Universal Atomic Swaps: Secure Exchange of Coins Across All Blockchains

Sri AravindaKrishnan Thyagarajan
Carnegie Mellon University
t.srikrishnan@gmail.com

Giulio Malavolta
Max Planck Institute for
Security and Privacy
giulio.malavolta@hotmail.it

Pedro Moreno-Sanchez
IMDEA Software Institute
pedro.morenosanchez@imdea.org

Abstract—Trading goods lies at the backbone of the modern economy and the recent advent of cryptocurrencies has opened the door for trading decentralized (digital) assets: A large fraction of the value of cryptocurrencies comes from the inter-currency exchange and trading, which has been arguably the most successful application of decentralized money. The security issues observed with centralized, custodial cryptocurrency exchanges have motivated the design of atomic swaps, a protocol for coin exchanges between any two users. Yet, somewhat surprisingly, no atomic swap protocol exists that simultaneously satisfies the following simple but desired properties: (i) non-custodial, departing from a third party trusted holding the coins from users during the exchange; (ii) universal, that is, compatible with all (current and future) cryptocurrencies; (iii) multi-asset, supporting the exchange of multiple coins in a single atomic swap.

From a theoretical standpoint, in this work we show a generic protocol to securely swap \( n \) coins from any (possible multiple) currencies for \( \tilde{n} \) coins of any other currencies, for any \( n \) and \( \tilde{n} \). We do not require any custom scripting language supported by the corresponding blockchains, besides the bare minimum ability to verify signatures on transactions. For the special case when the blockchains use ECDSA or Schnorr signatures, we design a practically efficient protocol based on adaptor signatures and time-lock puzzles. As a byproduct of our approach, atomic swaps transactions no longer include custom scripts and are identical to standard one-to-one transactions. We also show that our protocol naturally generalizes to any cycle of users, i.e., atomic swaps with more than two participants. To demonstrate the practicality of our approach, we have evaluated a prototypical implementation of our protocol for Schnorr/ECDSA signatures and observed that an atomic swap requires below one second on commodity machines. Even on blockchains with expressive smart contract support (e.g., Ethereum), our approach reduces the on-chain cost both in terms of transaction size and gas cost.

Index Terms—Atomic Swaps, Adaptor Signatures, Blockchain

I. INTRODUCTION

Blockchains coexisting today are no longer isolated siloes, but their value rather comes from the exchange of assets across them. Cross-chain communication \([1]\) is thus found a critical component on cryptocurrency exchanges and transfers. In fact, trading is arguably among the top applications in the cryptocurrency landscape and numerous competing interoperability projects, attempting to unite otherwise independent blockchains, have been deployed in practice creating a multi-billion dollar industry \([2]–[5]\).

On a technical level, a cross-chain swap involves two ledgers \( \mathbb{B}_A, \mathbb{B}_B \) and two users Alice and Bob. Alice holds \( \alpha \) assets on \( \mathbb{B}_A \) while Bob holds \( \beta \) assets on \( \mathbb{B}_B \). The atomic swap problem consists in ensuring that Alice transfers \( \alpha \) to Bob in \( \mathbb{B}_A \) if and only if Bob transfers \( \beta \) to Alice in \( \mathbb{B}_B \). We then say that a solution is atomic if it ends in either of the following outcomes: (i) Bob owns \( \alpha \) in \( \mathbb{B}_A \) and Alice owns \( \beta \) in \( \mathbb{B}_B \) (i.e., asset swap); or (ii) Alice owns \( \alpha \) in \( \mathbb{B}_A \) and Bob owns \( \beta \) in \( \mathbb{B}_B \) (i.e., asset refund). Moreover, an atomic swap must preserve fungibility, that is, an observer (other than Alice and Bob) of the ledger should not distinguish a transfer executed as part of an atomic swap from a standard asset transfer in such ledger.

Despite being a fundamental problem in the cryptocurrencies landscape, the state of the art for atomic swaps protocols is rather chaotic: Atomic swaps protocols are typically tailored to the characteristics offered by a restricted set of currencies (e.g., Turing-complete scripting languages) \([6]–[9]\), require the existence of a third ledger to coordinate the swap \([5], [10]\), require additional trust assumptions such as trusted hardware \([11]\), or require the presence of a trusted third party like an online exchange. As a result, adding a new cryptocurrency in the market most likely requires one to design a new ad-hoc atomic swap protocol. Even among existing cryptocurrencies, atomic swaps (and consequently secure exchange) are limited to a handful of combinations of cryptocurrencies, or require one to accept strong trust assumptions.

On a conceptual level, such atomic swap of coins across currencies enables the most basic form of inter-chain connectivity, irrespective of the application. In this work, we ask whether one can obtain a universal solution for such atomic swaps: a Swiss Army knife protocol that works for all cryptocurrencies with minimal support from the blockchain. In order to answer this question, we first analyze existing approaches and argue why they fall short of our quest for a universal solution.

A. Where Existing Approaches Fall Short

We provide a summary of the comparison in Table I and we elaborate on each approach individually in the following. Third Ledger. Migration protocols like cryptocurrency-backed assets or side-chains, mimic the atomic swap functionality by requiring that Alice and Bob migrate their assets from (perhaps language-restricted) ledgers \( \mathbb{B}_A \) and \( \mathbb{B}_B \) into \( \mathbb{B}_C \) with a more expressive scripting language (like Ethereum) \([10]\). Once the funds are in \( \mathbb{B}_C \), they are swapped and sent back to the initial ledgers \( \mathbb{B}_A \) and \( \mathbb{B}_B \). Such an approach has the
following drawbacks: (i) the swap protocol imposes transaction and cost overhead not only to $B_A$ and $B_B$ but also $B_C$; and (ii) it is not an universal solution as it potentially requires a different asset migration as well as atomic swap protocols for each ledger that plays the role of $B_C$ in the aforementioned example. Even in the unlikely case that everybody agrees to use $B_C$ for their atomic swaps, we need to enhance every blockchain with the capability of migrating assets to/from $B_C$.

**Use of Trusted Hardware.** An alternative approach would be to defer the atomic swap functionality to a trusted execution environment (TEE) [11]. Besides the fact that this solution requires all users to have such a TEE (which is unrealistic), recent works have shown serious TEE vulnerabilities [12], [13].

**Hash Timelock Contracts.** The closest solution to a universal protocol for atomic swaps (which is implemented in the large majority of trustless exchange protocols) relies on hash-time lock contract (HTLC), an excerpt of script that implements the following logic: On input a tuple $(\alpha, h, t, A, B)$, where $\alpha$ denotes the assets to be transferred, $h$ is a hash value, $t$ denotes a certain timeout, $A$ and $B$ represent two users in the ledger, the HTLC contract transfers $\alpha$ to $B$ if it is invoked before time $t$ on input a value $r$ such that $h = H(r)$, where $H(\cdot)$ is a cryptographic hash function. If the HTLC contract is invoked after time $t$, it transfers the $\alpha$ assets to $A$ unconditionally.

Using HTLCs as building block, an atomic cross-chain swap where Alice exchanges $\alpha$ assets in $B_A$ for $\beta$ assets from Bob in $B_B$, is then realized as follows: Alice chooses $r$, computes $h = H(r)$, transfers $\alpha$ into an HTLC$(\alpha, h, t, A, B)$ in $B_A$ and sends $h, t$ to Bob. Note that at this point Bob cannot claim the HTLC because $r$ is only known to Alice. Instead, Bob finishes the setup of the exchange by choosing a time $t' < t$ and transferring his $\beta$ assets into an HTLC$(\beta, h', t', B, A)$ in $B_B$. The atomicity of the swap is enforced by the logic of the HTLC: (Case i) Alice claims the HTLC in $B_A$, effectively revealing $r$ to Bob (and anyone observing $B_B$) before $t'$. Bob can then use $r$ to claim the HTLC in $B_A$; (Case ii) Alice does not claim the HTLC in $B_B$ before time $t'$, then she does not reveal $r$, ensuring that Bob cannot claim the HTLC in $B_A$.

HTLC-based cross-chain swaps are deployed in practice [14], [15] and have a wide range of applications [18]–[20]. However, they incur high execution costs (e.g., gas in Ethereum), as well as large transaction sizes (due to large scripts), and they have inherent challenges that reduce their utility, which we summarize next.

(1) **Compatibility of the Hash Function.** Both ledgers must support compatible hash functions within their scripting language. In fact, they both should support the same hash function and each ledger must use the same number of bits to represent it, otherwise atomicity does no longer hold [21] since one ledger may not allow pre-images of a large enough size. Apart from atomicity, the use of the same $h$ value at two ledgers raises a privacy concern, as an observer can link both HTLC as part of the same swap. Finally, a perhaps more fundamental issue, is that there exist several cryptocurrencies such as Monero [22], Mimblewimble [23], Ripple [24], Stellar [25] or Zcash [20] (shielded addresses) that do not support the computation of the HTLC contract in their scripting language.

(2) **Presence of the Timelock.** Another issue with the HTLC approach is that both ledgers must support the timelock functionality in their scripting language, in other words, the possibility to lock the spending of coins by a certain user until a certain time (e.g., defined as block height) is reached. Adding this timelock functionality is at odds with privacy because (i) if implemented naively, it makes time-locked transfers easier to distinguish from those transfers without time restrictions [27]; (ii) including it conflicts with other privacy operations already available at the ledger [28]; (iii) if possible to include and implemented in a privacy-preserving manner, it adds a non-trivial overhead to the computation and storage overhead of the ledger [27], [29]. In this state of affairs, there exist cryptocurrencies that have been created with privacy by design and avoid the use of timelocked assets as a design principle [30].

(3) **Single-Asset Swap.** The swap is restricted to only two HTLC, one per ledger, and thus to the exchange of $\alpha$ assets in $B_A$ and $\beta$ assets in $B_B$ (e.g., $\alpha$ bitcoins by $\beta$ ethers). However, given the huge differences in value in current cryptocurrencies (e.g., 1 bitcoin is worth 100× more than 1 ether at the time of writing), current atomic swaps are restricted to small values of $\alpha$ (or $\beta$) to be able to match the exchange offer. However, in practice there exist users (e.g., market makers or exchanges) that have a varied portfolio and hold assets at different ledgers, who could potentially use several of their assets to match a swap offer, if multi-asset swaps were available. For instance, multi-asset swaps for the first time would allow Alice to use coins she owns at Ethereum, Monero and Ripple to match an exchange offer from Bob of 1 Bitcoin through a single atomic swap operation.

**B. Our contributions**

Our goal is to design a universal atomic swap protocol, that does not make any assumption on any specific features of the blockchain, or scripting functionality of the currencies and only assumes the (arguably minimal) ability to verify signatures on transactions. Besides establishing an important feasibility result, such a protocol immediately enables secure exchange protocols across all combination of cryptocurrencies, excluded by current ad-hoc solutions. As a byproduct of such a generality, such atomic swap transactions are identical to standard one-to-one transactions, thereby increasing the fungibility of the swapped coins. The contributions of this work can be summarized as follows.

