Users $P_0$ and $P_1$ run a 2PC protocol $\Gamma_{\mathcal{S}ig}$ with input $(sk(0), sk(1), tx_{swap})$ (Fig.12) and obtain pre-signature $\sigma_{swap}^{0}$.

4. After step 3 is successful, $P_0$ and $P_1$ run a 2PC protocol $\Gamma_{\mathcal{S}ig}$ with input $(sk(0), sk(1), tx_{swap})$ (Fig.12) and obtain signature $\sigma(0)$, and then run 2PC protocol $\Gamma_{\mathcal{S}ig}$ with input $(sk(0), sk(1), Y, \mathcal{E}_{swap})$ (Fig.12) and obtain pre-signature $\sigma_{swap}^{0}$.

(B2) Swap Complete Phase

5. User $P_0$ does the following:
   1) It computes $\sigma_{swap}^{0} \leftarrow \mathcal{SIG}.\mathcal{S}ig(sk_{swap}, tx_{swap})$ and posts $(\mathcal{E}_{swap}, tx_{swap})$ on blockchain $B_1$ before time $T_1 - \Delta$;
   2) It computes $\sigma_{swap}^{0} \leftarrow \mathcal{SIG}.\mathcal{S}ig(sk_{swap}, tx_{swap})$ and posts $(tx_{swap}, \sigma_{swap}^{0})$ on blockchain $B_1$ if $\mathcal{E}_{swap}$ is finally confirmed.

6. Upon receiving $(\mathcal{E}_{swap}, \sigma_{swap}^{0})$, user $P_0$ does the following:
   1) It computes $y \leftarrow \mathcal{SIG}.\mathcal{E}xt(\sigma_{swap}^{0}, \sigma_{swap}^{0}, Y)$ and $\sigma(1)_{\mathcal{S}ig} \leftarrow \mathcal{SIG}.\mathcal{E}xt(\sigma_{swap}^{0}, \sigma_{swap}^{0}, Y)$;
   2) It posts $(\mathcal{E}_{swap}, \sigma_{swap}^{0})$ on blockchain $B_0$

(C) Swap Refund Phase

1. If user $P_0$ fails to post $(\mathcal{E}_{swap}, \sigma_{swap}^{0})$ on blockchain $B_1$ before time $T_1$, then user $P_1$ does the following:
   1) It finishes computing $sk_{0}^{(0)} \leftarrow \mathcal{VTD}.\mathcal{F}orceOp(C(1))$ and computes $sk_{1}^{(0)} := sk_{0}^{(0)} \oplus sk_{1}^{(0)}$;
   2) It computes $\sigma_{frz}^{1} \leftarrow \mathcal{SIG}.\mathcal{S}ig(sk_{frz}, tx_{frz})$ and posts $(\mathcal{E}_{frz}, \sigma_{frz}^{1})$ on blockchain $B_1$;
   3) If $(\mathcal{E}_{frz})$ is finally confirmed, it computes $\sigma_{frz}^{1} \leftarrow \mathcal{SIG}.\mathcal{S}ig(sk_{frz}, tx_{frz})$ and posts $(tx_{frz}, \sigma_{frz}^{1})$ on blockchain $B_1$.

2. Similarly, if user $P_1$ fails to post $(\mathcal{E}_{frz}, \sigma_{frz}^{0})$ on blockchain $B_0$ before time $T_0$, then user $P_0$ does the following:
   1) It finishes computing $sk_{1}^{(0)} \leftarrow \mathcal{VTD}.\mathcal{F}orceOp(C(0))$ and computes $sk_{0}^{(0)} := sk_{0}^{(0)} \oplus sk_{1}^{(0)}$;
   2) It computes $\sigma_{frz}^{0} \leftarrow \mathcal{SIG}.\mathcal{S}ig(sk_{frz}, tx_{frz})$ and posts $(\mathcal{E}_{frz}, \sigma_{frz}^{0})$ on blockchain $B_0$.

