Sweep-UC: Swapping Coins Privately

Lucjan Hanzlik
CISPA Helmholtz Center for Information Security
hanzlik@cispa.de

Julian Loss
CISPA Helmholtz Center for Information Security
loss@cispa.de

Sri AravindaKrishnan Thyagarajan
NTT Research
t.srikrishnan@gmail.com

Benedikt Wagner
CISPA Helmholtz Center for Information Security
Saarland University
benedikt.wagner@cispa.de

2024 IEEE Symposium on Security and Privacy (SP) | 979-8-3503-3130-1/24/$31.00 ©2024 IEEE | DOI: 10.1109/SP54263.2024.00081

***Abstract*—Fair exchange (also referred to as *atomic swap*) is a fundamental operation in any cryptocurrency that allows users to atomically exchange coins. While a large body of work has been devoted to this problem, most solutions lack on-chain privacy. Thus, coins retain a public transaction history which is known to degrade the *fungibility* of a currency. This has led to a flourishing line of related research on fair exchange with privacy guarantees. Existing protocols either rely on heavy scripting (which also degrades fungibility and leads to high transaction fees), do not support atomic swaps across a wide range of currencies, or come with incomplete security proofs.**

To overcome these limitations, we introduce *Sweep-UC*[1], the first fair exchange protocol that simultaneously is efficient, minimizes scripting, and is compatible with a wide range of currencies (more than the state of the art). We build Sweep-UC from modular sub-protocols and give a rigorous security analysis in the UC framework. Many of our tools and security definitions can be used in standalone fashion and may serve as useful components for future constructions of fair exchange.

***Index Terms*—Atomic Swap; Unlinkable exchange; Coin Mixing; Blind Signatures;**

1. Introduction

One of the most fundamental financial operations is exchanging one currency for another. Suppose that Alice has one unit of currency A that she wants to exchange for a unit of currency B. In the case of fiat currencies, she can rely on a centralized authority such as a bank to fairly implement the exchange on her behalf. Here, 'fair' means that Alice can be sure that the bank will pay her with an equivalent amount of currency of type B. When dealing with decentralized cryptocurrencies, however, things are not as simple. One can no longer rely on a trusted bank to provide a fair exchange, as the main goal of such a system is to avoid a single point of trust. Thus, rather than relying on a centralized service, a large body of work has studied the problem of *fair exchange* between two parties Alice (holding a unit of currency A) and Bob (holding a unit of currency B) [1], [2], [3], [4], [5], [6], [7]. The crucial security feature studied in these works is *atomicity* (or *fairness*): at the end of the exchange, either Alice has a coin (i.e., a unit of currency) of type B and Bob has a coin of type A, or both Alice and Bob keep their original coins. These proposals use the scripting languages of the underlying blockchains to enforce specific spending behaviours which can be leveraged to facilitate the exchange. Some of these solutions [2], [3] use a special type of script called *Hash Timelock Contract* (HTLC). Roughly speaking, Alice can use an HTLC script with the hash function H to temporarily freeze some of her coins as follows: The HTLC specifies a value h such that if Bob presents x with $H(x) = h$, Bob obtains Alice's coins. The HTLC script also specifies some time T, after which Alice is refunded her frozen coins if Bob has not claimed them. Other solutions rely on trusted hardware [7] or smart contracts [1], [4], [5], [6] such as those supported by Ethereum. Unfortunately, it is well known that using special scripts or contracts for swapping coins has severe drawbacks:

1) The resulting protocol is incompatible with currencies that do not offer such contracts, e.g., Monero [8].
2) The protocol results in expensive transactions for the users swapping their coins, as verifying special scripts or contracts on the blockchain incurs a higher fee than regular scripts like verifying signatures on transactions.
3) It results in poor on-chain privacy, or in other words, degrades the *fungibility* of swapped coins. In line with the latin proverb *pecunia non olet*, money should not be tainted by its origins. A currency is said to be fungible if all units/coins in the currency have the same value, independent of their history. However, the coins of transactions using special scripts are clearly distinguishable from those of regular transactions using only signature verification scripts. As a result, these coins accumulate a so-called *pseudo-value* which may ultimately lead to their censorship or being ransomed [9].

Existing Constructions. To overcome these issues, Thyagarajan, Malavolta and Moreno-Sanchez proposed *universal swaps* [10]. Their protocol enables the fair exchange of coins across arbitrary currencies while only requiring the bare minimum script from the underlying blockchain for verifying payments, namely, the verification of digital signatures. Unfortunately, their protocols do not offer an efficient solution for blockchains without support for adaptor

1. Read as *Sweep Ur Coins*.

DOI 10.1109/SP54263.2024.00081

signatures [11]. This strongly limits the applicability to important systems including Monero or the Chia network [12]. Due to the result of Erwig et al. [11], Chia (and any other system based on unique signatures) *provably* lacks support for adaptor signatures.

Tumblebit [13], A^{2}L [14], and BlindHub [15] are atomic swap protocols that take an alternate route. In these protocols, Alice changes her coins with an *untrusted intermediate party*, a *tumbler* (in the case of Tumblebit), or a *hub* (in the case of A^{2}L). While the intermediary can deny its service to Alice, atomicity of the exchange between Alice and the intermediary is guaranteed. Specifically, Alice can make a payment of a coin in currency A to the intermediary, and in return is guaranteed to get a payment of a coin in currency B from the intermediary. Relying on such an intermediary has many benefits. For example, Alice no longer has to solve the *bootstrapping problem* [2], [3], [1], [4], [5], [6], which is to find another user Bob to swap with. Instead, she only needs to interact with the (permanently available) intermediary. Thus, we call such an intermediary-based protocol a *bootstrapped* protocol. As a second benefit, these protocols also offer a privacy property called *unlinkability*. Informally, unlinkability asserts that neither the intermediary nor any other party can link the concrete coins that it swaps, provided there are many swaps happening simultaneously. In this manner, unlinkability can be used to break the transaction history of coins and improve on-chain privacy. Several academic and applied works [16], [17], [18], [19], [20] have shown that mere pseudonyms do not guarantee privacy or anonymity for the users and their coins. Many instances [21] have showcased the importance of privacy and anonymity of coins and there has been considerable effort like CoinJoin [22], CoinShuffle [23], [24], among many others to improve coin privacy. Even new currencies with enhanced privacy were developed from scratch [8], [25].

Unfortunately, Tumblebit critically relies on the support of HTLC scripts from the underlying blockchains, which results in poor compatibility (with systems like Monero). The use of HTLC scripts also results in higher transaction fees than standard transactions with signature verification scripts and poor fungibility (see above), as HTLC transactions can be easily traced and tracked. While this issue is improved in A^{2}L, it was found in a later work [26] that there was a gap in their security model that allowed for key recovery attacks on specific instantiations. The authors of [26] also propose fixes to A^{2}L called A^{2}L$^+$, but only prove security in an idealized model (the linear-only encryption (LOE) model) [27] with game-based security guarantees. They also propose a version called A^{2}L$^{\mathsf{UC}}$ in the *Universal Composability (UC)* framework [28], which unfortunately requires heavy cryptographic tools like general-purpose two party computation (GP-2PC). This makes A^{2}L$^{\mathsf{UC}}$ inefficient for immediate use. Moreover, both A^{2}L$^+$ and A^{2}L$^{\mathsf{UC}}$ do not offer compatibility with systems lacking adaptor signature support. A more recent protocol, BlindHub [15], is the first to allow payments with different amounts, thereby significantly increasing the anonymity set. Unfortunately, BlindHub is also restricted to adaptor signatures and relies on general-purpose zero-knowledge proofs (GP-ZK), introducing a similar efficiency penalty as for A^{2}L$^{\mathsf{UC}}$. Further, only one part of BlindHub, called BlindChannel, is shown to be UC secure, and parts of the analysis rely on the LOE model. We summarize these existing solutions in Table 1.