(1) **Universal Swaps.** We establish the theoretical foundations of universal atomic swaps by presenting the first protocol to securely exchange any $n$ coins (possibly) from different currencies for any $\tilde{n}$ coins from another (possibly any disjoint)

<table>
<thead>
<tr>
<th>Table 1: Comparison among different approaches</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Approach</strong></td>
</tr>
<tr>
<td>HTLC [12], [13]</td>
</tr>
<tr>
<td>Smart contracts [10], [16], [17]</td>
</tr>
<tr>
<td>Tailored (This work)</td>
</tr>
</tbody>
</table>

*Technical Note: The table above compares different approaches for universal atomic swaps, highlighting their required functionalities, fungibility, and support for n-to-h exchanges.*
We show that our protocols can be adapted to securely without requiring additional trust assumptions such as use of fungibility. One appealing aspect of our approach is that cyclic swaps. We show that our protocols naturally lends efficient protocol for Schnorr/ECDSA. For the special transactions in the Bitcoin testnet [37]–[41] (the solution can the reader identify such atomic swap transaction among the five swap of a certain amount of bitcoin for ether. We let the Bitcoin testnet transaction that corresponds to the atomic the size of the blockchain. to improve the fungibility of the coins and the scalability of trusted hardware or a trusted third party. This has the potential this is the first protocol that achieves this form of privacy standard one-to-one transactions. To the best of our knowledge, signed transactions resulting from atomic swaps are identical to privacy preserving currency. Our techniques can also be efficiently extended to the transaction scheme of Monero [22], [35], [36], the largest market capitalization such as Bitcoin, Ethereum, Ripple, or Stellar. Our techniques can also be efficiently extended to the transaction scheme of Monero [22], [35], [36], the largest privacy preserving currency.

(3) Cyclic Swaps. We show that our protocols naturally lends themselves to interesting extensions, such as supporting cyclic swaps, i.e. atomic swaps involving more than two users. As an example, consider the scenario where Alice wants to exchange some ether for some bitcoins with Carol, who accepts only credit in Ripple. This can be done with the help of Bob, who is willing to exchange ethers for ripples. This translates into

\[
\begin{align*}
\text{Alice} & \xrightarrow{\text{1 ETH}} \text{Bob} \xrightarrow{\text{1 XRP}} \text{Carol} \xrightarrow{\text{1 BTC}} \text{Alice}
\end{align*}
\]

We show that our protocols can be adapted to securely implement swaps among user cycles (for any cycle length) without the need to place additional trust assumptions. We defer the details to the full version.

(4) Fungibility. One appealing aspect of our approach is that signed transactions resulting from atomic swaps are identical to standard one-to-one transactions. To the best of our knowledge, this is the first protocol that achieves this form of privacy without requiring additional trust assumptions such as use of trusted hardware or a trusted third party. This has the potential to improve the fungibility of the coins and the scalability of the currency, since it decreases the impact of atomic swaps on the size of the blockchain.

As an amusing exercise for the reader, we have carried out the Bitcoin testnet transaction that corresponds to the atomic swap of a certain amount of bitcoin for ether. We let the reader identify such atomic swap transaction among the five transactions in the Bitcoin testnet [37]–[41] (the solution can be found in Appendix A).

(5) Implementation and Optimizations. We have implemented a prototype of our Schnorr/ECDSA based protocol (Section VI) and evaluated it showing that one instance of universal swap can be executed in less than one second. In order to achieve such performance, we not only take advantage of the parallelization possible in our protocol as operations for different coins are independent of each other, but also describe implementation-level optimizations that greatly improve the performance in practice. Our evaluation also shows that our protocol reduces the on-chain gas cost between 2-6 times and the transaction size when compared to Hash TimeLock Contract (HTLC) contract, demonstrating the best suitability of our protocol for any blockchain (including those with expressive scripting language support).

II. Solution Overview

In this outline, we mostly focus on our generic protocol, which is compatible with any blockchain, assuming the minimal ability to verify signatures on transactions (for any signature scheme). This lays out the main ideas of our approach and it is the basis for our efficient protocol for the special case of Schnorr/ECDSA signatures.

A. Outline of Our Generic Solution

Assume a setting where a party \(P_0\) owns the coin \(v^{(0)}\) at ledger \(B_0\) and \(P_1\) owns the coin \(v^{(1)}\) at ledger \(B_1\), which they want to securely swap. We assume that the parties have a bootstrapping mechanism (e.g., a forum where they can match their orders and find each other for swapping their coins). A detailed study of this assumption is out of the scope of this paper. The party \(P_0\) could naively transfer \(v^{(0)}\) to \(P_1\) in \(B_0\) with the hope that afterwards \(P_1\) transfers \(v^{(1)}\) to \(P_0\) in \(B_1\). However, the success of such a swap crucially relies on the honesty of the users: Should \(P_1\) not forward the coins, \(P_0\) would incur a loss.

The central challenge that our protocol needs to address is in ensuring atomicity of the swap even in the presence of malicious parties, which is guaranteed by the HTLC-based protocols. Drawing inspiration from that approach, an immediate barrier that we encounter is that the absence of scripting language does not allow us to set “time-outs” on transactions. To avoid users being stuck in deadlocks, we resort to different techniques.

1) Simulating Transaction Timeouts: A timeout \(t\) for a transaction \(tx\) means that the transaction is accepted by the nodes in the network, only after time \(t\) has expired. Typically, this is implemented by expressing \(t\) in terms of a block number and leveraging a timelock script, that is explicitly included in the transaction and checks whether the block number expressed in \(t\) has already been reached. That is, even if the transaction has a valid signature \(\sigma\) but time \(t\) has not expired, the timelock script prevents the transaction from being processed. Our objective will be to simulate such a functionality without using any on-chain script. Our main leverage to achieve this will be verifiable timed signatures (VTS) [34]. A VTS lets a user (or a committer) generate a timed commitment \(C\) of a signature \(\sigma\) on a message \(m\) under a public key \(pk\). The commitment


1. Setup

\[ P_0 : t_x^{(0)}; \text{VTS}_{T_0} \quad P_0 : t_x^{(0)}; \sigma_x^{(0)} \text{ on } B \]

\[ P_1 : t_x^{(1)}; \text{VTS}_{T_1} \quad P_1 : t_x^{(1)}; \sigma_x^{(1)} \text{ on } B \]

2. Lock

\[ P_1 : t_x^{(1)}; \ell_k \quad P_1 : t_x^{(1)}; \sigma_x^{(1)} \text{ at } T_1 \]

\[ P_0 : t_x^{(0)}; \sigma_x^{(0)} \quad P_0 : t_x^{(0)}; \sigma_x^{(0)} \text{ at } T_0 \]

3. Complete

\[ P_0 : t_x^{(0)}; \text{VTS}_{T_0} \quad P_0 : t_x^{(0)}; \sigma_x^{(0)} \text{ on } B \]

\[ P_1 : t_x^{(1)}; \text{VTS}_{T_1} \quad P_1 : t_x^{(1)}; \sigma_x^{(1)} \text{ on } B \]

4. Timeout

\[ P_0 : t_x^{(0)}; \ell_k \quad P_0 : t_x^{(0)}; \sigma_x^{(0)} \text{ at } T_0 \]

\[ P_1 : t_x^{(1)}; \ell_k \quad P_1 : t_x^{(1)}; \sigma_x^{(1)} \text{ at } T_1 \]

Fig. 1: Results of different phases for parties \( P_0 \) and \( P_1 \) in a 1-to-1 swap. In each phase, the first party to receive the output in the phase is written at the top followed by the second party to receive the output in the phase.

C must hide the signature \( \sigma \) for time \( T \) (which can be chosen arbitrarily by the committer). At the same time, the committer also generates a proof \( \pi \) that proves that the commitment \( C \) contains a valid signature \( \sigma_C \). This guarantees that \( \sigma \) can be publicly recovered in time \( T \) by anyone who solves the computational puzzle.

To build some intuition on how to use this tool to simulate a transaction timelock, consider the case of two users (Alice and Bob) sharing an address \( pk_{AB} \) (where each party owns a share of the corresponding secret key). Before sending the funds to \( pk_{AB} \), Alice and Bob jointly sign a “refund” transaction \( t_x^{\text{refnd}} \) that transfers all funds from \( pk_{AB} \) back to the address of Alice, in such a way that only Bob learns the signature. Bob then generates a VTS (using time parameter \( T \)) on this refund signature and provides Alice with the resulting commitment \( C \) and proof \( \pi \). Note that, if after time \( T \) some funds in \( pk_{AB} \) remain unspent, then Alice can immediately refund them by posting the transaction \( t_x^{\text{refnd}} \) together with the valid signature that she learned from \( C \).

2. One-to-One Atomic Swaps: Equipped with a VTS scheme, we first present a simple single-currency one-to-one atomic swap protocol. The protocol consists of four phases, that we describe in the following. Figure 1 shows the parties’ outputs in each phase. For convention, keys, transactions and signatures with \((01)\) are involved in the payment from \( P_0 \) to \( P_1 \) and \((10)\) are involved in the payment from \( P_1 \) to \( P_0 \).

Swap Setup Phase. In the setup phase, the parties transfer their coins to new joint addresses \( pk_{x}^{(01)} \) and \( pk_{x}^{(10)} \) (one for each coin) that both parties together control. More concretely, we have that

\[ sk_{x}^{(01)} = sk_{x}^{(01)} + sk_{x}^{(01)} \text{ and } sk_{x}^{(10)} = sk_{x}^{(10)} + sk_{x}^{(10)} \]

where each party possesses one share of each signing key. However, before transferring the coins to the joint keys, the parties need to ensure that the coins will not be locked forever in the joint address, in case one party goes offline.

As briefly discussed above, this is done by generating two refund transactions \( v_{\text{refnd}} \) of the form

\[ tx^{(0)}_{\text{refnd}} : pk^{(01)}_{x} \xrightarrow{v^{(0)}} P_0 \quad tx^{(10)}_{\text{refnd}} : pk^{(10)}_{x} \xrightarrow{v^{(1)}} P_1. \]

Then \( P_1 \) generates a VTS for the signature on the former refund transaction with time parameter \( T_0 \) and \( P_0 \) generates a VTS for the signature on the latter refund transaction with time parameter \( T_1 \). The time parameters are set such that \( T_0 = T_1 + \Delta \), where \( \Delta \) is a conservatively chosen delay parameter. This gap \( \Delta \) prevents race conditions and ensures that an adversarial \( P_0 \) cannot wait until the last moment to retrieve the coin \( v^{(1)} \), in the hope that the transaction swapping \( v^{(0)} \) expires in the meantime, effectively stealing coins from \( P_1 \). Once both VTS are verified, the parties proceed by transferring the coins to the shared addresses by signing the freeze transactions of the form

\[ tx^{(0)}_{\text{swp}} : P_0 \xrightarrow{v^{(0)}} pk^{(01)}_{x} \quad tx^{(1)}_{\text{swp}} : P_1 \xrightarrow{v^{(1)}} pk^{(10)}_{x}. \]

Swap Lock Phase. After a successful setup phase, parties generate payment “locks” on transactions that spend from the joint keys. Specifically, they define the following swap transactions

\[ tx^{(1)}_{\text{swp}} : pk^{(01)}_{x} \xrightarrow{v^{(0)}} P_1 \quad tx^{(0)}_{\text{swp}} : pk^{(10)}_{x} \xrightarrow{v^{(1)}} P_0. \]

Since the secret keys of the joint addresses are secret shared among the parties, a natural idea is to compute the respective signatures using a secure 2-party computation (2PC) protocol. However, this naive attempt leads to an insecure scheme: The 2PC protocol does not guarantee any form of guaranteed output delivery \( [1] \) so nothing prevents \( P_0 \) from going offline after receiving a valid signature on \( tx^{(0)}_{\text{swp}} \). Instead, the parties first compute via a 2PC, a “locked” version of a signature on \( tx^{(1)}_{\text{swp}} \), i.e.,

\[ \ell_k := \sigma_{x}^{(1)} \oplus H(\sigma_{x}^{(0)}) \]

where \( H \) is a hash function (modelled as a random oracle \( [2] \)). Observe that the \( \oplus \) operation mimics the one-time pad encryption with \( H(\sigma_{x}^{(0)}) \) as the encryption key and \( \sigma_{x}^{(1)} \) as the encrypted message. Note that at this point, neither party knows the valid signature \( \sigma_{x}^{(1)} \) since it is masked by the output of \( H \). However, now we know that if \( P_0 \) were to somehow publish the signature on \( tx^{(0)}_{\text{swp}} \), then \( P_1 \) would immediately learn a valid signature on \( tx^{(1)}_{\text{swp}} \), by recomputing \( \ell_k \) and removing the mask. Only after this “lock” step is successfully completed, both parties engage in a 2PC to jointly compute a signature on \( tx^{(0)}_{\text{swp}} \), which is now safe to reveal.