Figure 11: pipeSwap: a secure cross-chain swap between users $P_0$ and $P_1$.
In particular, pipeSwap requires 6.4 kb for Schnorr and 7 kb for ECDSA, which is dominated by that of respectively exchanging the VTD.Commit-proof of secret key share.

Table 4: The communication overheads (bytes)

<table>
<thead>
<tr>
<th></th>
<th>Setup Phase</th>
<th>Lock Phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schnorr</td>
<td>pipeSwap</td>
<td>5690</td>
</tr>
<tr>
<td></td>
<td>UAS*</td>
<td>1518</td>
</tr>
<tr>
<td>ECDSA</td>
<td>pipeSwap</td>
<td>1240</td>
</tr>
<tr>
<td></td>
<td>UAS*</td>
<td>1012</td>
</tr>
</tbody>
</table>

*Universal Atomic Swaps [42].

**Efficiency Comparison.** To compare pipeSwap (Fig.6) with Universal Atomic Swaps (its Fig.5 in [42]) w.r.t. the operations required of both users together, we evaluate pipeSwap and Universal Atomic Swaps with the same setting and libraries and security parameters for all cryptographic implementations. Before delving into details, let us be clear first that Universal Atomic Swaps repeat the same operations for each swapped coin, thus we only consider the one-to-one atomic swap in [42]. Universal Atomic Swaps complete in 1.6 seconds for Schnorr and 1.617 seconds for ECDSA, and requires 5.1 kb for Schnorr and 5.4 kb for ECDSA. Therefore, pipeSwap is only 7 ms slower and incurs extra ≤ 1.6 kb communication overhead, which is acceptable, even though we need to prepare and sign two extra transactions (the pre-swap and pre-refund transactions). Additionally, we stress that pipeSwap sets the same timeout parameters \( T_0 \) and \( T_1 \) as Universal Atomic Swaps, while the actual hardness parameter for user \( P_1 \) is \( T_1 < T_1 \), which means that pipeSwap takes less computational costs.

Therefore, pipeSwap not only achieves stronger atomicity and universality, but also is efficient with low overhead.

### 7 CONCLUSIONS AND FUTURE WORKS

In this paper, we identify a new form of attack, called double-claiming attack, against Universal Atomic Swaps [IEEE S&P’22]. This attack can lead to honest user losing coins with overwhelming probability and atomicity property is directly broken. We introduce a novel approach of utilizing two-hop swap and two-hop refund techniques to secure coin flows, and design pipeSwap, a universal atomic cross-chain swap protocol.

Several interesting questions can be considered in future work. pipeSwap is efficiently instantiated with standard signature schemes Schnorr and ECDSA, and direct building on other signature schemes may need further care. It is interesting to explore extensions to some more complex but practical scenarios, e.g., multi-hop swaps \( P_1 \to P_2 \to \cdots \to P_n \to P_1 \), where each intermediate user only holds the desired coins of its right neighbor. Also, we leave to further work how to apply the pipelined coins flow paradigm to scriptless payment channel networks protocols providing stronger security.
REFERENCES


A RELATED WORKS

Tier Nolan first introduced the conceptualisation of “atomic swap” [32]. Its fundamental security atomicity states that the swap either ends with success (i.e., the owners of involved coins are exchanged) or failure (i.e., the involved coins are refunded to their original owners) [18, 47].