Our Goal. With this state of affairs, achieving UC security without using general-purpose 2PC, and extending the supported signature class beyond adaptor seems to be challenging. We are interested in a protocol that overcomes these limitations. Concretely, we ask the following question:

Is there a UC secure bootstrapped protocol for efficient and privacy-preserving fair exchange across a wide range of currencies?

1.1. Our Contribution

We answer the above question positively by presenting Sweep-UC. Like Tumblebit and A^{2}L (series), Sweep-UC is bootstrapped with an intermediary called the *sweeper* and can be used to swap coins unlinkably and atomically. We compare our protocol with existing solutions in Table 1 and summarize its properties below.

Efficiency and Security. Sweep-UC achieves the strong notion of UC security. At the same time, in contrast to [10], [26], it does not rely on any heavy cryptographic machinery such as general-purpose 2PC. In particular, we thereby solve the challenge raised in [26]. On the way, we introduce novel cut-and-choose techniques so as to avoid inefficient and theoretically unsound computations which treat random oracles as arithmetic circuits. We show the practicality of this approach by evaluating a prototype. We implement the algorithms required by the exchange and redeem protocols. In both cases, the sweeper's part requires less than a second on a standard laptop. The user's part requires around five seconds on the same platform to verify the cut-and-choose and around one second to finalize the protocol.

Compatibility. To support swaps between currencies A and B, Sweep-UC relies only on minimal scripting for verifying signatures[2]. As discussed, this preserves on-chain privacy and fungibility of the currencies involved. In terms of supported signature schemes, Sweep-UC is the first protocol that does not only support adaptor signatures. Namely, our techniques enable the support of both unique signatures and adaptor signatures in currencies A and B, in any combination. We give concrete instantiations for discrete-logarithm adaptor signatures, e.g., Schnorr or ECDSA [11], and BLS [30][3]. Our techniques carry over to many other signature schemes of this kind.

Modularity. Sweep-UC is presented and analyzed in a modular way. That is, we define two exchange-like primitives in

2. Similar to Tumblebit, A^{2}L, and its variants, it also relies on timelocks (e.g., the `locktime` script available in Bitcoin). Timelocks allow coins to be locked, such that they can be spent only after a delay. This much weaker scripting functionality can be eliminated using [29].

3. If we are willing to accept NIZK proofs about random oracles, we show that A can use any adaptor or unique signature scheme, and B can use any signature scheme.

Protocol	Scripts	Signature	UC	Amounts	Comments
Tumblebit [13]	HTLC	ECDSA	P	Fixed	
A^2L [14]	SV	Adaptor	✗	Fixed	Gap in proof
A^2L^+ [26]	SV	Adaptor	✗	Fixed	need LOE model
A^2L^{UC} [26]	SV	Adaptor	✓	Fixed	need GP-2PC
BlindHub [15]	SV	Adaptor	P	Variable	need GP-ZK, LOE Model
Sweep-UC	SV	Adaptor, BLS	✓	Fixed	

TABLE 1. COMPARISON OF OUR PROTOCOL Sweep-UC WITH PREVIOUS PROTOCOLS. WE COMPARE THE REQUIRED SCRIPTING FUNCTIONALITY, WHERE SV STANDS FOR SIGNATURE VERIFICATION. WE ALSO COMPARE THE SUPPORTED SIGNATURE SCHEMES, AS WELL AS THE SECURITY THAT IS PROVEN. FOR THAT, A "P" INDICATES THAT ONLY PARTS OF THE PROTOCOL ARE ANALYZED IN UC. FINALLY, WE COMPARE WHETHER THE PROTOCOLS REQUIRE A FIXED PAYMENT AMOUNT. ALL PROTOCOLS ADDITIONALLY REQUIRE A TIMELOCK SCRIPT, WHICH CAN BE REMOVED USING TOOLS FROM [29].

a game-based way (one per currency that is involved). Then, we show the UC security of Sweep-UC based on the game-based security of these sub-protocols in a black-box fashion. We think the definition of these sub-protocols is of great interest for two reasons. First, one may use these definitions and our constructions in other protocols. Second, it makes Sweep-UC easily extendable. For example, to support other currencies or further improve efficiency, one only has to focus on the construction of these game-based sub-protocols instead of doing a cumbersome UC proof again.

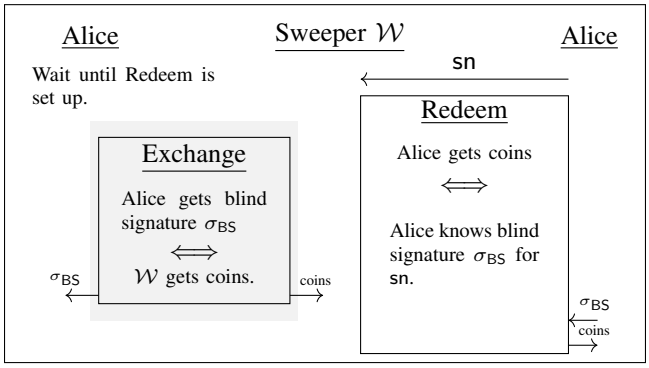

Figure 1. Informal overview of the high level structure of protocol Sweep-UC. The protocol is run between the sweeper $\mathcal{W}$ and a user Alice, using a redeem protocol and an exchange protocol as sub-protocols. The gray area represents an anonymous channel. The sweeper acts as a signer in the blind signature scheme.

2. Technical Overview

In this section, we give an overview of our construction and techniques. For our explanation, we follow a top-down approach. We first describe the protocol blueprint and how we model its security, and then show how to define and instantiate necessary building blocks. We consider a setting where a user Alice wants to swap coins with an intermediary called the *sweeper* $\mathcal{W}$[4]. This should be done in an atomic and unlinkable way.

Blueprint. Assume Alice owns addresses pk_{in} and pk_{out} and the sweeper owns $\mathsf{pk}_{\mathcal{W}}$. Our goal is to coordinate two payments $\mathsf{tx}_{\text{out}} = \mathsf{pk}_{a,\text{out}} \to \mathsf{pk}_{\mathcal{W}}$ in currency A and $\mathsf{tx}_{\text{in}} = \mathsf{pk}_{\mathcal{W}} \to \mathsf{pk}_{a,\text{in}}$ in currency B[5]. To coordinate these payments, Sweep-UC implements a form of Chaum's E-Cash [31], which is also the common high level structure of previous protocols [13], [14], [26], [32]. In this E-Cash approach, Alice signs tx_{out} using her secret key $\mathsf{sk}_{a,\text{out}}$ (associated with $\mathsf{pk}_{a,\text{out}}$) and obtains a voucher in exchange. Then, Alice can use that voucher to get a signature (valid with respect to $\mathsf{pk}_{\mathcal{W}}$) for tx_{in}. Let us now explain the steps of Sweep-UC in a bit more detail. An overview can be found in Figure 1. We assume that the sweeper holds the secret key $\mathsf{sk}_{\mathsf{BS}}$ for a blind signature scheme BS, and the corresponding public key $\mathsf{pk}_{\mathsf{BS}}$ is known to every user.