Swap Complete Phase. After a successful lock phase, each party can post the respective transaction-signature pair on the

1. We assume the parties come to an agreement on the transaction fee for all swap related transactions.
2. The standard security notion for 2PC is security with aborts, which allows an adversarial party to learn the output of the computation while preventing honest parties to do so.
3. A standard model instantiation is possible if we let \( H \) be a sufficiently stretching leakage resilient pseudorandom generator that can be constructed from Extremely Lossy Functions [2].
4. We assume the signing algorithm is deterministic.
blockchain. As mentioned before, while \( P_0 \) can simply read the signature in plain, \( P_1 \) recovers his signature by computing
\[
\sigma_{\text{swap}}^{(1)} = \ell k \oplus H \left( \sigma_{\text{swap}}^{(0)} \right).
\]
This guarantees the atomicity of the swap: \( P_0 \) cannot obtain \( v^{(1)} \) without \( P_1 \) transferring \( v^{(0)} \), and vice versa.

**Swap Timeout Phase.** If at any point in the setup or lock phase either of the party goes offline, then party \( P_b \) can recover her coins from the joint address using the time-locked signature \( \sigma_{\text{swap}}^{(b)} \), which she eventually learns by opening the VTS. Here, as shown in Figure 1, party \( P_1 \) can recover her coins first as \( T_1 < T_0 \). Thus, the parties are guaranteed to not lose their coins if any participant goes offline for extended periods of time.

3) **Multi-Asset Atomic Swaps:** The most general (and realistic) setting that we consider is when \( P_0 \) holds \( n \) coins (possibly from different currencies) and \( P_1 \) holds \( n \) coins (from a possibly disjoint set of currencies), which they want to swap atomically. This situation is presented in Figure 2.

![Figure 2: Setting for a n-to-n swap.](image)

Before delving into the details on how to modify our vanilla protocol to support multi-asset swaps, let us pause and discuss what kind of security we (intuitively) expect from such a protocol. On \( P_0 \) side, we want to ensure that \( P_1 \) cannot claim (or transfer) any of \( P_0 \)'s coin before \( P_0 \) holds signatures on transactions for all of \( P_1 \)'s coin. \( P_0 \) can obtain \( P_1 \)'s coins by posting the transactions and the corresponding signatures. The coins are then considered transacted to \( P_0 \). Conversely, we want to guarantee that if any of \( P_1 \)'s coin is transacted to \( P_0 \), then \( P_1 \) can simultaneously learn signatures on all transactions that spend \( P_0 \)'s coins to \( P_1 \).

To do this, we proceed by viewing the swap as a collection of \( n \) separate n-to-1 atomic swaps. Intuitively, this forces \( P_0 \) to wait until the end of all \( n \) iteration before publishing any signature, as otherwise \( P_1 \) would be able to claim all of \( P_0 \)'s coins. To implement a single n-to-1 atomic swap we need to be able to generate a single \( \ell k \) that simultaneously locks \( n \) signatures \( (\sigma_1, \ldots, \sigma_n) \) and condition their release on a single signature \( \hat{\sigma} \). We realize this extending our one-to-one locking mechanism and computing \( \ell k \) as
\[
\ell k = (\sigma_1 \oplus H(1||\hat{\sigma}), \ldots, \sigma_n \oplus H(n||\hat{\sigma})).
\]
Since we model \( H \) as a random oracle, this allows us to stretch the randomness extracted from \( \hat{\sigma} \) as much as we need. Note that once \( \hat{\sigma} \) is revealed, all of the other signatures are simultaneously unmasked. Running \( n \) copies of this modified protocol once for each of \( P_0 \)'s coins, allows us to achieve a secure \( n \)-to-\( n \) multi-asset swap.

**Shielded Addresses of Zcash.** Our approach can also be extended to atomic swaps of shielded addresses of Zcash [26], with the aid of 2PC protocols for generating SNARK proofs [43]. Zcash provides two types of addresses: shielded (that retain privacy) and unshielded (that are transparent like Bitcoin addresses), and prior works only support atomic swap for the unshielded addresses of Zcash [44].

**Trade-off with HTLC based Solutions.** Note that in HTLC based solutions (described in Section E-A) we implicitly assume the adversary cannot force parties to go offline for too long. In our VTS based solution, we implicitly assume that an adversary cannot force a party not to perform local computation for too long, for example by cutting off the computation power of the party for extended periods of time. An advantage of our approach is that a party only have to spend computational effort to open their VTS if the other party does not respond quickly. In other words, opening of VTS is only a deterrent mechanism in case the parties do not instantaneously complete the swap. On the other hand, in the HTLC approach, parties have to pay the transaction fee/gas cost associated with the HTLC invariably.

**B. The Case of Schnorr/ECDSA Signatures**

While our generic protocol satisfies all desired properties simultaneously (non-custodial, universal, multi-asset), it falls short in achieving concrete computational efficiency: The usage of generic tools (such as general-purpose 2PC) might make the cost of running such a protocol prohibitive for some users. However, we can use the general blueprint established by the approach to develop efficient protocols for specific signature schemes.

In this regard, we revisit our general framework to dramatically improve its efficiency for the case where the signature scheme verified by the ledgers is either Schnorr or ECDSA. While this is a downgrade for the generality of our approach, we remark that virtually all major cryptocurrencies rely on Schnorr or ECDSA signatures, which allows us to remain compatible with the vast majority of coins. In the following, we give a cursory outlook at the aspects that we improve and we refer the reader to Section V for a precise description of the protocol.

**More Efficient VTS.** Time-lock encryption [45] in principle lets us encrypt messages to the future, however there is no known practically efficient instantiation of their proposal. Practically efficient VTS constructions tailored for Schnorr/ECDSA signatures were presented in [34]. However, we observe that we can further improve the efficiency by committing to the whole signature key corresponding to public keys and (instead of the signature), since the public keys are used only once. Recall that in Schnorr/ECDSA signatures, secret keys are integers \( x \) and public keys are of the form \( (G, xG) \), where \( G \) is the generator of some cyclic group of prime order. Thus, instead of VTS, we can generate commitments to a secret key \( x \) via a verifiable timed discrete logarithm (VTD) scheme [34]. Concretely, the
VTD construction from [34] is far more efficient than the VTS constructions for Schnorr/ECDSA signatures in terms of commitment generation and verification, as the committer no longer needs to prove that the committed signature is valid. Instead, the committer needs to prove that the committed value is a valid discrete logarithm of some known element, which is a simpler algebraic statement. This allows us to significantly boost the efficiency of our setup phase. However, in our protocol we need to ensure that the parties generate valid VTD commitments to their shares of the secret key $x$ as neither party has access to the whole secret key at the beginning of the protocol. We defer the details to Section V-D.

Avoiding General-Purpose 2PC. Instead of relying on a general-purpose 2PC protocol to compute a "locked" signature, we leverage atomic multi-hop locks [46] originally introduced in the context of payment channel networks. On a high level, their 2PC protocol lets party $P_0$ and $P_1$ jointly compute a pre-signature $\tilde{\sigma}$ on a message $m$ under the joint public key $pk$, with respect to an instance $Y$ of a hard relation $R$. The property of interest here is that $\tilde{\sigma}$ is not a valid signature on the message $m$, but can be adapted into a valid signature $\sigma$ with the knowledge of a witness $y$, such that $(Y, y) \in R$. Additionally, given the pre-signature $\tilde{\sigma}$ and the valid signature $\sigma$, one can efficiently extract the witness $y$ for the instance $Y$. A first approach would be to use their protocol instead of a 2PC, however it turns out that we can do even better by tweaking the composition of different protocol instances.

Reducing the Number of Iterations. We show how to reduce the number of iterations for the execution of lock protocol from $\tilde{n} \cdot (n + 1)$ to an additive $n + \tilde{n}$. The overall idea to do this, is to let the parties engage in the 2PC protocol from [46] for each swap transaction ($n + \tilde{n}$ in total) exactly once, generating a pre-signature on the transaction under the corresponding joint public key. Importantly, all the pre-signatures are generated with respect to the same instance $Y$ of a hard relation $R$, where one of the two parties (in our case $P_0$) knows the corresponding witness $y$.

Care must be taken to ensure an ordering such that party $P_1$ first obtains all the pre-signatures on the transactions from $P_0$ to $P_1$, before party $P_0$ obtains any pre-signature on any transaction from $P_1$ to $P_0$. This is to prevent $P_0$ from completing any pre-signature, since he knows the witness $y$. Once all the $n + \tilde{n}$ pre-signatures are generated and available with both parties, the swap can be completed by $P_0$, using the knowledge of $y$. On the other hand, $P_1$ can extract the witness $y$ from any of the signatures posted by $P_0$ and therefore turn his pre-signatures into valid ones. This completes the swap.

Optimisations. We have two possible optimisations to reduce computational work for both parties in terms of opening the VTD commitments. In the first optimisation, party $P_0$ instead of computing on $n$ VTD commitments, can homomorphically combine those commitments and work on opening only a single VTD commitment. Similar optimisation is possible for party $P_1$ also. The technique is called batching VTD commitments [48] and we leave the details to the full version.

In Section V-D we implement the second optimization where instead of $n + \tilde{n}$ VTD commitments, only 2 VTD commitments are generated. Now the parties solve one commitment each instead of $n$ and $\tilde{n}$ like before. To do this, we exploit the key structure in Schnorr/ECDSA signature schemes, where opening one VTD commitment helps $P_0$ learn $n$ secret keys and vice versa for $P_1$. The parties additionally need to execute a joint coin tossing protocol $n + \tilde{n}$ times which is significantly cheaper than computing on $n + \tilde{n}$ VTD commitments.

Optimising Number of Swaps. With the above optimisation, the parties are still required to perform persistent computation to open their respective VTD commitments. This could limit the number of coins ($n$ and $\tilde{n}$) that the parties may want to swap simultaneously with many other parties. However, the persistent computation of opening a VTD commitment can be securely outsourced to a decentralized service [49] at a market determined cost. This relieves both parties of any potentially heavy computation related to VTD opening. As a consequence, provided the parties have enough funds to outsource VTD openings, the number of coin swaps of a party is no longer limited by its own computational power.

Optimistic Efficiency. More importantly, in the optimistic case (i.e., where both parties $P_0$ and $P_1$ are honest and remain online until the end of the protocol) our protocol terminates instantly, without either party having to invest computational resources into opening the VTD commitments. In practice, we expect that the presence of VTD commitments will mostly function as a deterrent for people not to misbehave and the parties will not have to open them, except for rare cases. Therefore, in the pessimistic case where parties need to use the deterrent mechanism, parties need to dedicate one CPU core for solving a certain number of batched VTD commitments.

Extensions. Finally, we show (Section V-E) how to handle a mixture of Schnorr/ECDSA signatures even when implemented over different curves (that define groups of different orders).
adversary $S$ we write $\text{EXEC}_{\mathcal{F},S,\mathcal{E}}$ to denote the distribution ensemble of the ideal world execution.

**Definition 1 (Universal Composability):** A protocol $\tau$ UC-realizes an ideal functionality $\mathcal{F}$ if for any PPT adversary $A$ there exists a simulator $\mathcal{S}$ such that for any environment $\mathcal{E}$ the ensembles $\text{EXEC}_{\tau,A,\mathcal{E}}$ and $\text{EXEC}_{\mathcal{F},S,\mathcal{E}}$ are computationally indistinguishable.