Essentially, a secure cross-chain swap between users \( P_0 \) and \( P_1 \) should fulfill two fundamental functionalities to guarantee the honest user \( P_0 \) cannot lose coins (1) if user \( P_1 \) has claimed \( P_0 \)'s frozen coins, \( P_0 \) is able claim \( P_0 \)'s frozen coins before \( P_1 \) can refund them; and (2) if user \( P_1 \) is malicious, \( P_0 \) can refund its frozen coins. While atomicity is easily realized by the trusted third party, the blockchain community has made significant efforts to achieve (fully) decentralized cross-chain swaps [30, 47, 48]. HTLC-based protocols use the rich scripting languages supported by the underlying blockchains to describe when and how the frozen coins can be unlocked [8, 31]. Subsequently, HTLC-style solutions have been widely applied and deployed in practice [10, 11, 19]. However, these protocols are far from the universal solution and suffer from the inherent drawbacks of HTLC, including high execution costs and large transaction sizes. Additionally, since these transactions are easier to distinguish from the standard one that does not include any custom scripts, thus these protocols are at odds with the blockchains that have already achieved privacy [38].

Recently, LightSwap [20] studies the swap that enables the user to run an instance of swap on a mobile phone and is committed to proposing the first secure atomic swap protocol that does not require the timeout functionality by one of the two participating users. However, it still requires one of the two involved blockchains supporting timelock functionality and thus cannot achieve universality. Universal Atomic Swaps [42] use cryptographic building blocks adaptor signature and timed commitment to present the first fully universal solution.

Besides, some literatures [24, 45] use a third blockchain as the coordinator, but it requires the involved users with the capability of transferring to/from coins from this blockchain. Also, the cross-chain swaps functionality is inserted into a trusted hardware [7], which is not only unrealistic but also exists serious vulnerabilities [9, 14].

The studies of payment channel networks (PCNs) enable any two users to complete payments even if they do not have a direct payment channel, and have become the most widely deployed solution for realizing blockchain scalability (e.g., lightning network [35]). Similarly, most of the existing PCN proposals are restricted to the Turing complete scripting language [16, 23, 28] thus suffering from the inherent drawbacks of HTLC. Anonymous Multi-Hop Locks [26], lockable signatures [44] and A²L [41] are recently proposed to construct scriptless PNC. However, these studies only defer successful atomic payment to specific signature schemes but still rely on on-chain time-lock functionality to ensure payment expiry, and thus is not universal.

B DEFINITIONS OF CRYPTOGRAPHIC BUILDING BLOCKS

**Definition 1.** (Adaptor Signature) [9] An adaptor signature scheme \( \Sigma_{AS}^\Sigma \) w.r.t. a hard relation \( \mathcal{R} \) and a digital signature scheme \( \Sigma_{SIG} := (KGen, Sig, Vf) \) consists of algorithms \( \{pSig, Adapt, pVf, Ext\} \) defined as:

1. \( pSig(sk,m,Y) \rightarrow \sigma \): The pre-signing algorithm inputs secret key \( sk \), message \( m \in \{0,1\}^\lambda \) and statement \( Y \in \mathcal{L}_R \), outputs pre-signature \( \sigma \);
2. \( pVf(pk,m,Y,\sigma) \rightarrow b \): The pre-verification algorithm inputs public key \( pk \), message \( m \in \{0,1\}^\lambda \), statement \( Y \in \mathcal{L}_R \) and pre-signature \( \sigma \), outputs a bit \( b \in \{0,1\} \);
3. \( Adapt(\sigma,y) \rightarrow \tilde{\sigma} \): The adaptor algorithm inputs pre-signature \( \sigma \) and witness \( y \), outputs signature \( \tilde{\sigma} \);
4. \( Ext(\sigma,\tilde{\sigma},Y) \rightarrow y \): The extraction algorithm inputs signature \( \sigma \), pre-signature \( \tilde{\sigma} \) and statement \( Y \in \mathcal{L}_R \), outputs witness \( y \) such that \( (Y,y) \in \mathcal{R} \).