In the first step (right-hand side), Alice registers a random nonce sn at the sweeper via a protocol that we call *redeem protocol*. Intuitively, this protocol should ensure that whenever Alice has a valid blind signature σ_{BS} for sn in the future, she will be able to learn a signature for transaction tx_{in}. She could then publish this signature to get coins from $\mathcal{W}$. We can ensure that $\mathcal{W}$ can not spend these coins in the meantime by locking them in a shared address for a certain time, which is a standard technique[6].

In the second step (left-hand side), Alice executes a blind signature protocol for message sn with the sweeper. Here, Alice acts as the user, $\mathcal{W}$ has the role of the signer, and the messages are sent via an anonymous channel. In practice, this would be done via Tor, similar to what is done in previous works [32], [33]. In exchange, the signed payment tx_{out} is published, i.e., $\mathcal{W}$ gets coins. To ensure fairness, we wrap what we call an *exchange protocol* around the blind signature interaction. Finally (right-hand side), Alice uses the received blind signature σ_{BS} on sn in the redeem protocol to get a signature on payment tx_{in} and publishes the signed payment. One of the major design challenges to be overcome is to set up both the left and the right-hand side in a compatible way. We will come back to the required security guarantees for the exchange and redeem protocols later.

4. $\mathcal{S}$ is reserved for the simulator in the UC proof.

5. In practice, the sweeper would use a different key for each currency. We use one key $\mathsf{pk}_{\mathcal{W}}$ to simplify presentation.

6. This is why our protocol, as well as previous protocols, requires timelocks.

2.1. Challenge 1: UC Modeling

Before we start thinking about a UC proof, we need to define an appropriate ideal functionality $\mathcal{F}_{\mathsf{ux}}$. Our first attempt to do this is to have three interfaces, covering the three phases as above. That is, we have interfaces where the user can (1) register a receiving key $\mathsf{pk}_{\mathsf{in}}$ (right-hand side), (2) add a payment (left-hand side) by specifying $\mathsf{pk}_{\mathsf{out}}$ and refering to the registered $\mathsf{pk}_{\mathsf{in}}$, and (3) get the payment for $\mathsf{pk}_{\mathsf{in}}$ (right-hand side). Defining the details appropriately, we can argue that this models an atomic and unlinkable swap between a user and the sweeper. However, we run into a problem when we want to prove the security of our protocol. This problem, as discussed extensively in [26], arises from the blindness of blind signatures. It is the reason why the UC proof of $\mathrm{A}^2\mathrm{L}$ [14] is flawed. In a UC proof, a simulator will communicate with a corrupted user Alice, and it has to call the interface (2) appropriately. Specifically, it needs to refer to some previously registered (via interface (1)) key $\mathsf{pk}_{\mathsf{in}}$. If blindness of BS is unconditional, the simulator can not do that, as it can not extract the correct $\mathsf{pk}_{\mathsf{in}}$. For the case of computational blindness, we refer the reader to [26] for a detailed discussion. Namely, based on the common structure of known blind signature schemes, the authors of [26] elaborate that there is only little hope to get a secure system if blindness is computational. The solution taken in [26] is to rely on idealized models, which we want to avoid.

Solution: New Interface. Solving this fundamental problem is our first technical contribution. We view the problem as follows: When Alice interacts with $\mathcal{W}$ (or the simulator), she does not commit to the registration call for which she gets a blind signature. In other words, we cannot rule out that Alice changes the receiving public key $\mathsf{pk}_{\mathsf{in}}$ after obtaining the blind signature on the left. At the same time, there is no reason why we should rule this out. Namely, even if Alice changes $\mathsf{pk}_{\mathsf{in}}$ to $\mathsf{pk}'_{\mathsf{in}}$ afterwards, this does steal coins from the sweeper, as long as she can not redeem coins (interface (3)) for *both* $\mathsf{pk}_{\mathsf{in}}$ and $\mathsf{pk}'_{\mathsf{in}}$. With this in mind, we add an additional interface `ChangePayment`, that allows the simulator to change $\mathsf{pk}_{\mathsf{in}}$ to $\mathsf{pk}'_{\mathsf{in}}$ in case Alice is corrupted and both $\mathsf{pk}_{\mathsf{in}}, \mathsf{pk}'_{\mathsf{in}}$ have been registered before. Note that the number of coins that the sweeper spends in total stays the same, so this is still secure for the sweeper. Now, we can solve the commitment problem in the proof. Namely, the simulator can just use an arbitrary $\mathsf{pk}_{\mathsf{in}}$, and call `ChangePayment` with the correct $\mathsf{pk}'_{\mathsf{in}}$ afterwards, once it learns sn in the third phase of the protocol. Combined with what follows, this weakening of the functionality allows us to get UC security without using heavy cryptographic machinery or idealized models as in [26].

2.2. Challenge 2: Appropriate Building Blocks

To build our protocol in a modular way, we want to define syntax and game-based security notions for the exchange on the left and the redeem protocol on the right (see Figure 1). It turns out that finding security notions that are strong enough to be used in the UC proof but still possible to instantiate is non-trivial. We view the precise definitions of the building blocks as our second technical contribution. In this overview, we want to motivate the security notions for redeem and exchange protocols starting from the UC proof. As an example, we focus on the case of a corrupted user Alice and an honest sweeper $\mathcal{W}$.

Intuition. We want to avoid that $\mathcal{W}$ loses coins and this should follow from one-more unforgeability of BS. This is because $\mathcal{W}$ loses coins if it pays more on the right than it receives on the left. Hopefully, if the user learns a blind signature on the left, $\mathcal{W}$ receives a coin, and if $\mathcal{W}$ pays on the right, then the user must have known a blind signature.

Proof Challenge. To make this intuition formal in the UC proof, we would need to rule out the bad event that $\mathcal{W}$ loses money. Namely, the probability of this bad event should be bounded using a reduction $\mathcal{R}$ from one-more unforgeability. To recall, such a reduction has access to the public key of BS as well as a signer oracle. If $\mathcal{W}$ loses money, then reduction $\mathcal{R}$ should output $\ell + 1$ valid blind signatures, while interacting at most ℓ times with the signer oracle, for some $\ell \in \mathbb{N}$. If we think about this reduction, we may get information about how to define security of exchange and redeem protocols appropriately. Naturally, we may want to argue that

$$
\begin{aligned}
&(1) && \mathcal{W} \text{ receives } \ell \text{ coins on the left} \\
&\implies && \mathcal{R} \text{ uses signer oracle } \ell \text{ times,} \\
&(2) && \mathcal{W} \text{ pays } \ell + 1 \text{ coins on the right} \\
&\implies && \mathcal{R} \text{ outputs } \ell + 1 \text{ blind signatures.}
\end{aligned}
$$

Thus, we should establish that there is (1) a one-to-one correspondence between the number of payments received on the left and the number of times $\mathcal{R}$ needs to access the signer oracle, and (2) a one-to-one correspondence between the number of times the sweeper pays on the right and the number of blind signatures that $\mathcal{R}$ learns.

Implications for Building Blocks. We start with (1). To establish this in our proof, we have to remove all usage of the blind signature secret key $\mathsf{sk}_{\mathsf{BS}}$ from both redeem and exchange protocols. The only exception is the case in which the exchange on the left is completed, and therefore we know that $\mathcal{W}$ receives coins. Even in this case, we can only rely on a signer oracle for our simulation, as the reduction only has access to such an oracle and not to $\mathsf{sk}_{\mathsf{BS}}$ directly. More precisely, as long as we are not sure that the exchange protocol is completed and $\mathcal{W}$ receives coins, we have to simulate the messages in the exchange protocol without calling the signer oracle or using $\mathsf{sk}_{\mathsf{BS}}$. Once we know the exchange protocol is complete, we are allowed to use the signer oracle to make the simulation look consistent. Similarly, all messages sent by $\mathcal{W}$ on the right that are computed using $\mathsf{sk}_{\mathsf{BS}}$ have to be simulated without using $\mathsf{sk}_{\mathsf{BS}}$ or any signer oracle.