**Digital Signatures.** A digital signature scheme DS, formally, has a key generation algorithm $\text{KGen}(1^n)$ that takes the security parameter $1^n$ and outputs the public/secret key pair $(pk, sk)$, a signing algorithm $\text{Sign}(sk, m)$ inputs a secret key and a message $m \in \{0,1\}^*$ and outputs a signature $\sigma$, and a verification algorithm $\text{Vf}(pk, m, \sigma)$ outputs $1$ if $\sigma$ is a valid signature on $m$ under the public key $pk$; and outputs $0$ otherwise. We require the standard notion unforgeability for the signature scheme $\text{[52]}$. A stronger notion of strong unforgeability for the signature scheme was shown to be equivalent to the UC formulation of security $\text{[53]}$.

**Hard Relations.** We recall the notion of a hard relation $R$ with statement/witness pairs $(Y, y)$. We denote by $\mathcal{L}_R$ the associated language defined as $\mathcal{L}_R := \{Y | \exists y \ s.t. \ (Y, y) \in R\}$. The relation is called a hard relation if the following holds: (i) There exists a PPT sampling algorithm $\text{GenR}(1^n)$ that outputs a statement/witness pair $(Y, y) \in R$; (ii) The relation is poly-time decidable; (iii) For all PPT adversaries $\mathcal{A}$ the probability of $\mathcal{A}$ on input $Y$ outputting a witness $y$ is negligible.

**2-Party Computation.** The aim of a secure 2-party computation (2PC) protocol is for the two participating users $P_0$ and $P_1$ to securely compute some function $f$ over their private inputs $x_0$ and $x_1$, respectively. Apart from correctness of output, we require privacy that states that the only information learned by the parties in the computation is that specified by the function output. Note that we require the standard security with aborts, where the adversary can decide whether the honest party will receive the output of the computation or not. I.e., we do not assume any form of fairness or guaranteed output delivery. For a comprehensive treatment of the formal UC definition we refer the reader to $\text{[31]}$. As standard in the UC settings, we work in the static corruption model, where the adversary declares which party will be corrupted ahead of time.

**Synchrony and Communication.** We assume synchronous communication between users, where the execution of the protocol happens in rounds. We model this via an ideal functionality $\mathcal{F}_{\text{clock}}$ $\text{[54], [55]}$, where all honest parties are required to indicate that they are ready to proceed to the next round before the clock proceeds. The clock functionality that we consider is fully described in $\text{[51]}$. This means that all entities are always aware of the given round. Users can abort a session at any given round by sending a distinguished message (abort). We also assume secure message transmission channels between users modelled by the ideal functionality $\mathcal{F}_{\text{smt}}$.

**Blockchain.** We assume the existence of an ideal ledger (blockchain) functionality $\mathcal{B}$ (just as in $\text{[9], [46], [56]}$) that maintains the list of coins currently associated with each address and that we model as a trusted append-only bulletin board. For simplicity of notation, we make use of $\mathcal{B}$ for the chain of all the currencies involved in the swap. The corresponding ideal functionality $\mathcal{F}_{\mathcal{B}}$ maintains the ledger $\mathcal{B}$ locally and updates it according to the transactions between users. More precisely, it offers an interface $\text{Post}(id, A, B, v)$ to transfer $v$ coins during a session with identifier id, from address $A$ (with associated verification key $pk_A$), to address $B$ (with associated verification key $pk_B$), if provided with $sk_A$. Users may use the interface to transact among themselves. An additional interface $\text{Register}$, is for users to register their address $A$ along with the associated verification key $pk_A$ and a value $v$, which is stored in the ledger.

At any point in the execution, any user $U$ can send a distinguished message $\text{Read}$ to $\mathcal{F}_{\mathcal{B}}$, who sends the whole transcript of $\mathcal{B}$ to $U$. We refer the reader to $\text{[56]}$ for a formal definition of this functionality.

**IV. Definitions for Atomic Swaps**

In the following we motivate and present the security definition of atomic swaps in the form of an ideal functionality. The main property the ideal functionality must guarantee is atomicity: Either both users interested in the swap successfully swap their coins, or the swap fails and everyone is back to their initial holdings. For the more general case of $n$-to-$n$ swap between users $U_0$ and $U_1$, the atomicity notion we set out to achieve is: if at least one of the $n$ coins is moved to $U_0$, it is possible for $U_1$ (if has not aborted) to swap all of the $n$ coins. This notion ensures if $U_0$ initiates the swap, $U_1$ does not lose coins, and if $U_0$ aborts the swap before initiation, the swap is aborted and she does not lose coins.

**Discussion.** Before delving into the description of the ideal functionality, we discuss why existing definitions fall short in capturing the security of universal atomic swaps.

The notion of cryptographic fairness (the analogue of output atomicity in multi-party computation) has been studied extensively in the literature, even in the context of blockchains $\text{[57], [58]}$. The UC modelling of such a functionality guarantees that honest participants in the computation receive their output if the adversary also receives its. This notion intuitively seems to capture the security for an atomic swap, i.e., if the adversary gets an honest party’s digital good, the honest party gets the adversary’s digital good. However, there are subtle aspects that make it insufficient to model security in our setting.

To exemplify the problem, consider the scenario where the adversary has a coin $v_a$ at address $A$ and the honest party has a coin $v_b$ at address $H$ that they want to swap. Both parties register the addresses $A$ and $H$ with the ideal functionality. To complete the swap, the coin $v_a$ must be transferred from $A$ to an address for which the honest party knows the authentication key, and vice versa for the coin $v_b$. A naive idea to implement this using the fair exchange functionality would be for the parties to simply exchange the secret keys of their addresses. Unfortunately, this attempt turns out to lead to a completely insecure protocol: During the execution of the fair exchange, the attacker might quickly move all the coins from $A$ to some other address, before the execution of the protocol is completed. In the end, the honest party will end up with the secret key
of the empty address A, whereas the attacker will learn the secret key for H, effectively stealing coins.

Prior works [6] address this issue by leveraging a special coin freezing functionality of the blockchain, thus preventing the above described attack. However, the modelling of [6] implicitly relies on special scripting capabilities of the blockchain to implement the coin freezing function. Since we are interested in settings, where the blockchain may not offer such scripts, we have to work around this issue. Our functionality (largely inspired by [6]) will implement coin freezing only by using the standard interfaces offered by any blockchain (i.e., address registration and transaction posting).

A. Ideal Functionality

We now formally describe our ideal functionality $F_{\text{swap}}$ (Figure 3) for atomic swaps of coins that circumvent the above problem. $F_{\text{swap}}$ interacts with the two users $U_0, U_1$ and the simulator $S$. In the first round, user $U_1$ initiates a swap with the message $(\text{swap}_1, id, V, V, PK, PK, SK)$ that specifies the coins $V := (v_1, \ldots, v_n)$ (owned by $U_0$) and $\tilde{V} := (\tilde{v}_1, \ldots, \tilde{v}_n)$ (owned by $U_1$) that are to be swapped. User $U_1$ also gives his authentication keys SK for the addresses PK corresponding to the coins in $V$.

In the second round, user $U_0$ also acknowledges the swap initiation by giving his authentication keys SK for the addresses PK corresponding to the coins $V$. The ideal functionality uses the Freeze subroutine to transfer each of those coins to an id-specific address that is controlled by the functionality. The functionality can do this because it knows the authentication keys of each of those coins and can interact with $F_B$ to transfer coins to these functionality controlled keys. If some transaction fails, then the functionality returns the coins frozen so far to the original owner.

In the third round, if user $U_0$ aborts, all the coins are unfrozen by the ideal functionality and transferred back to the respective users. On the other hand, if the functionality receives buy for some index set $J \subseteq \{n\}$ of coins from $U_0$, then it uses the Transfer subroutine to transfer the coins $\tilde{v}_j$ for $j \in J$ from its control to user $U_0$.

In the last round, if user $U_1$ aborts, all the frozen coins of user $U_0$ (from $V$) are unfrozen by the ideal functionality and transferred back to user $U_0$. On the other hand, all coins of $U_1$ that remained frozen (coins not in $J$) are unfrozen and transferred back to $U_1$. Otherwise, if user $U_1$ responds with buy for some index $I \subseteq \{n\}$, the functionality uses the Transfer subroutine to transfer the coins $v_i$ for $i \in I$ to user $U_1$. Finally, all coins still controlled by the functionality are unfrozen and transferred to their initial owners.

Atomicity. The functionality ensures that if user $U_0$ in round 3 aborts the swap, all the coins are refunded to both parties. Instead if $U_0$ initiates a swap for any subset of $\tilde{n}$ coins in round 3, user $U_1$ is allowed to complete the swap in round 4 for all $n$ coins. Since no party can unilaterally transfer a coin (as it is locked with functionality), the functionality guarantees atomicity for the $n$-to-$\tilde{n}$ swap.

The ideal functionality $F_{\text{swap}}$ (in session id) interacts with users $U_0$ and $U_1$ and the ideal adversary $S$.

(Round 1) Upon receiving $(\text{swap}_1, id, V, V, PK, PK, SK)$ from $U_1$, where $V := (v_1, \ldots, v_n) \in \mathbb{N}$, $\tilde{V} := (\tilde{v}_1, \ldots, \tilde{v}_n) \in \mathbb{N}$, $PK := (pk_1, \ldots, pk_n)$, $\tilde{PK} := (\tilde{pk}_1, \ldots, \tilde{pk}_n)$, $SK := (sk_1, \ldots, sk_n)$. Send $(\text{swap}_1, id, V, V, PK, PK, U_1)$ to $S$ and store the input tuple.

(Round 2) Upon receiving $(\text{swap}_2, id, V, V, PK, PK, SK)$ from user $U_0$, where $\tilde{SK} := (\tilde{sk}_1, \ldots, \tilde{sk}_n)$, call the subroutines $\text{Freeze}(id, v_i, pk_i, sk_i, i)$ and $\text{Freeze}(id, \tilde{v}_j, \tilde{pk}_j, sk_j, j)$ for all $i \in \{n\}$ and $j \in \{\tilde{n}\}$. If all of the invocations are successful, send $(\text{swap}_2, id, V, V, PK, PK, U_0)$ to $S$, store the input tuple and proceed to round 3. Otherwise, revert the coin freezing by invoking the corresponding subroutines $\text{UnFreeze}(id, v_i, pk_i, i)$ and $\text{UnFreeze}(id, \tilde{v}_j, \tilde{pk}_j, j)$.

(Round 3) Upon receiving $(\text{abort}, id)$ from the user $U_0$, revert the freezing by calling $\text{UnFreeze}(id, v_i, pk_i, i)$ and $\text{UnFreeze}(id, \tilde{v}_j, \tilde{pk}_j, j)$ for all $i \in \{n\}$ and $j \in \{\tilde{n}\}$. Otherwise, upon receiving $(\text{buy}, id, J, pk)$, for some $J \subseteq \{\tilde{n}\}$, call the subroutine $\text{Transfer}(id, J, pk)$. Store $J$ and proceed to round 4.

(Round 4) Upon receiving $(\text{abort}, id)$ from the user $U_1$, call $\text{UnFreeze}(id, v_i, pk_i, i)$ for all $i \in \{n\}$ and $\text{UnFreeze}(id, \tilde{v}_j, \tilde{pk}_j, j)$ for all $j \in J$ and terminate. Otherwise, upon receiver $(\text{buy}, id, I, pk)$ from $U_1$ do the following:

- If $I \neq \emptyset$ call $\text{Transfer}(id, I, pk)$, then
  - $\text{UnFreeze}(id, \tilde{v}_j, \tilde{pk}_j, j)$ for all $j \in J$ and
  - $\text{UnFreeze}(id, v_i, pk_i, i)$ for all $i \notin I$.
- If $I = \emptyset$ call $\text{UnFreeze}(id, v_i, pk_i, i)$ and $\text{UnFreeze}(id, \tilde{v}_j, \tilde{pk}_j, j)$ for all $i \in \{n\}$ and $j \notin J$.