**Definition 2.** (Digital Signature) A digital signature scheme \( \Sigma_{SIG} \) consists of algorithms \( \{KGen, Sig, Vf\} \) defined as:

1. \( KGen(\lambda) \rightarrow (pk,sk) \): The key generation algorithm inputs security parameter \( \lambda \) and outputs a public-secret key pair \( (pk,sk) \);
2. \( Sig(sk,m) \rightarrow \sigma \): The signing algorithm inputs secret key \( sk \) and a message \( m \in \{0,1\}^\lambda \), outputs a signature \( \sigma \);
3. \( Vf(pk,m,\sigma) \rightarrow b \): The verification algorithm inputs the verification key \( pk \), message \( m \) and signature \( \sigma \), outputs \( b = 1 \) if \( \sigma \) is a valid signature of \( m \) under public key \( pk \) and \( b = 0 \) otherwise.

**Definition 3.** (Hard Relation) A hard relation \( \mathcal{R} \) is described as \( \mathcal{L}_R := \{Y|3y, s.t. (Y,y) \in \mathcal{R}\} \) and satisfies:

1. \( RGen(\lambda) \rightarrow (Y,y) \): The sampling algorithm takes in input security parameter \( \lambda \) and outputs statement-witness pair \( (Y,y) \in \mathcal{R} \);
2. The relation is poly-time decidable;
3. There is no adversary \( A \) with statement \( Y \) can output witness \( y \) with non-negligible probability.
Definition 4. (Verifiable Timed Dlog) A VTD w.r.t. a group \( \mathbb{G} \) with prime order \( q \) and generator \( g \) consists of four algorithms (Commit, \( V.f \), Open, ForceOp) defined as:

1) \( \text{Commit}(x, r, T) \rightarrow (C, \pi) \): The commitment algorithm inputs discrete log \( x \in \mathbb{Z}_q^* \), randomness \( r \in \{0, 1\}^\lambda \) and timing hardness \( T \), outputs commitment \( C \) and proof \( \pi \);
2) \( \text{V.f}(H, C, \pi) \rightarrow b: \) The verification algorithm inputs group element \( H := g^x \), \( C \) and \( \pi \), outputs \( b \) \( = 1 \) if \( C \) is a valid commitment of \( x \) with hardness \( T \) and \( b = 0 \) otherwise;
3) \( \text{Open}(C) \rightarrow (x, r): \) The open algorithm inputs commitment \( C \), outputs the committed value \( x \) and randomness \( r \);
4) \( \text{ForceOp}(C) \rightarrow x: \) The force open algorithm inputs commitment \( C \) and outputs the committed value \( x \).

C SECURITY ANALYSIS

Theorem 1. (Atomicity) Assume \( \Sigma_{SIG} \) is a secure adapter signature scheme w.r.t. a secure digital signature scheme \( \Sigma_{SIG} \) and a hard dlog relation \( R_d \); protocols \( \Gamma_{SIG}^{\text{Sig}} \) and \( \Gamma_{AdpSIG} \) are UC-secure 2PC protocols for jointly computing \( \Sigma_{SIG}.\text{Sig} \) and \( \Sigma_{SIG}.\text{AdpSig} \); VTD is a secure timed commitment. Then protocol pipeSwap running in the (\( F_3 \), \( F_{\text{ent}} \))-hybrid world UC-realizes ideal functionality \( F \).

Proof. We now prove that protocol pipeSwap (Fig.6) UC-realizes the cross-chain swap ideal functionality \( F \) (Fig.8).

To show the indistinguishability between the ideal world and the real world, we construct a simulator \( S \) to simulate the protocol pipeSwap in the real world while interacting with the ideal functionality \( F \). At the beginning, \( S \) corrupts one or more of \( \{P_0, P_1\} \) as \( A \) does. We begin with the real world protocol execution, gradually change the simulation in these hybrids and then argue about the proximity of neighbouring experiments.