For (2), note that in the real protocol, $\mathcal{W}$ may never learn the blind signatures with which the user redeems its coins. This is because turning the blind signature into a transaction

signature has to be done locally without any interaction[7], and $\mathcal{W}$ only sees the resulting transaction signature on the chain. Therefore, the redeem protocol should provide some knowledge-style (online) extractor for the UC proof. This extractor should extract blind signatures whenever a user publishes a transaction signature.

2.3. Challenge 3: Efficient Instantiation

Next, we discuss the instantiation of exchange and redeem protocols, which is our third contribution. We have constructions for both unique signatures and adaptor signatures. In both cases, the blind signature scheme BS is the BLS blind signature scheme. Note that BS can be chosen independently of the used currencies, as blind signatures are only processed off-chain in our solution. In this blind signature scheme, the signing interaction consists of two messages $\mathsf{bsm}_1 \in \mathbb{G}$ and $\mathsf{bsm}_2 = \mathsf{bsm}_1^{\mathsf{sk}_{\mathsf{BS}}} \in \mathbb{G}$ in a cyclic group $\mathbb{G}$ of prime order p. For this overview, we focus on the case where BLS is used as the transaction signature scheme. The other constructions use similar ideas, replacing the need for uniqueness with the adaptor signature functionality.

A First Attempt. We start with the redeem protocol on the right. Here, the user Alice should be able to get a transaction signature σ for transaction $\mathsf{tx}_{\text{in}} = \mathsf{pk}_{\mathcal{W}} \rightarrow \mathsf{pk}_{a,\text{in}}$ once it knows the blind signature σ_{BS}. This should be possible without further interaction with $\mathcal{W}$, as $\mathcal{W}$ could go offline. A naive approach would be to let $\mathcal{W}$ encrypt σ into a ciphertext ct using σ_{BS} as a symmetric key. To convince Alice that she can really decrypt, i.e., ct is well-formed, $\mathcal{W}$ could append a non-interactive zero-knowledge proof (NIZK) π. That is, to set up the redeem protocol, $\mathcal{W}$ would send a "promise" message $\mathsf{prom} = (\mathsf{ct}, \pi)$ to Alice. Once Alice verified this message (by verifying π), Alice can be sure that she can decrypt σ using σ_{BS}, and start interacting on the left. While this solution seems to work intuitively, we encounter a problem in our analysis. Recall from our discussion about the security of building blocks that we would have to simulate ct and π without having access to $\mathsf{sk}_{\mathsf{BS}}$ or σ_{BS}. The challenge here is that once the user knows σ_{BS} (e.g., because it behaves honestly), the ciphertext ct should look consistent again. To implement this, we define $\mathsf{ct} := \mathsf{H}(\sigma_{\mathsf{BS}}) \oplus \sigma$ instead and use the programmability of the random oracle H. Namely, we send a simulated π and a random ct first, and program $\mathsf{H}(\sigma_{\mathsf{BS}}) := \mathsf{ct} \oplus \sigma$ once it is queried. We can use a similar approach for the exchange on the left, applying appropriate tweaks. Namely, we first establish that signing tx_{out} requires two signatures $\sigma_{\mathcal{W}}$ and σ_a by $\mathcal{W}$ and Alice, respectively[8]. We encrypt the blind signature response bsm_2 using transaction signature $\sigma_{\mathcal{W}}$ for transaction tx_a in the same way, i.e., $\mathsf{ct} := \mathsf{H}(\sigma_{\mathcal{W}}) \oplus \mathsf{bsm}_2$. When Alice receives ct and a NIZK π, she sends her share σ_a if π verifies. Then, once $\mathcal{W}$ publishes $\sigma_{\mathcal{W}}, \sigma_a$, Alice derives bsm_2 from ct.

7. Otherwise, if a user relied on interaction with the sweeper in this step, then a malicious sweeper could go offline and violate fairness.

8. This can be implemented using a multi-signature address.

The constructions sketched here have a significant shortcoming: We use NIZKs to prove relations defined by random oracle H. This non-standard use of the random oracle has unclear security implications, and we want to avoid it.

Strawman's Cut-and-Choose Solution. The challenge is that our current strategy crucially relies on the observability and programmability of the random oracle as well as the verifiability of the NIZK. We have to find a way to exploit these features of the random oracle while avoiding generic NIZKs about random oracle relations. In the following, we explain our solution for the redeem protocol only. The exchange protocol can be constructed through suitable modifications and switching roles as in our naive attempt. We also omit some minor details for readability. Roughly, our idea is to use cut-and-choose to implement the proof π. In such a technique, $\mathcal{W}$ would repeat the naive attempt in 2λ instances independently, and has to open λ randomly chosen instances to convince Alice of consistency. Clearly, this does not work directly because any opened instance already allows Alice to obtain money without knowing σ_{BS}. A first attempt to solve this problem using secret sharing is as follows:

1) $\mathcal{W}$ sends a ciphertext $\mathsf{ct}_0 = h^{f'(0)} \cdot \sigma$, and ciphertexts $\mathsf{ct}_j = \mathsf{H}(\sigma_{\mathsf{BS}}, j) \oplus h^{f'(j)}$, $j \in [2\lambda]$, where h is a generator of $\mathbb{G}$, and f' is a random polynomial of degree λ over $\mathbb{Z}_p$. $\mathcal{W}$ also commits to f' by sending its coefficients in the exponent. $\mathcal{W}$ can prove well-formedness of ct_0 using an efficient NIZK, as the statement is purely algebraic[9].
2) Alice challenges $\mathcal{W}$ to open ct_k by sending σ_{BS} and $f'(k)$ for λ randomly chosen k[10].
3) Using the opening, Alice can check consistency by recomputing ct_k for all opened k, and checking in the exponent that $f'(k)$ indeed lies on the polynomial f'. The cut-and-choose technique guarantees that, with overwhelming probability, at least one unopened ciphertext ct_{k^*} is also consistent.
4) To redeem, once Alice learns σ_{BS} from the exchange on the left, she can decrypt ct_{k^*} to learn $h^{f'(j)}$. Now, she has $\lambda+1$ evaluations of f' (in the exponent of h), which allows her to compute $h^{f'(0)}$ and decrypt ct_0.

This approach allows the user to check consistency without requiring the NIZK π. At the same time, we can still use the observability and programmability of the random oracle as in the naive attempt. However, note that this solution is heavily flawed: When $\mathcal{W}$ opens ct_k by sending σ_{BS} and $f'(k)$, the user learns σ_{BS}, and can therefore redeem its coins without interacting on the left. On a technical level, simulating the promise without knowing σ_{BS} will fail.