Notify $S$ of the outcome of the operation.

Description of the subroutines

$\text{Freeze}$: On input a tuple $(id, v, pk, sk, i)$, transfer $v$ coins from the address specified by $pk$ (using $sk$) to some id-specific address $pk_id$ controlled by the ideal functionality via $\text{Post}(id, pk_id, pk, v)$. The function is successful if the transaction is accepted by $F_B$.

$\text{UnFreeze}$: On input a tuple $(id, v, pk, pk_id)$ transfer back the $v$ frozen coins to the corresponding public key $pk$, via $\text{Post}(id, pk_id, pk, v)$.

$\text{Transfer}$: On input a tuple $(id, I, pk)$, transfer all frozen coins corresponding to the index set $I$ to the public key $pk$, via $\text{Post}(id, pk_id, pk, \sum_{i \in I} v_i)$.

Fig. 3: Ideal functionality $F_{\text{swap}}$ for fair swap of coins

Fungibility. Notice that the functionality only makes standard Transfer calls to the blockchain $B$. No additional information is present, besides verification keys and transected values. Thus these calls are syntactically identical to regular transactions.
V. Atomic Swaps from Adaptor Signatures

Here we present an efficient atomic swap protocol for n-to-n swap of coins between two users P₀ and P₁, under the condition that transactions are signed using Schnorr or ECDSA signatures. The fundamental building blocks of our protocol are adaptor signatures and verifiable timed dlog (VTD), both of which are defined below.

A. Adaptor Signature

Adaptor signatures [47] let users generate a pre-signature on a message m which by itself is not a valid signature, but can later be adapted into a valid signature if the user knows some secret value. The formal definition of adaptor signatures is given below.

Definition 2 (Adaptor Signatures): An adaptor signature scheme ΠAS w.r.t. a hard relation R and a signature scheme ΠDS = (KGen, Sign, Vf) consists of algorithms (pSign, Adap, pVf, Ext) defined as:

σ ← pSign(sk, m, Y): The pre-sign algorithm takes as input a secret key sk, message m ∈ {0, 1}*, and statement Y ∈ LR, outputs a pre-signature σ.

0/1 ← pVf(pk, m, Y, σ): The pre-verify algorithm takes as input a public key pk, message m ∈ {0, 1}*, statement Y ∈ LR and pre-signature σ, outputs a bit b.

σ ← Adap(σ, y): The adapt algorithm takes as input a pre-signature σ and witness y, outputs a signature σ.

y ← Ext(σ, Y): The extract algorithm takes as input a signature σ, pre-signature σ and statement Y ∈ LR, outputs a witness y such that (Y, y) ∈ R, or ⊥.

In terms of security, we want (i) unforgeability that is similar to the unforgeability of standard signature schemes, except that we additionally want that producing a forgery σ for some message m is hard even given a pre-signature σ on m, with respect to some uniformly sampled instance Y ∈ LR. We then want (ii) witness extractability that guarantees that given a valid signature σ and a pre-signature σ for a message m and an instance Y, one can efficiently extract a witness y for Y. Finally, we want (iii) pre-signature adaptability that guarantees that any valid pre-signature σ generated with respect to an instance Y, can be adapted into a valid signature σ if given the witness y for the instance Y.

In this work, we consider the constructions proposed in [47] of adaptor signatures where the signature scheme ΠDS is of Schnorr or ECDSA. The hard relation R used in their constructions is that of the discrete log relation, where the language is defined as: LR : = {H : ∃x ∈ Zq, s.t. H = Gx}, where (G, G, q) are the group description, its generator, and its order, respectively.

B. Verifiable Timed DLog

A Verifiable timed dlog [34] is defined with respect to a group G of order q and a generator G. Here, a committer generates a timed commitment of timing hardness T of a value x ∈ Zq such that H = Gx where H is public and x is referred to as the dlog (discrete logarithm) of H (wrt. G). The verifier checks the well-formedness of the timed commitment and can learn the value x by force opening the commitment in time T. The formal definition is as follows.

Definition 3 (Verifiable Timed Dlog): A VTD for the group G with generator G and order q is a tuple of four algorithms (Commit, Verify, Open, ForceOp) where:

(C, π) ← Commit(x, T): the commit algorithm (randomized) takes as input a discrete log value x ∈ Zq (generated using KGen(1λ)) and a hiding time T and outputs a commitment C and a proof π.

0/1 ← Verify(H, C, π): the verify algorithm takes as input a group element H, a commitment C of hardness T and a proof π and outputs 1 if and only if, the value x embedded in C satisfies H = Gx. Otherwise it outputs 0.

(x, r) ← Open(C): the open algorithm (run by committer) takes as input a commitment C and outputs the committed value x and the randomness r used in generating C.

x ← ForceOp(C): the force open algorithm takes as input the commitment C and outputs a discrete log value x.

The security requirements for a VTD scheme are that of (i) soundness where the user is convinced that, given C, the ForceOp algorithm will produce the commited dlog. value x in time T and (ii) privacy, where all PRAM algorithms whose running time is at most t (where t < T) succeed in extracting x from the commitment C and π with at most negligible probability. Formal definitions are in the full version.

The construction of VTD from [34] makes use of time-lock puzzles [59], a primitive that lets a committer embed a secret inside a puzzle with the guarantee that the puzzle can be opened only after time T. Specifically, the committer embeds the dlog. value x inside a time-lock puzzle and uses a special NIZK proof to prove its validity. Specifically, the NIZK proves that the time-lock puzzle can be solved in time T and what is embedded inside satisfies the equation H = Gx. They construct such an efficient NIZK proof using cut-and-choose techniques, Shamir secret sharing [60] and homomorphic time-lock puzzles [61].

C. 2-Party Protocols

Joint Key Generation. We require that two parties P₀ and P₁ jointly generate (ECDSA and Schnorr) keys. We denote this interactive protocol by I(SIG)JKGen, where SIG = Schnorr or SIG = ECDSA. The joint key generation protocol I(SIG)JKGen takes as input the security parameter 1λ and the group description (G, G, q) as input from both parties. The protocol outputs the public key pk to both parties and outputs the secret key share sk₀ to P₀ and sk₁ to P₁. We efficiently instantiate I(Schnorr)JKGen with the interactive protocol from [62], where in the end we have pk = Gsk (sk is the secret key), and sk = (sk₀ + sk₁). On the other hand, we instantiate I(ECDSA)JKGen with the protocol from [63] where in the end we have pk = Gsk (sk is the secret key), and sk = (sk₀ · sk₁).

Joint Adaptor Signing. We require the parties P₀ and P₁ to jointly generate adaptor signatures on messages with respect to both parties and outputs the secret key share sk₀ to P₀ and sk₁ to P₁. We efficiently instantiate I(Schnorr)JKGen with the interactive protocol from [62], where in the end we have pk = Gsk (sk is the secret key), and sk = (sk₀ + sk₁).
to an instance $Y$ of a hard (dlog) relation $R$, for a joint public key $pk$. We denote this interactive protocol by $\Pi_{\text{AdpSG}}^{\text{SIG}}$ which takes as common input a message $m$, and an instance $Y$ of the hard relation $R$. As private input, party $P_0$ and $P_1$ input their secret key share $sk_0$ and $sk_1$, respectively. Here $sk_0$ and $sk_1$ are shares of the secret key $sk$ whose corresponding public key is $pk$. The output of the protocol for both parties is the pre-signature $\tilde{\sigma}$, such that it holds that $\Pi_{\text{AdpSG}}^{\text{SIG}}.p\text{Vf}(pk, m, Y; \tilde{\sigma}) = 1$. We efficiently instantiate $\Pi_{\text{AdpSG}}^{\text{SIG}}$ for $\text{SIG} = \text{Schnorr}$ and $\text{SIG} = \text{ECDSA}$ with the protocols from $[46]$.

D. Our Protocol

We now describe our atomic swap protocol for transaction schemes based on Schnorr and ECDSA signature schemes. In Figure 5 and Figure 6, we present the protocol for $n$-to-$\tilde{n}$ atomic swaps that, is, coins $v_i^{(0)}$ for $i \in [n]$ of party $P_0$ are swapped with coins $v_k^{(1)}$ for $k \in [\tilde{n}]$ belonging to party $P_1$.

**On Choosing the Timing Hardness $T_0$ and $T_1$.** The parties shall make conservative estimates of each other’s computational power in force opening the VTD commitments. This is to prevent scenarios where $P_0$ or $P_1$ with a powerful machine force opens its VTD commitments earlier than expected in terms of real time. The party could potentially steal the coins of the other party during the swap lock or swap complete phase. In particular, the parties must ensure that $\Delta$ (such that $T_0 = T_1 + \Delta$) is large enough such that it tolerates the time differences to open the VTD commitments.

**Security Analysis.** In the below theorem, we precisely state the security of the above atomic swap protocol. For a formal proof, we refer the reader to Appendix C.

**Theorem 5.1:** Let $\text{SIG} \in \{\text{Schnorr, ECDSA}\}$ and let $\Pi_{\text{AdpSG}}^{\text{SIG}} := (\text{pSign}, \text{pVf}, \text{Adapt}, \text{Ext})$ be a secure adaptor signature scheme with respect to a strongly unforgeable signature scheme $\Pi_{\text{DS}}^{\text{SIG}} := (\text{KGen}, \text{Sign}, \text{Vf})$ and a hard dlog relation $R$. Let $\Gamma_{\text{JKGen}}, \Gamma_{\text{AdpSG}}$ be UC secure 2PC protocols for computing $\text{JKGen}$ and $\text{pSign}$, respectively. Let $\Pi_{\text{VTD}}$ be a timed private and sound verifiable timed dlog scheme. Then the atomic swap protocol described in Figure 5 with access to $(\mathcal{F}_B, \mathcal{F}_\text{amt}, \mathcal{F}_\text{clock})$, UC-realizes the functionality $\mathcal{F}_{\text{swap}}$.

**E. Cross-Curve Swaps**

The protocol described in Figure 5 assumes that the Schnorr/ECDSA signatures used for spending different coins are implemented on the same curve. We can easily extend our protocol to a case where all of the $n + \tilde{n}$ coins are from transaction schemes using Schnorr/ECDSA signatures but are implemented on different curves, possibly over groups of different orders. In such a scenario we have for $i \in [n]$, coin $v_i^{(0)}$ is in a currency whose transaction scheme uses $\text{SIG} \in \{\text{Schnorr, ECDSA}\}$ implemented in the group $\mathbb{G}_i$ with generator $G_i$ and order $q_i$. Similarly, for $k \in [\tilde{n}]$, coin $v_k^{(1)}$ is in a currency whose transaction scheme uses $\text{SIG} \in \{\text{Schnorr, ECDSA}\}$ implemented in the group $\mathbb{G}_k$ with generator $G_k$ and order $q_k$.