- **Hybrid \( H_0 \)**: It is the same as the real world protocol execution (Fig.11);
- **Hybrid \( H_1 \)**: It is the same as the above execution except that the 2PC protocol \( \Gamma_{SIG}^{\text{Sig}} \) in the Swap Setup Phase and Swap Lock Phase to generate signatures is simulated using the 2PC simulators \( S_{\text{pc}, 1} \) for the corrupted user (notice that such a simulator exists for a secure 2PC protocol \( \Gamma_{SIG}^{\text{Sig}} \));
- **Hybrid \( H_2 \)**: It is the same as the above execution except that the 2PC protocol \( \Gamma_{AdpSIG} \) in the Swap Lock Phase to generate pre-signatures is simulated using the 2PC simulators \( S_{\text{pc}, 2} \) for the corrupted user;
- **Hybrid \( H_3 \)**: It is the same as the above execution except that the adversary corrupts user \( P_1 \) and outputs a valid swap transaction \((tx_{\text{supp}}, \sigma_{\text{supp}})\) before the simulator initiates swap operation on behalf of \( P_0 \), the simulator aborts;
- **Hybrid \( H_4 \)**: It is the same as the above execution except that the adversary corrupts user \( P_0 \) and outputs a valid swap transaction \((tx_{\text{supp}}, \sigma_{\text{supp}})\) before timeout \( T_1 \). The simulator outputs \((tx_{\text{supp}}, \sigma_{\text{supp}})\) and if \( \Sigma_{SIG}.V.f(pk^{(0)}, tx_{\text{supp}}, \sigma_{\text{supp}}) \neq 1 \), the simulator aborts;
- **Hybrid \( H_5 \)**: It is the same as the above execution except that the adversary corrupts users \( P_0 \) and \( P_1 \) and outputs a valid swap transaction \((tx_{\text{supp}}, \sigma_{\text{supp}})\) before timeout \( T_1 \), the simulator cannot obtains its valid swap transaction. The probability of the event triggered in \( H_5 \) is negligible;
$\mathcal{H}_4 \approx \mathcal{H}_5 \approx \mathcal{H}_6$: The only difference between the hybrids is that in $\mathcal{H}_5$ and $\mathcal{H}_6$ the simulator aborts, if the adversary initiates a (valid) swap operation at timeout $T_1$, the simulator cannot post its valid swap transaction or refund transaction. With the security of underlying blockchain, adaptor signature and VTD, the probability of the events triggered in $\mathcal{H}_5$ and $\mathcal{H}_6$ is negligible;

$\mathcal{H}_6 \approx \mathcal{H}_7$: The only difference between the hybrids is that in $\mathcal{H}_7$ the simulator aborts, if the adversary initiates a swap operation after timeout $T_1$, the simulator cannot post its valid refund transaction. With the security of underlying blockchain and VTD, the probability of the event triggered in $\mathcal{H}_7$ is negligible;

$\mathcal{H}_7 \approx \mathcal{H}_8$: The only difference between the hybrids is that in $\mathcal{H}_8$ the simulator aborts, if the adversary $P_0$ outputs a valid refund transaction before timeout $T_0$. With the security of VTD, the probability of the event triggered in $\mathcal{H}_8$ is negligible;

$\mathcal{H}_8 \approx \mathcal{H}_9$: The only difference between the hybrids is that in $\mathcal{H}_9$ the simulator aborts, if the adversary $P_1$ outputs a valid refund transaction before timeout $T_1$. With the security of underlying blockchain and VTD, the probability of the event triggered in $\mathcal{H}_9$ is negligible;

$\mathcal{H}_9 \approx \mathcal{H}_{10}$: The only difference between the hybrids is that in $\mathcal{H}_{10}$ the simulator aborts, if it cannot post a valid refund transaction after timeout $T_1$. With the security of underlying blockchain and VTD, the probability of the event triggered in $\mathcal{H}_{10}$ is negligible;

$\mathcal{H}_{10} \approx \mathcal{H}_{11}$: The only difference between the hybrids is that in $\mathcal{H}_{11}$ the simulator aborts, if it cannot post a valid refund transaction after timeout $T_0$. With the security of VTD, the probability of the event triggered in $\mathcal{H}_{11}$ is negligible.