Our Cut-and-Choose Solution. To solve this, we introduce another layer of secret sharing. We use the structure of BLS blind signatures to share $\sigma_{\mathsf{BS}} = \mathsf{H}(\mathsf{sn})^{\mathsf{sk}_{\mathsf{BS}}}$ into $\sigma_j, j \in [2\lambda]$ using a random polynomial f of degree λ such that

$$f(0) = \mathsf{sk}_{\mathsf{BS}}, \qquad \mathsf{pk}_{\mathsf{BS}} = g^{\mathsf{sk}_{\mathsf{BS}}},$$
$$\mathsf{pk}_{\mathsf{BS},j} = g^{f(j)}, \qquad \sigma_j = \mathsf{H}(\mathsf{sn})^{f(j)}.$$

9. The relation is defined by h and first coefficients of f', and $\mathsf{pk}_{\mathcal{W}}$.

10. This can be done non-interactively using the Fiat-Shamir heuristic.

Then, each ciphertext has the form $\mathsf{ct}_j = \mathsf{H}(\sigma_j) \oplus h^{f'(j)}$, and can be opened by sending σ_j. Again, we publish coefficients of f in the exponent, which allows to publicly compute $\mathsf{pk}_{\mathsf{BS},j}$. Now, the user can check consistency of σ_j using $\mathsf{pk}_{\mathsf{BS},j}$ and BLS verification. Also, note that Alice (computationally) only learns λ points of f in the exponent of basis $\mathsf{H}(\mathsf{sn})$. Once Alice bought the blind signature σ_{BS} on the left, this serves as the $(\lambda+1)$st share, and she can reconstruct f in the exponent of basis $\mathsf{H}(\mathsf{sn})$, i.e., she learns all σ_j. Then, the argument is as before. Namely, soundness of the cut-and-choose guarantees that there is at least one unopened k^* for which ct_{k^*} is consistent. As in the strawman solution, she can now compute $h^{f'(0)}$ and therefore σ from ct_0. It turns out that, if implemented carefully, the random oracle-based simulation strategy works out now. Our simulator can know which indices are opened in advance. Then, without knowing $\mathsf{sk}_{\mathsf{BS}}$ and σ_{BS}, it can define the polynomial f such that $f(0) = \mathsf{sk}_{\mathsf{BS}}$ implicitly in the exponent, while still knowing λ points of f over $\mathbb{Z}_p$. These can be used to open consistently, and the unopened ct_j are sampled at random as in the naive attempt. Then, once Alice queries $\mathsf{H}(\sigma_j)$ for some unopened σ_j, the simulator can compute f entirely in the exponent of basis $\mathsf{H}(\mathsf{tx})$, and program $\mathsf{H}(\sigma_j) = \mathsf{ct}_j \oplus h^{f'(j)}$ for all unopened j.

3. Preliminaries

The security parameter $\lambda \in \mathbb{N}$ is given in unary to all algorithms implicitly as input. We write $x \leftarrow\!\!\$\, S$ if x is sampled uniformly at random from a finite set S. We write $x \leftarrow \mathcal{D}$ if x is sampled according to a distribution $\mathcal{D}$. An algorithm is said to be PPT if its running time is bounded by a polynomial in its input size. For an algorithm $\mathcal{A}$, we write $y \leftarrow \mathcal{A}(x)$, if y is output from $\mathcal{A}$ on input x with random coins sampled uniformly at random. We write $y := \mathcal{A}(x;\rho)$ to make the random coins ρ explicit. The notation $y \in \mathcal{A}(x)$ means that y is a possible output of $\mathcal{A}(x)$. A function $f : \mathbb{N} \to \mathbb{R}_+$ is said to be negligible in its input λ, if $f \in \lambda^{-\omega(1)}$. The first K natural numbers are denoted by $[K] := \{1, \ldots, K\}$. Next, we introduce the cryptographic primitives we use. For formal definitions, we refer to our full version [34], Appendix A.

Digital Signatures. A signature scheme $\mathsf{SIG} = (\mathsf{Gen}, \mathsf{Sig}, \mathsf{Ver})$ consists of three PPT algorithms. The key generation algorithm $\mathsf{Gen}(1^\lambda)$ generates a key pair $(\mathsf{pk}, \mathsf{sk})$. We require the public keys pk generated by Gen to have high entropy. The signing algorithm $\mathsf{Sig}(\mathsf{sk}, \mathsf{m})$ generates a signature σ on the message m. The verification algorithm $\mathsf{Ver}(\mathsf{pk}, \mathsf{m}, \sigma)$ validates the signature σ with respect to message m and public key pk and returns either 1 for valid, or 0 for invalid. A signature scheme is said to be *unique* if for any public key pk and message m, there exists exactly one σ with $\mathsf{Ver}(\mathsf{pk}, \mathsf{m}, \sigma) = 1$. The security property of interest is that of *unforgeability*. Here, an adversary without access to the secret key sk, should not be able to forge a valid signature on a fresh message, even given access to signatures on any arbitrary messages of its choice. (EUF-CMA security). Finally, we may require the signature scheme to be smooth, meaning that for any (not necessarily honestly generated) public key and message, a random string in the signature space is a valid signature only with negligible probability.

Blind Signatures. In a blind signature scheme [31] a user can obtain a signature on a message from a signer such that the signer does not learn the message itself. Formally, a blind signature scheme is a tuple $\mathsf{BS} = (\mathsf{Gen}, \mathsf{S}, \mathsf{U}, \mathsf{Ver})$, where Gen and Ver are as before. Signatures are generated in an interactive protocol between a user $\mathsf{U}(\mathsf{pk}, \mathsf{m})$ and a signer $\mathsf{S}(\mathsf{sk})$. We only consider two-move blind signature schemes, for which the interaction is as follows: $(\mathsf{bsm}_1, St) \leftarrow \mathsf{U}_1(\mathsf{pk}, \mathsf{m}), \quad \mathsf{bsm}_2 \leftarrow \mathsf{S}(\mathsf{sk}, \mathsf{bsm}_1), \quad \sigma \leftarrow \mathsf{U}_2(St, \mathsf{bsm}_2)$. We write $\sigma \leftarrow \mathsf{BS.Sig}(\mathsf{sk}, \mathsf{m})$ as a shorthand notation for this interaction. A *unique* blind signature scheme is defined exactly as in the case of standard digital signatures. In terms of security, two notions are considered. *Blindness* states that it should be infeasible for an adversarial signer to link the signing interaction to the message m and the resulting signature σ. For this work, we only need a relaxed version of this property referred to as *weak blindness*, where the adversary is not given σ, but only if σ was a valid signature or not. The second notion is *one-more unforgeability*, which guarantees that it is infeasible for an adversarial user to return $\ell + 1$ valid signatures after completing at most ℓ interactions with the signer.

Threshold Secret Sharing. We use Shamir secret sharing [35] and Lagrange interpolation over fields and in the exponent of a cyclic group. To this end, let p be a prime, and $\mathbb{G}$ be a cyclic group of order p, generated by $g \in \mathbb{G}$. Let $z \in \mathbb{Z}_p$ be fixed. We define algorithms $\mathsf{reconst}_p((x_0, y_0), \ldots, (x_\lambda, y_\lambda))$ and $\mathsf{reconst}_{g,z}((x_0, h_0), \ldots, (x_\lambda, h_\lambda))$ that take as input pairs $(x_i, y_i) \in \mathbb{Z}_p^2$ and $(x_i, h_i) \in \mathbb{Z}_p \times \mathbb{G}$, respectively, as follows: Both define polynomials $\ell_j(X) := \prod_{m \in \{0,\ldots,\lambda\}, m \neq j} (X - x_m)/(x_j - x_m) \in \mathbb{Z}_p[X]$. Algorithm $\mathsf{reconst}_p$ outputs $L(X) := \sum_{j=0}^{\lambda} y_j \cdot \ell_j(X) \in \mathbb{Z}_p[X]$, and $\mathsf{reconst}_{g,z}$ outputs $\prod_{j=0}^{\lambda} h_j^{\ell_j(z)}$. Further, given λ indices $(k_j)_{j\in[\lambda]}$ for $k_j \in [2\lambda]$, we define algorithm $\mathsf{polyGen}_{g,p}(\lambda, \mathsf{coeff}_0, (k_j)_{j\in[\lambda]})$ that internally generates a polynomial $f(X) \in \mathbb{Z}_p[X]$ of degree λ and outputs λ evaluations $((k_j, s_{k_j} := f(k_j))_{j\in[\lambda]}$ and λ coefficients $(\mathsf{coeff}_j)_{j\in[\lambda]}$. For the outputs, we have $g^{f(k_j)} = \prod_{i=0}^{\lambda} (\mathsf{coeff}_i)^{(k_j)^i}$ for all $j \in [\lambda]$ and $g^{f(0)} = \mathsf{coeff}_0$.