It is not hard to see that the only bridge that connects the different signature schemes is the hard instance $Y$ and as soon as the signature schemes are defined over different groups, it becomes unclear how to sample $Y$. The natural solution for this problem is to define a different $Y_i$ for each signature $\text{SIG} = G^y_i$ (or $\tilde{Y}_k = G^\tilde{y}_k$, respectively), where $G_i$ is the generator of the $i$-th group. At this point, we can compute the $i$-th pre-signature with respect to the corresponding instance $Y_i$, instead of using a fixed $Y$. Since the witness $y$ (i.e., the discrete logarithm) is identical for all $Y_i$, correctness is preserved. However, nothing prevents $P_0$ from diverging from the protocol, which is the reason why we also need to include a Non-Interactive Zero Knowledge (NIZK) $[64]$ proof to certify that all instances $(Y_1, \ldots, Y_n, \tilde{Y}_1, \ldots, \tilde{Y}_{\tilde{n}})$ have the same discrete logarithm. Fortunately, this NIZK can be efficiently instantiated extending the construction from $[65]$ to support proving of equality of discrete logarithm in $n + \tilde{n}$ groups. The rest of the protocol is the same as in Figure 5.

We now argue the security of the protocol with this modification in place. The analysis largely follows along the same lines of Theorem 5.1 with the exception that we can no longer reduce to the standard dlog problem, since we are now given access to many instances (across different groups) with the same discrete logarithm. While this problem has been (implicitly) used before, to the best of our knowledge, it has never been formalized. We call this problem the cross-group short discrete logarithm problem and define it below.

**Definition 4 (n-Cross-Group Short Discrete Logarithm):** Consider $\{G_i, G_i, q_i\}_{i \in [n]}$ be $n$ uniformly sampled groups of $\lambda$-bits orders $q_i$. Let $q_i = \min(q_1, \ldots, q_n)$. Let $y \leftarrow \mathbb{Z}_{q_i}$. Then for all PPT adversaries $A$ there exists a negligible function $\negl$ such that

$$\Pr[A(\{G_i, G_i, q_i, G_i^{y_i}\}_{i \in [n]} = y) = \negl(\lambda)].$$

Observe that $y$ cannot be sampled to be perfectly uniform over all groups ($\mathbb{Z}_{q_1}, \ldots, \mathbb{Z}_{q_n}$), since the groups may have different orders, therefore we sample it uniformly from $\mathbb{Z}_{q_i}$, where $q_i$ is the smallest prime. The work of Corrigan-Gibbs and Kogan $[66]$ shows that solving the short exponent discrete logarithm problem is as hard as solving the standard discrete logarithm problem (in a group of order $q_i$) in the generic group model. That is, if $y \leftarrow \mathbb{Z}_{q_i}$, then the running time of the best known generic algorithm to compute $y$ given $G_i^{y_i}$ (for some $i \neq i^*$) is $O(\sqrt{q_i})$. We conjecture that this equivalence is still true even when given all ($G_i^{y_1}, \ldots, G_i^{y_n}$). This generalization is a natural analogue of the computational Diffie-Hellman $[67]$ problem across groups, where different group elements are replaced with different generators of different groups. This assumption was implicit in prior works $[68]$.

**VI. PERFORMANCE ANALYSIS**

We first evaluate each building block of our protocol separately, followed by the atomic swap protocol (Section V-D).

A. Verifiable Time DLog

We developed a prototype C implementation of $\Pi_{\text{VTD}}$ $[34]$ to demonstrate the feasibility of our construction. The implementation encompasses the Commit and Verify algorithms,
simulating thus the functionality that would be carried out by the atomic swap participants during a successful swap. We omit the Setup algorithm as this can be pre-computed and shared across several instances of atomic swaps, and the ForceOp algorithm, as it is only executed in case the coins are not swapped and the running time is pre-defined by the timing hardness $T$ \cite{72}. For the time-lock puzzles, we leveraged the implementation available in \cite{70}.

**Computation time.** We evaluated the VTD construction $\Pi_{\text{VTD}}$ for different values of the statistical parameter $n$. We set the threshold to $t = n/2$ (not the same as $T$ the hiding parameter in $\Pi_{\text{VTD}}$) as required for soundness of the VTD \cite{34}. We obtain the results shown in Table II. We observe that even with the value $n = 256$ (possibly too high to be used in practice), the overall running time is below 1 second, which is even below the block confirmation time of virtually all cryptocurrencies today that is in the order of several minutes. The practical advantage of VTD shows up when comparing it to verifiable timed signatures (VTS). As reported in \cite{31}, a VTS for the variants of Schnorr can be pre-computed and shared across several instances of atomic swaps, and the ForceOp algorithm, as it is only executed in case the coins are not swapped and the running time is pre-defined by the timing hardness $T$ \cite{72}. For the time-lock puzzles, we leveraged the implementation available in \cite{70}.

**Communication overhead.** Apart from the $\text{crs}$ and public parameters of the underlying time-lock puzzle, the overhead imposed by VTD is of (i) $n$ group elements for verifiability of Shamir secret sharing; (ii) a total of $4 - n$ group elements for the time-lock puzzles and related proofs; and (iii) a total of $t - 1$ pairs of two group elements $(x, r)$ for the cut and choose style proof. In summary, each party needs to store $5 \cdot n + (t - 1) \cdot n$ values of $Z_q$ to handle a VTD.

### B. 2-Party Protocols for Adaptor Signatures

Here, we have used the implementation of the 2 party computation protocol of adaptor signatures available at \cite{70}.

**TABLE II: Computation time of $\Pi_{\text{VTD}}$ (in seconds).**

<table>
<thead>
<tr>
<th>$n$</th>
<th>$t$</th>
<th>$\Pi_{\text{VTD}},\text{Commit}$</th>
<th>$\Pi_{\text{VTD}},\text{Verify}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>16</td>
<td>0.04</td>
<td>0.03</td>
</tr>
<tr>
<td>64</td>
<td>32</td>
<td>0.041</td>
<td>0.033</td>
</tr>
<tr>
<td>128</td>
<td>64</td>
<td>0.091</td>
<td>0.076</td>
</tr>
<tr>
<td>256</td>
<td>128</td>
<td>0.236</td>
<td>0.255</td>
</tr>
</tbody>
</table>

**Computation time.** We evaluated both the Schnorr and the ECDSA implementations. We observe that the joint pre-signature generation (Schnorr) requires 4.85 ms while that for ECDSA requires 266.30 ms. For both implementations, the pre-signature verification, witness extraction and pre-signature adaption are rather fast (i.e. $< 1$ ms). As expected, these results are similar to those in \cite{46}.

**Communication overhead.** The communication overhead is the same as reported in \cite{31} and we report it here for completeness. In particular, the 2PC protocol $\Pi_{\text{Schnorr AdpSg}}$ requires 256 bytes of communication between the two parties while its ECDSA counterpart $\Pi_{\text{ECDSA AdpSg}}$ requires 416 bytes.

### C. Atomic Swap Protocol

For the sake of clarity we chose not to include several optimisations to the description in Figure 5. We discuss them below and also incorporate them in our implementation.

**Implementation-level Optimizations.** In the setup phase (Figure 5), we create a new VTD commitment for each coin that is being swapped, requiring thus $n + \tilde{n}$ VTD commitments for both parties put together. However, in practice, one can have the same functionality while each party has to create and force open only a single VTD commitment.

To do this, assume for a moment that fungibility of the transactions is not a problem. Then each party could create a single key pair, embed the corresponding secret key in the VTD and use the corresponding public key repeatedly as an address for each of the coins that it owns (either $n$ or $\tilde{n}$). In order to gain fungibility back (i.e., make each public key look random to the eyes of the blockchain observer), the two parties carry out a coin toss protocol to generate a common randomness $r$ and generate each public key with respect to it. For instance, consider that $(pk, sk)$ was the key generated by $P_0$. The $i$-th public key can be computed as $pk^{H(r||i)}$. Note that given the original public key $pk$ and the randomness $r$, party $P_1$ can verify that the $i$-th key is correctly generated. While knowing the secret key $sk$, the party can derive $sk_i := sk \cdot H(r||i)$. We can also parallelise the computation of different instances of the 2PC protocol $\Pi_{\text{AdpSg}}$. In practice, it would be possible to use the multi-core architecture available at current commodity...
Global input: \( \{ v_{i}^{(0)}, p_{k_{i}}^{(0)} \}_{i \in [n]}, \{ v_{k}^{(1)}, p_{k_{k}}^{(1)} \}_{k \in [n]} \), T_{0}, T_{1}, \Delta. \) Here T_{0}, \Delta \in \mathbb{N} and T_{1} = T_{0} - \Delta.

Party P_{0}'s input: \( \{ sk_{i}^{(0)} \}_{i \in [n]} \), Party P_{1}'s input: \( \{ sk_{k}^{(1)} \}_{k \in [n]} \).

**Swap Setup Phase - Freezing Coins**

Parties P_{0} and P_{1} freeze the coins that they want to swap by doing the following:

1. For \( i \in [n] \), party P_{0} plays the role of U_{0} and party P_{1} plays the role of U_{1} in the freeze algorithm (Figure 4). Specifically,
   - Party P_{0}'s input: \( v_{i}^{(0)}, p_{k_{i}}^{(0)}, sk_{i}^{(0)}, T_{0} \), and party P_{1}'s input: \( v_{k}^{(0)}, p_{k_{k}}^{(0)}, T_{0} \).
   - Party P_{0}'s output: \( tx_{frz,i}, \sigma_{frz,i}, p_{k_{i}}^{(1)}, sk_{i}^{(1)}, C_{i}^{(0)}, \pi_{i}^{(0)} \), and party P_{1}'s output: \( p_{k_{i}}^{(0)}, sk_{i}^{(0)} \).

2. For \( k \in [n] \), party P_{0} plays the role of U_{0} and party P_{1} plays the role of U_{0} in the freeze algorithm (Figure 4). Specifically,
   - Party P_{0}'s input: \( v_{k}^{(1)}, p_{k_{k}}^{(1)}, T_{1} \), and party P_{1}'s input: \( v_{k}^{(1)}, p_{k_{k}}^{(1)}, sk_{k}^{(1)}, T_{1} \).
   - Party P_{0}'s output: \( pk_{k}^{(10)}, sk_{k}^{(10)} \), and party P_{1}'s output: \( tx_{frz,k}, \sigma_{frz,k}, pk_{k}^{(10)}, sk_{k}^{(10)}, C_{k}^{(1)}, \pi_{k}^{(1)} \).

**Swap Lock Phase**

1. Party P_{0} picks \( (Y,y) \in R \) using GenR(1^{\lambda}) and sends Y to party P_{1}.
2. Parties then setup the transactions doing the following:
   - Party P_{1} generates \( (pk_{k}^{\text{swap},k}, sk_{k}^{\text{swap},k}) \leftarrow \Pi_{\text{SIG}}^{\text{DS-KGen}}(1^{\lambda}) \) for \( i \in [n] \).
   - Party P_{1} generates swap transaction \( tx_{\text{swap},i}^{(1)} = tx(pk_{k}^{(01)}, p_{k_{k}}^{(1)}, v_{i}^{(0)}) \) for \( i \in [n] \) and sends it to party P_{0}.
   - Party P_{0} generates \( (pk_{k}^{\text{swap},k}, sk_{k}^{\text{swap},k}) \leftarrow \Pi_{\text{SIG}}^{\text{DS-KGen}}(1^{\lambda}) \) for \( k \in [n] \).
   - Party P_{0} generates swap transaction \( tx_{\text{swap},k}^{(0)} = tx(pk_{k}^{(10)}, p_{k_{k}}^{(0)}, v_{k}^{(1)}) \) and sends it to party P_{1}.
3. For \( i \in [n] \), parties P_{0} and P_{1} run a 2PC protocol \( \Gamma_{\text{SIG}}^{\text{AdpSg}} \) that does the following:
   - The 2PC protocol takes as common input \( tx_{\text{swap},i}^{(0)} \) from the parties, and as private input \( sk_{i}^{(01)} \) from party P_{0} and as private input \( sk_{k}^{(01)} \) from party P_{1}.
   - If \( \text{SIG} = \text{Schnorr} \), compute \( s_{k}^{(01)} := (sk_{0,i}^{(01)} + sk_{1,i}^{(01)}) \mod q \), and if \( \text{SIG} = \text{ECDSA} \), compute \( s_{k}^{0} := (sk_{0,i}^{(01)} \cdot sk_{1,i}^{(01)}) \mod q \).
   - It computes \( \sigma_{\text{swap},i}^{(1)} \leftarrow \Pi_{\text{SIG}}^{\text{AdpSg}} \cdot \text{pSign} \left( sk_{i}^{(01)}, tx_{\text{swap},i}^{(0)}, Y \right) \) and outputs \( \sigma_{\text{swap},i}^{(1)} \) first to \( P_{0} \) and then to \( P_{1} \).
   - Both parties check if \( \Pi_{\text{SIG}}^{\text{AdpSg}} \cdot \text{pVf} \left( pk_{k}^{(01)}, tx_{\text{swap},i}^{(0)}, Y; \sigma_{\text{swap},i}^{(0)} \right) = 1 \), and abort otherwise.
   - In the end, both parties \( P_{0} \) and \( P_{1} \) obtain \( \{ \sigma_{\text{swap},i}^{(1)} \}_{i \in [n]} \).
4. After the above step is successful, for all \( k \in [n] \), party P_{0} and P_{1} run the 2PC protocol \( \Gamma_{\text{SIG}}^{\text{AdpSg}} \) that does the following:
   - The 2PC protocol takes as common input \( tx_{\text{swap},k}^{(0)} \) from the parties, and as private input \( sk_{k}^{(10)} \) from party P_{0} and as private input \( sk_{1,k}^{(10)} \) from party P_{1}.
   - If \( \text{SIG} = \text{Schnorr} \), compute \( s_{k}^{(10)} := (sk_{0,k}^{(10)} + sk_{1,k}^{(10)}) \mod q \), and if \( \text{SIG} = \text{ECDSA} \), compute \( s_{k}^{0} := (sk_{0,k}^{(10)} \cdot sk_{1,k}^{(10)}) \mod q \).
   - It computes \( \sigma_{\text{swap},k}^{(0)} \leftarrow \Pi_{\text{SIG}}^{\text{AdpSg}} \cdot \text{pSign} \left( sk_{k}^{(10)}, tx_{\text{swap},k}^{(0)}, Y \right) \) and outputs \( \sigma_{\text{swap},k}^{(0)} \) to both parties first to \( P_{0} \) and then to \( P_{1} \).
   - Both parties check if \( \Pi_{\text{SIG}}^{\text{AdpSg}} \cdot \text{pVf} \left( pk_{k}^{(10)}, tx_{\text{swap},k}^{(0)}, Y; \sigma_{\text{swap},k}^{(0)} \right) = 1 \), and abort otherwise.
   - In the end, both parties \( P_{0} \) and \( P_{1} \) obtain \( \{ \sigma_{\text{swap},k}^{(0)} \}_{k \in [n]} \).

Fig. 5: Atomic Swap protocol run between parties \( P_{0} \) and \( P_{1} \) where \( \text{SIG} = \{ \text{Schnorr}, \text{ECDSA} \} \)

machines to perform this parallelization for moderate amount of coin swaps.

**Performance.** With the aforementioned implementation-level optimizations, we do a back-of-the-envelope calculation of the performance of our protocol by extracting the number of invocations to the underlying building blocks, namely (i) adaptor signatures \( \Pi_{\text{SIG}}^{\text{AdpSg}} \); (ii) VTID commit and verify; and (iii) standard digital signature scheme \( \Pi_{\text{DS-Sign}} \). We show our results in Table III. Overall we observe that the protocol uses several operations only once. The costliest operation is
1) For all $k \in [\tilde{n}]$ party $P_0$ computes $\alpha_{swp,k}^{(0)} \leftarrow \Pi_{\text{AS}}^{\text{Adapt}}(\tilde{\alpha}_{swp,k}^{(0)}, y)$, and posts $(tx_{swp,k}^{(0)}, \sigma_{swp,k}^{(0)})$ on the blockchain $\mathbb{B}$.

2) Party $P_1$ picks $j \in [\tilde{n}]$, and does the following:
   - Compute $y \leftarrow \Pi_{\text{AS}}^{\text{Ext}}(\tilde{\alpha}_{swp,j}^{(0)}, \tilde{\alpha}_{swp,j}^{(0)}, Y)$, and for all $i \in [\tilde{n}]$, compute $\sigma_{swp,i}^{(1)} \leftarrow \Pi_{\text{AS}}^{\text{Adapt}}(\tilde{\alpha}_{swp,i}^{(1)}, y)$
   - It posts $(tx_{swp,i}^{(1)}, \sigma_{swp,i}^{(1)})$ for all $i \in [\tilde{n}]$ on the blockchain $\mathbb{B}$ (corresponding to currency of $i$-th coin).

**Swap Timeout Phase**

1) If party $P_0$ fails to post $(tx_{swp,j}^{(0)}, \sigma_{swp,j}^{(0)})$ for any $j \in [\tilde{n}]$ on chain before time $T_1$, party $P_1$ does the following:
   - Finish computing $sk_{0,j}^{(10)} \leftarrow \Pi_{\text{VTD}}.\text{ForceOp}(C_{j}^{(1)}).
   - If $\text{SIG} = \text{Schnorr}$, compute $sk_{j}^{(10)} := (sk_{0,j}^{(10)} + sk_{1,j}^{(10)}) \mod q$, else if $\text{SIG} = \text{ECDSA}$, compute $sk_{j}^{(10)} := (sk_{0,j}^{(10)} \cdot sk_{1,j}^{(10)}) \mod q$.
   - Generate fresh key pairs $(pk_{i}^{(1)}, sk_{i}^{(1)}) \leftarrow \Pi_{\text{DS}}^{\text{KGen}}(1\lambda)$, and generate redeem transaction $tx_{\text{rfnd},j}^{(1)} := tx(\sigma_{j}^{(1)}, tx_{\text{rfnd},j}^{(1)})$ on the redeem transaction.
   - Generate a signature $\sigma_{j}^{(1)} \leftarrow \Pi_{\text{DS}}^{\text{Sign}}(sk_{j}^{(10)}, tx_{\text{rfnd},j}^{(1)})$ on the redeem transaction.
   - Reclaim the $v_{j}^{(1)}$ coins by posting $(tx_{\text{rfnd},j}^{(1)}, \sigma_{j}^{(1)})$ on the blockchain $\mathbb{B}$ (corresponding to currency of $j$-th coin).

2) Similarly, if party $P_1$ fails to post a valid transaction-signature pair $(tx_{swp,j}^{(1)}, \sigma_{swp,j}^{(1)})$ for any $j \in [\tilde{n}]$ on chain before time $T_0$, party $P_0$ follows steps analogous to above and reclaims the $v_{j}^{(0)}$ coins by posting $(tx_{\text{rfnd},j}^{(0)}, \sigma_{j}^{(0)})$ on the blockchain $\mathbb{B}$.

![Fig: Swap complete and timeout phase of the atomic swap protocol between $P_0$ and $P_1$ where $\text{SIG} = \{\text{Schnorr, ECDSA}\}$](image)

**TABLE IV: Gas cost of swaps in Ethereum**

<table>
<thead>
<tr>
<th></th>
<th>Create</th>
<th>Redeem</th>
<th>Refund</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTLC</td>
<td>134320</td>
<td>79752</td>
<td>58065</td>
</tr>
<tr>
<td>Our $n$-to-$\tilde{n}$ Schnorr/ECDSA Swap</td>
<td>21000</td>
<td>21000</td>
<td>21000</td>
</tr>
</tbody>
</table>

**TABLE III: Operations required in our atomic swap protocol.**

<table>
<thead>
<tr>
<th>Op type</th>
<th>Setup Phase</th>
<th>Lock Phase</th>
<th>Complete Phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joint key generation</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Key generation</td>
<td>0</td>
<td>$(n + \tilde{n})$</td>
<td>0</td>
</tr>
<tr>
<td>Signing</td>
<td>$(n + \tilde{n})$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Signature verify</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>VTD commit</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>VTD verify</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Joint pre-signing</td>
<td>0</td>
<td>$(n + \tilde{n})$</td>
<td>0</td>
</tr>
<tr>
<td>Pre-signature verify</td>
<td>0</td>
<td>$(n + \tilde{n})$</td>
<td>0</td>
</tr>
<tr>
<td>Pre-signature adapt</td>
<td>0</td>
<td>$(n + \tilde{n})$</td>
<td>0</td>
</tr>
<tr>
<td>Witness extract</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Comparison with HTLC. We have taken the HTLC implementation in Solidity available at [71] and calculated the gas costs to compute the three operations required in a swap: (i) create the HTLC contract; (ii) redeem the contract with a valid hash preimage; and (iii) refund the contract in case the timeout expires. The gas costs are shown in Table IV.

We observe that in the case of universal swap, we only require a standard transfer of ETH between two accounts, an inexpensive operation (since it is the basic one) in Ethereum. In contrast, the HTLC-based operations require between 2.7x and 6.3x more gas.

Apart from higher gas cost, the HTLC-based approach also requires a higher transaction size, since it needs to store the parameters of the contract (i.e., hash value, receiver and timeout). This additional overhead not only hinders the scalability of the underlying blockchain as the on-chain space is limited, but also the fungibility of the swap, as it is trivial for an observer to match two coins swapped as they share the same hash value.

In summary, although the swap functionality would be easily implementable in a smart contract such as HTLC, our protocol is preferable due to smaller on-chain cost both in gas and transaction size, fungibility guarantees and backwards compatibility with virtually all blockchains available today.

**VII. Conclusion**

In this work we investigate the problem of atomic swaps of multiple assets across all cryptocurrencies. We propose a universal protocol supporting $n$-to-$\tilde{n}$ swaps, without relying on any special scripts from the blockchain apart from the ability to verify signatures. Following the outline of the generic protocol we give a highly efficient protocol for the specific cases where the transactions are signed with ECDSA or Schnorr signatures. We also explore extensions to multi-party cyclic swaps and cross-curve swaps. In terms of future work, we aim at developing efficient solutions for other signature schemes with different properties (e.g., post-quantum security or aggregatable signatures).
REFERENCES


[27] https://thecharlatan.ch/Monero-Unlock-Time-Privacy/.


We give an overview of the related work in the following.

**HTLC-Based Atomic Swaps.** First introduced by Tier-Nolan [72], HTLC-based atomic swaps have constituted the core building block for several cryptocurrency transfers and exchange protocols [7], [73]. These protocols however suffer from the drawbacks on HTLC- and timelock-compatibility as described previously in this section.

**Other Approaches for Atomic Swaps.** Atomic swaps based on the Atomic Multi-Hop Locks (AHML) primitive [46] depart from HTLC-based ones in that the cryptographic condition to coordinate the swap is embedded in the creation of the digital signature that authorizes the swap, thereby departing from the need of hash functions. Yet, AMHL-based swaps as in [46] still rely on ledgers compatible with timelock functionality and support single-asset swaps. Recently, [68] have proposed an atomic swap protocol between Bitcoin and Monero that does not require hash functionality at any of the ledgers while timelock functionality is only required at one of the ledgers (i.e., in this case Bitcoin). Yet, this protocol is single-asset swap and tailored to these two ledgers.

**Decentralized Exchanges.** Leveraging the expressive scripting language available at cryptocurrencies like Ethereum, decentralized exchanges are implemented as a smart contract that encodes the (somewhat complex) logic to exchange multiple assets among multiple users. This approach is, however, supported only by those cryptocurrencies with expressive scripting language such as that in Ethereum and forces users with assets in other ledgers to migrate their assets to Ethereum [74], [75].