4. Security Model

Here, we discuss the security properties that we want to achieve and introduce our security model.

Security Properties and Threat Model. Throughout, all parties, including the sweeper, can be fully malicious and deviate from the protocol specification. Our protocol should satisfy three security properties, namely security for users, security for the sweeper, and unlinkability. Our protocol should achieve *security for users* in the sense that the sweeper should not be able to steal users coins. In other

words, whenever an honest user pays to the sweeper, it is guaranteed that it will be paid back by the sweeper, even if for example the sweeper goes offline. On the other hand, our protocol should ensure *security for the sweeper*. This means that colluding users should only be able to get coins from the sweeper if they paid before. Finally, we aim for *unlinkability*. This property means that if a lot of users interact with the sweeper at the same time, then neither the sweeper nor any outsider can link the interaction and payment in which the user paid to the sweeper to the interaction and payment in which the sweeper paid to the user. More concretely, let us denote an interaction between a user $\mathcal{P}_i$ and the sweeper in our protocol by two vertices a_i, b_i in a graph. Vertex a_i corresponds to the payment from $\mathcal{P}_i$ to the sweeper, and b_i corresponds to the payment from the sweeper to $\mathcal{P}_i$. Given a set of such users, consider the complete bipartite graph G on partitions $A = \{a_i\}$ and $B = \{b_i\}$. The actual payments induce a matching $M^* = \{(a_i, b_i)\}$. Our unlinkability definition now roughly states that both sweeper and outsiders obtain no information about M^*, except for what is already revealed by G. Note that we did not yet specify which users we consider in this model, i.e., the anonymity set. This will be made clear once we discuss the functionality.

UC Framework. We model the security of our protocol in the universal composabililty (UC) framework [28] with static corruptions. In terms of communication, our protocol makes use of secure and anonymous channels. In practice, one would implement the anonymous channel using Tor, similar to what is done in previous works [32], [33]. Also, we consider a synchronous model of communication. This means that we implicitly assume a global clock functionality [36], and protocols are executed in rounds. Every party is aware of the current round. Thus, parties and functionalities can expect messages to be received at a certain time. Assuming a synchronous model implicitly using a clock functionality is similar to other works in this area [37], [33], [38], [14], [10], [26]. Further, our construction makes use of random oracles within sub-protocols, for which we prove game-based security properties. In our UC proof, we then treat the sub-protocols as a black box. Especially, we remark that we do not use the global random oracle model [39], [40]. This heuristic is common practice in the area [13], [33], [38], [14], [10], where the sub-protocols are treated as a black-box during security analysis but instantiated in the random oracle model for practical efficiency.

Ledger Functionality. As in previous works [37], [10], we model the blockchain as a global ledger functionality $\mathcal{L}^{\mathsf{SIG}}$ parameterized by a signature scheme SIG. We postpone the formal presentation of $\mathcal{L}^{\mathsf{SIG}}$ to our full version [34], Figure 16. The functionality holds the current balances $\mathsf{bal}[\mathsf{pk}] \in \mathbb{N}_0$ of public keys pk. Parties can call $\mathcal{L}^{\mathsf{SIG}}.\mathtt{Pay}(\mathsf{pk}_s, \mathsf{pk}_r, c, \mathsf{sk}_s)$ to pay c coins from address pk_s to address pk_r using secret key sk_s. Further, we allow functionalities to call interfaces $\mathcal{L}^{\mathsf{SIG}}.\mathtt{Freeze}(\mathsf{pk}, c)$ and $\mathcal{L}^{\mathsf{SIG}}.\mathtt{Unfreeze}(\mathsf{pk}', c)$ to freeze c coins of an address pk or to unfreeze them into an address pk'. Also, our protocol makes use of a functionality $\mathcal{F}_s$, formally specified in our full version [34], Figure 17. Via interface $\mathcal{F}_s.\mathtt{OpenSh}(T, \mathsf{pk}_{in}, \mathcal{P}_b, c, \mathsf{sk}_{in})$ this functionality allows a party $\mathcal{P}_a$ to open a shared address $(\mathsf{pk}_a, \mathsf{pk}_b)$ with party $\mathcal{P}_b$ by paying c coins from pk_{in} into it. As a result, $\mathcal{P}_a$ gets secret key share sk_a and $\mathcal{P}_b$ gets secret key share sk_b. Later, it can be closed using $\mathcal{F}_s.\mathtt{CloseSh}(\mathsf{pk}_a, \mathsf{pk}_b, \mathsf{pk}_{out}, c, \sigma_a, \sigma_b)$, where σ_a, σ_b are valid signatures on a closing transaction tx with respect to $\mathsf{pk}_a, \mathsf{pk}_b$, respectively. In this case, the c coins are transferred to pk_{out}. If the shared address is not closed after timeout T, the coins go back to pk_{in}. For simplicity, we make use of the component-wise multi-signature here. It should be noted that everything easily carries over to more efficient and scriptless multi-signature schemes, the shared address consists of a single public key. We note that in the description of our protocol, the interfaces $\mathcal{L}^{\mathsf{SIG}}.\mathtt{Freeze}$ and $\mathcal{L}^{\mathsf{SIG}}.\mathtt{Unfreeze}$ are only called by $\mathcal{F}_s$, and it is well known [10] how to instantiate such a shared address functionality without scripts in existing cryptocurrencies like Bitcoin. Therefore, these two interfaces only serve for modeling purposes and do not introduce special scripts.

Unlinkable Exchange Functionality. We model the properties that our protocol should achieve as an ideal functionality $\mathcal{F}_{\mathsf{ux}}$ for unlinkable exchanges. The formal presentation of the functionality is postponed to Figure 3. The functionality interacts with $\mathcal{L}^{\mathsf{SIG}}$. It is parameterized by a timeout parameter T and an amount amt. All payments will have this fixed amount, which is important to maximize the anonymity set. When a user $\mathcal{P}$ wants to use $\mathcal{F}_{\mathsf{ux}}$ to exchange coins with the sweeper $\mathcal{W}$, it first calls interface $\mathcal{F}_{\mathsf{ux}}.\mathtt{Register}(\mathsf{pk}_b)$, which freezes amt coins of some fixed public key $\mathsf{pk}_{\mathcal{W}}$ of $\mathcal{W}$. Here, the adversary learns $\mathcal{P}, \mathsf{pk}_b$. Next, party $\mathcal{P}$ calls $\mathcal{F}_{\mathsf{ux}}.\mathtt{AddPayment}(\mathsf{pk}_a, \mathsf{sk}_a, \mathsf{pk}_b)$, which leads to amt coins of pk_a being transferred to $\mathsf{pk}_{\mathcal{W}}$. Here, the adversary only learns pk_a, and not $\mathcal{P}, \mathsf{pk}_b$. Finally, party $\mathcal{P}$ calls $\mathcal{F}_{\mathsf{ux}}.\mathtt{GetPayment}(\mathsf{pk}_b)$. If the corresponding calls to `Register` and `AddPayment` were issued correctly, this leads to unfreezing the amt coins that were frozen in `Register` into address pk_b. In this way, $\mathcal{P}$ paid amt coins from address pk_a to $\mathcal{W}$ and received amt coins to pk_b from $\mathcal{W}$. In addition to the natural interfaces above, we also introduce an interface `ChangePayment`, that allows the simulator to change receiving public keys pk_b if the party that called `AddPayment` is corrupted. The reason for this is discussed in the technical overview. Observe that the number of coins that $\mathsf{pk}_{\mathcal{W}}$ pays stays the same when calling this interface.

Let us argue how the informal security properties discussed above are captured by $\mathcal{F}_{\mathsf{ux}}$. A malicious $\mathcal{W}$ is always allowed to make the calls to `Register` and `AddPayment` abort. However, whenever `Register` and `AddPayment` were issued without such an abort, there is no way to stop the coin transfer to pk_b in `GetPayment`. Thus, the functionality provides security for users. On the other hand, a call to `GetPayment` will only lead to coins being transferred to pk_b, if `AddPayment` has been called before. This implies that the functionality provides security for the sweeper. Finally, note that the adversary can not link the calls to `AddPayment`

to the calls to Register, GetPayment using the outputs of $\mathcal{F}_{\mathsf{ux}}$. The only way he can link these calls is by their order in comparison with calls from other parties. Before, we described this unlinkability guarantee using a graph G and a matching M^*. What remains is to define under what conditions two users $\mathcal{P}_i$ and $\mathcal{P}_j$ that call the interfaces Register, AddPayment, and GetPayment belong to the same graph or anonymity set. For $x \in \{r = \texttt{Register}, a = \texttt{AddPayment}, g = \texttt{GetPayment}\}$ and $k \in \{i, j\}$ let $t_{x,k}$ be the time when user k calls interface x. Then, $\mathcal{P}_i$ and $\mathcal{P}_j$ belong to the same graph, if and only if $t_{r,i}, t_{r,j} < t_{a,i}, t_{a,j} < t_{g,i}, t_{g,j}$.

Simplifications. Let us now discuss the simplifications that we make and explain how one would have to deal with them when using our protocol in practice. It is easy to see that these simplifications do not change the security guarantees that we give. First, we do not include any fee for the sweeper in our model. In practice, a fee is necessary to incentivize the sweeper as a service. Also, in a practical application, it may be useful to introduce some epochs in which the users run the sub-protocols for Register, AddPayment, GetPayment. This would have a positive effect on the size of the anonymity set. Finally, to avoid clutter, we modeled our protocol for one ledger functionality, and thus one currency. However, the reader should notice that both our functionality and our construction can be trivially adapted to the setting of two different currencies. This is because the calls to $\mathcal{L}^{\mathsf{SIG}}$ in Register and GetPayment are completely independent of the calls to $\mathcal{L}^{\mathsf{SIG}}$ in AddPayment.

5. Building Blocks for Sweep-UC

In this section, we first define an exchange protocol and give different instantiations of it. Then, we define a redeem protocol and present constructions. In a nutshell, using an exchange protocol, a user will buy a blind signature from the sweeper. Then, using the redeem protocol, it can turn it in to get a signed transaction from the sweeper. Throughout, we use the terminology "on the left/right" following Figure 1.

5.1. Exchange Protocol

We define the syntax and security of the exchange protocol on the left. Later, we give instantiations of it.

Setting. Consider the following scenario for a signature scheme SIG and a blind signature scheme BS. A buyer and a seller[11] opened a shared address $(\mathsf{pk}_b, \mathsf{pk}_s)$ for SIG, where the buyer knows the secret key sk_b corresponding to pk_b, and the seller knows the secret key sk_s corresponding to pk_s. Both parties are aware of a public key $\mathsf{pk}_{\mathsf{BS}}$ for BS, and the seller knows the corresponding secret key $\mathsf{sk}_{\mathsf{BS}}$. Assume that the signing protocol of BS consists of two messages, bsm_1 and bsm_2. Then, the buyers has some nonce sn that should be signed (with respect to BS) by the seller. However, to get the signature, it should pay with a signature for a transaction tx under the shared address $(\mathsf{pk}_b, \mathsf{pk}_s)$. More precisely, first, the buyer sends the first message bsm_1 of the blind signature interaction. Then, both parties run an exchange protocol to fairly exchange the message bsm_2 for a signature (σ_b, σ_s) on transaction tx.

11. In the context of protocol Sweep-UC, the user Alice will act as the buyer, and the sweeper will have the role of the seller.

Syntax. In our syntax, we assume that the parameters $\mathsf{xpar} := (\mathsf{pk}_{\mathsf{BS}}, \mathsf{bsm}_1, \mathsf{pk}_b, \mathsf{pk}_s, \mathsf{tx})$ are known to the seller and the buyer. Then, the seller first sends a message xm_1 to the buyer, which is computed using the first message bsm_1 and the secret key $\mathsf{sk}_{\mathsf{BS}}$, and may already encapsulate the second message bsm_2 in some sense. Then, the buyer responds with a message xm_2. Now, the seller can derive the signature σ_b from xm_2. Whenever the seller publishes (σ_b, σ_s), the buyer can derive a valid second message bsm_2 from the transcript $\mathsf{xm}_1, \mathsf{xm}_2$ and (σ_b, σ_s).

Definition 1 (Exchange Protocol)**.** *Let* $\mathsf{SIG} = (\mathsf{SIG.Gen}, \mathsf{SIG.Sig}, \mathsf{SIG.Ver})$ *be a digital signature scheme. Further, let* $\mathsf{BS} = (\mathsf{BS.Gen}, \mathsf{BS.S}, \mathsf{BS.U} = (\mathsf{U}_1, \mathsf{U}_2), \mathsf{BS.Ver})$ *be a two-move blind signature scheme. An exchange protocol for* SIG *and* BS *is a tuple of PPT algorithms* $\mathsf{EXC} = (\mathsf{Setup}, \mathsf{Buy}, \mathsf{Sell}, \mathsf{Get})$ *with the following syntax:*

- $\mathsf{Setup}(\mathsf{xpar}, \mathsf{sk}_{\mathsf{BS}}, \mathsf{sk}_s) \to (\mathsf{xm}_1, St)$ *takes as input exchange parameters* xpar*, a secret key* $\mathsf{sk}_{\mathsf{BS}}$*, and a secret key* sk_s*, and outputs a message* xm_1 *and a state* St*.*
- $\mathsf{Buy}(\mathsf{xpar}, \mathsf{sk}_b, \mathsf{xm}_1) \to \mathsf{xm}_2$ *takes as input exchange parameters* xpar*, a secret key* sk_b*, and a message* xm_1*, and outputs a message* xm_2*.*
- $\mathsf{Sell}(St, \mathsf{xm}_2) \to \sigma_b$ *is deterministic, takes as input a state* St *and a message* xm_2*, and outputs a signature* σ_b*.*
- $\mathsf{Get}(\mathsf{xpar}, \mathsf{xm}_1, \mathsf{xm}_2, \sigma_b, \sigma_s) \to \mathsf{bsm}_2$ *is deterministic, takes as input parameters* xpar*, messages* xm_1 *and* xm_2*, and signatures* σ_b *and* σ_s*, and outputs a message* bsm_2*.*