**Other Cross-chain Communication.** Apart from the cryptocurrency transfers and exchanges, cross-chain communication is a critical component for scalability solutions such as sharding, feature extensions via sidechains, as well as bootstrapping of new systems. Although these applications are out of the scope of this paper, we find them an interesting future research direction. We refer the reader to [76] for further reading.
Scriptless Payment Channel Network. Lockable Signature was a recently introduced tool [77] useful to construct payment channel networks scriptlessly from any signature scheme. However the resulting protocol still relies on time-lock scripts from the currencies to realize payment expiry, and therefore is not universal.

C. Security analysis of construction of Atomic Swaps from Adaptor Signatures for Schnorr and ECDSA signatures

Proof 1 (Proof of Theorem 5.7). We now prove that our protocol in Figure 5 securely US-realizes the atomic swap functionality from Figure 6.

We describe a simulator $S$ that handles either of the parties $P_0$ or $P_1$ that are corrupted by a PPT $A$ and simulates the real world execution protocol while interacting with the ideal functionality $F_{\text{swap}}$. We have a static corruption where the environment $E$ at the beginning of a session specifies the corrupted parties and the honest parties. The simulator $S$ faithfully impersonates the honest party. For operations exclusively among corrupted users, the environment does not expect any interaction with the simulator. Similarly, communications exclusively among honest nodes happen through secure channels and therefore the attacker does not gather any additional information other than the fact that the communication took place. For simplicity, we omit these operations in the description of our simulator.

The operations to be simulated for a $n$-to-$\tilde{n}$ atomic swap are described in the following.

In describing $S$’s operations for swapping, we begin by describing a series of hybrid executions, where we begin with a real world execution and gradually change the simulation in these hybrids and then argue about the proximity of neighbouring experiments. Simulator $S$’s execution for the payment operation is defined as the final hybrid’s execution. Below we describe the hybrid executions first and later argue about their proximity. Note that the switching of hybrid executions is performed over every session, but one at a time and we only discuss here a single time for simplicity and readability.

Hybrid $H_0$: This is the same as the real execution of the protocol in Figure 5.

Hybrid $H_1$: This is the same as the above execution except now the 2PC protocol $\Gamma^{\text{Sig}}_{\text{JKGen}}$ in the freezing coins of swap setup phase to generate shared keys is simulated using the 2PC simulator $S_{2pc,1}$ for the corrupted parties. Notice that such a simulator is guaranteed to exist for the 2PC protocol $\Gamma^{\text{Sig}}_{\text{JKGen}}$. Rest of the execution is unchanged from $H_0$.

Hybrid $H_2$: This is the same as the above execution except now the 2PC protocol $\Gamma_{\text{AdpSig}}$ in the swap lock phase to generate the pre-signatures is simulated using $S_{2pc,2}$ for the corrupted parties.

Hybrid $H_3$: This is the same as the above execution except now if in some session $q_0$, the adversary corrupts user $P_1$ and the adversary outputs $\sigma^*$ on a transaction $\tau^{(1)}_{\text{swap},i}$ under the public key $p_{k_{i}^{(1)}}$ for some $i \in [n]$ such that $\Pi_{\text{DS}}^{\text{SIG}} \mathbf{Vf} \left( p_{k_{i}^{(1)}}, \tau^{(1)}_{\text{swap},i}, \sigma^* \right) = 1$, before the simulator initiates the swap on behalf of $P_0$, the simulator aborts by outputting abort$_1$.

Hybrid $H_4$: This is the same as the above execution except now if in some session $q_4$, the adversary corrupts $P_0$ and outputs $\sigma^*$ on a transaction $\tau^{(0)}_{\text{swap},k}$ under the public key $p_{k_{i}^{(0)}}$ for some $k \in [\tilde{n}]$ such that $\Pi_{\text{DS}}^{\text{SIG}} \mathbf{Vf} \left( p_{k_{i}^{(0)}}, \tau^{(0)}_{\text{swap},k}, \sigma^* \right) = 1$. The simulator computes $y' \leftarrow \Pi_{\text{AS}}^{\text{SIG}} \mathbf{Ext} \left( \sigma^*, \sigma^{(0)}_{\text{swap},k}, V \right)$ and if $(Y, y') \notin R$, the simulator aborts by outputting abort$_2$.

Hybrid $H_5$: This is the same as the above execution except now if in some session $q_5$, the adversary corrupts $P_0$ and outputs $\sigma^*$ on a transaction $\tau^{(0)}_{\text{swap},k}$ under the public key $p_{k_{i}^{(1)}}$ for some $k \in [\tilde{n}]$ such that $\Pi_{\text{DS}}^{\text{SIG}} \mathbf{Vf} \left( p_{k_{i}^{(0)}}, \tau^{(0)}_{\text{swap},k}, \sigma^* \right) = 1$.

- the simulator computes $y_1 := \Pi_{\text{AS}}^{\text{SIG}} \mathbf{Ext} \left( \sigma^*, \sigma^{(0)}_{\text{swap},k}, V \right)$ and $(Y, y) \in R$
- the simulator computes for all $i \in [n]$, $\sigma_{\text{swap},i}^{(1)} := \Pi_{\text{AS}}^{\text{SIG}} \mathbf{Adapt} \left( \sigma_{\text{swap},i}^{(0)}, y \right)$

and there exists some $i^* \in [n]$ such that

$$\Pi_{\text{DS}}^{\text{SIG}} \mathbf{Vf} \left( p_{i^*}^{(1)}, \tau_{\text{swap},i^*}^{(1)}, \sigma_{\text{swap},i^*}^{(1)} \right) = 0$$

the simulator aborts by outputting abort$_3$.

Hybrid $H_6$: This is the same as the above execution except now if in some session $q_6$, an adversarial $P_0$, outputs any transaction $\tau^*$ and a signature $\sigma^*$ such that $\Pi_{\text{DS}}^{\text{SIG}} \mathbf{Vf} \left( p_{k_{i}^{(1)}}, \tau^*, \sigma^* \right) = 1$ for some $i \in [n]$ before time $T_0$, the simulator aborts by outputing abort$_{\text{PRIV},0}$.

Hybrid $H_7$: This is the same as the above execution except now if in some session $q_7$, an adversarial $P_1$, outputs any transaction $\tau^*$ and a signature $\sigma^*$ such that $\Pi_{\text{DS}}^{\text{SIG}} \mathbf{Vf} \left( p_{k_{i}^{(0)}}, \tau^*, \sigma^* \right) = 1$ for some $k \in [\tilde{n}]$ before time $T_1$, the simulator aborts by outputing abort$_{\text{PRIV},1}$.

Hybrid $H_8$: This is the same execution as above except now, if in some session $q_8$, the adversary corrupts $P_0$, and the simulator obtains $\mathbf{sk}' \leftarrow \Pi_{\text{VTD}} \mathbf{ForceOp} \left( C_{j}^{(1)} \right)$ for some $j \in [\tilde{n}]$. The simulator aborts with abort$_4$, if $\mathbf{sk}' \neq \mathbf{sk}_{j_{0},3}^{(10)}$ where $\mathbf{sk}_{j_{0},3}^{(10)}$ is the secret share of the adversary generated in the freeze coin phase.

Hybrid $H_9$: This is the same execution as above except now, if in some session $q_9$, the adversary corrupts $P_1$, and the simulator obtains $\mathbf{sk}' \leftarrow \Pi_{\text{VTD}} \mathbf{ForceOp} \left( C_{j}^{(0)} \right)$ for some $j \in [n]$. The simulator aborts with abort$_5$, if $\mathbf{sk}' \neq \mathbf{sk}_{j_{1},3}^{(10)}$ where $\mathbf{sk}_{j_{1},3}^{(10)}$ is the secret share of the adversary generated in the freeze coin phase.

Simulator $S$: The execution of the simulator is defined as the execution in $H_0$ while it interacts with the ideal functionality $F_{\text{swap}}$. The simulator receives $(\mathbf{swap}_1, \mathbf{id}, V, V, PK, PK, U_1)$ and $(\mathbf{swap}_2, \mathbf{id}, V, V, PK, PK, U_0)$ from $F_{\text{swap}}$. It proceeds as in the execution of $H_0$ by simulating the view of the adversary appropriately as it receives messages from the ideal.
we use the notation ≈. We show that the probability of the adversary corrupts the simulator initiates the swap on behalf of $P_i$ on a transaction $P_i$ the adversary corrupts user simulator extracts the adversary’s key shares in security of the 2PC protocol for the pre-signature generation $\Gamma_{\text{AdpSg}}$. The security of the 2PC protocols, namely $\Gamma_{\text{AdpSg}}$ guarantees the existence of $S_{2pc,1}$. $\Gamma_{\text{AdpSg}}$ guarantees the existence of $S_{2pc,2}$. Notice that the simulator extracts the adversaries key shares in $\Gamma_{\text{AdpSg}}$ using $S_{2pc,1}$ and uses them in the simulation of $S_{2pc,2}$.

The only difference between the hybrids is that in $H_3$ the simulator aborts with abort$_1$ if in some session $q_3$, the adversary corrupts user $P_1$ and the adversary outputs $\sigma^*$ on a transaction $tx_{\text{swp},i}^{(1)}$ under the public key $pk_i^{(01)}$ for some $i \in [n]$ such that $\Pi_{\text{DS}}^{\text{SIG},Vf} \left( pk_i^{(01)}, tx_{\text{swp},i}^{(1)}, \sigma^* \right) = 1$, before the simulator initiates the swap on behalf of $P_0$.

The only difference between the hybrids is that in $H_4$, the simulator aborts with abort$_2$ if in some session $q_4$, the adversary corrupts $P_0$ and does the following:

- the adversary outputs $\sigma^*$ on a transaction $tx_{\text{swp},k}^{(0)}$ under the public key $pk_k^{(10)}$ for some $k \in [n]$ such that $\Pi_{\text{DS}}^{\text{SIG},Vf} \left( pk_k^{(10)}, tx_{\text{swp},k}^{(0)}, \sigma^* \right) = 1$.
- the simulator computes $y' \leftarrow \Pi_{\text{AS}}^{\text{SIG},\text{Ext}} \left( \sigma^*, \sigma_{\text{swp},k}^{(0)}, Y \right)$ and we have $(Y, y') \notin R$.

The only difference between the hybrids is that in $H_5$, the simulator aborts with abort$_3$ if in some session $q_5$, the adversary corrupts $P_0$ and does the following:

- the adversary outputs $\sigma^*$ on a transaction $tx_{\text{swp},k}^{(0)}$ under the public key $pk_k^{(10)}$ for some $k \in [n]$ such that $\Pi_{\text{DS}}^{\text{SIG},Vf} \left( pk_k^{(10)}, tx_{\text{swp},k}^{(0)}, \sigma^* \right) = 1$.
- the simulator computes $y \leftarrow \Pi_{\text{AS}}^{\text{SIG},\text{Ext}} \left( \sigma^*, \sigma_{\text{swp},k}^{(0)}, Y \right)$ and $(Y, y) \in R$.
- the simulator computes for all $i \in [n]$, $\sigma_{\text{swp},i}^{(1)} \leftarrow \Pi_{\text{AS}}^{\text{SIG},\text{Adapt}} \left( \sigma_{\text{swp},i}^{(1)}, Y \right)$ and there exists some $i^* \in [n]$ such that $\Pi_{\text{DS}}^{\text{SIG},Vf} \left( pk_{i^*}^{(01)}, tx_{\text{swp},i^*}^{(1)}, \sigma_{\text{swp},i^*}^{(1)} \right) = 0$.

We show that the probability of abort$_3$ event triggered in $H_5$ is 0.