It is required that the following completeness property holds: For all transactions tx*, messages* sn*, keys* $(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sk}_{\mathsf{BS}}) \in \mathsf{BS.Gen}(1^\lambda)$*, and all* $(\mathsf{pk}_b, \mathsf{sk}_b) \in \mathsf{SIG.Gen}(1^\lambda), (\mathsf{pk}_s, \mathsf{sk}_s) \in \mathsf{SIG.Gen}(1^\lambda)$*, we have* $b_1 = 1$ *and* $b_2 = 1$ *with probability* 1 *in the following experiment:*

$$
\begin{aligned}
&(\mathsf{bsm}_1, St) \leftarrow \mathsf{U}_1(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sn}),\\
&\mathsf{xpar} := (\mathsf{pk}_{\mathsf{BS}}, \mathsf{bsm}_1, \mathsf{pk}_b, \mathsf{pk}_s, \mathsf{tx}),\\
&(\mathsf{xm}_1, St) \leftarrow \mathsf{Setup}(\mathsf{xpar}, \mathsf{sk}_{\mathsf{BS}}, \mathsf{sk}_s),\\
&\mathsf{xm}_2 \leftarrow \mathsf{Buy}(\mathsf{xpar}, \mathsf{sk}_b, \mathsf{xm}_1),\\
&\sigma_b := \mathsf{Sell}(St, \mathsf{xm}_2),\ \sigma_s \leftarrow \mathsf{Sig}(\mathsf{sk}_s, \mathsf{tx})\\
&\mathsf{bsm}_2 := \mathsf{Get}(\mathsf{xpar}, \mathsf{xm}_1, \mathsf{xm}_2, \sigma_b, \sigma_s),\\
&\sigma_{\mathsf{BS}} \leftarrow \mathsf{U}_2(St, \mathsf{bsm}_2),\\
&b_1 := \mathsf{SIG.Ver}(\mathsf{pk}_b, \mathsf{tx}, \sigma_b),\ b_2 := \mathsf{BS.Ver}(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sn}, \sigma_{\mathsf{BS}})
\end{aligned}
$$

We require that an exchange protocol has well distributed signatures. That is, signatures on a transaction tx obtained from the exchange protocol should be distributed identically to freshly computed signatures. We postpone the formal definition of this property to our full version [34], Section 5.1.

Security. Next, we define security of such an exchange in a game-based fashion. Informally, security should ensure that the following two properties hold:

1) *Security Against Malicious Sellers:* Without learning xm_2, the seller should not be able to derive a signature on tx. The seller should only be able to derive a signature for the given transaction tx. Finally, the seller should not be able to derive a signature from which the buyer can not derive a blind signature.
2) *Security Against Malicious Buyers:* The buyer should only be able to learn blind signatures if the seller derived a valid signature σ_b. We formalize this via simulators that do not get $\mathsf{sk}_{\mathsf{BS}}$ as input. Our definition captures the intuition that the only information about $\mathsf{sk}_{\mathsf{BS}}$ that is revealed is bsm_2, and this is only revealed once the signatures σ_b, σ_s are published.

Intuitively, blindness of BS is preserved, even when running in composition with such an exchange. The reason is that the algorithms $\mathsf{Buy}, \mathsf{Get}$ that are executed by the buyer do not take the secret state St of the user U as input.

Definition 2 (Security Against Malicious Sellers)**.** *Let* $\mathsf{EXC} = (\mathsf{Setup}, \mathsf{Buy}, \mathsf{Sell}, \mathsf{Get})$ *be an exchange for* SIG *and* BS *as in Definition 1. For any algorithm* $\mathcal{A}$*, consider the following game:*

1) *Run* $\mathcal{A}$ *and obtain a public key* $\mathsf{pk}_{\mathsf{BS}}$ *and a message* sn.
2) *Run* $(\mathsf{bsm}_1, St) \leftarrow \mathsf{U}_1(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sn})$.
3) *Sample keys* $(\mathsf{pk}_b, \mathsf{sk}_b) \leftarrow \mathsf{SIG.Gen}(1^\lambda)$.
4) *Run* $\mathcal{A}$ *on input* pk_b *and* bsm_1. *Obtain* $\mathsf{pk}_s, \mathsf{tx}$*, and* xm_1 *from* $\mathcal{A}$. *Set* $\mathsf{xpar} := (\mathsf{pk}_{\mathsf{BS}}, \mathsf{bsm}_1, \mathsf{pk}_b, \mathsf{pk}_s, \mathsf{tx})$.
5) *If* $\mathsf{xm}_1 \neq \bot$*, run* $\mathsf{xm}_2 \leftarrow \mathsf{Buy}(\mathsf{xpar}, \mathsf{sk}_b, \mathsf{xm}_1)$ *and give* xm_2 *to* $\mathcal{A}$. *Otherwise, give* $\mathsf{xm}_2 := \bot$ *to* $\mathcal{A}$.
6) *Obtain* tx' *and* σ_b, σ_s *from* $\mathcal{A}$ *and run* $\mathsf{bsm}_2 := \mathsf{Get}(\mathsf{xpar}, \mathsf{xm}_1, \mathsf{xm}_2, \sigma_b, \sigma_s)$ *and* $\sigma_{\mathsf{BS}} \leftarrow \mathsf{U}_2(St, \mathsf{bsm}_2)$.
7) *If* $\mathsf{SIG.Ver}(\mathsf{pk}_b, \mathsf{tx}', \sigma_b) = 0$ *or* $\mathsf{SIG.Ver}(\mathsf{pk}_s, \mathsf{tx}', \sigma_s) = 0$*, output* 0.
8) *Output* 1 *if one of the following holds, else output* 0*:*
   a) $\mathsf{tx} \neq \mathsf{tx}'$.
   b) $\mathsf{tx} = \mathsf{tx}'$ *and* $\mathsf{xm}_2 = \bot$.
   c) $\mathsf{tx} = \mathsf{tx}'$, $\mathsf{xm}_2 \neq \bot$, *and* $\mathsf{BS.Ver}(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sn}, \sigma_{\mathsf{BS}}) = 0$.

We say that EXC *is secure against malicious sellers, if for all PPT algorithms* $\mathcal{A}$*, the probability that the above game outputs* 1 *is negligible.*

Definition 3 (Security Against Malicious Buyers)**.** *Let* $\mathsf{EXC} = (\mathsf{Setup}, \mathsf{Buy}, \mathsf{Sell}, \mathsf{Get})$ *be an exchange for* SIG *and* BS *as in Definition 1. For any algorithm* $\mathcal{A}$*, algorithms* $\mathsf{Sim}_1, \mathsf{Sim}_{RO}, \mathsf{Sim}_2, \mathsf{Sim}_3$*, which may share state, observe and program random oracles, and bit* $b \in \{0,1\}$*, consider the following game:*

1) *Sample a key pair* $(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sk}_{\mathsf{BS}}) \leftarrow \mathsf{BS.Gen}(1^\lambda)$.
2) *Let* O *be an oracle that takes as input* bsm_1 *and returns* $\mathsf{bsm}_2 \leftarrow \mathsf{BS.S}(\mathsf{sk}_{\mathsf{BS}}, \mathsf{bsm}_1)$.
3) *Run* $\mathcal{A}$ *on input* $\mathsf{pk}_{\mathsf{BS}}$ *with access to oracle* O *and an interactive oracle* O^**, which is defined as follows:*
   a) *Upon receiving a call, run* $(\mathsf{pk}_s, \mathsf{sk}_s) \leftarrow \mathsf{SIG.Gen}(1^\lambda)$ *and return* pk_s.
   b) *Upon receiving a key* pk_b*, a transaction* tx*, and* bsm_1*, set* $\mathsf{xpar} := (\mathsf{pk}_{\mathsf{BS}}, \mathsf{bsm}_1, \mathsf{pk}_b, \mathsf{pk}_s, \mathsf{tx})$. *If* $b = 0$*, run* $(\mathsf{xm}_1, St) \leftarrow \mathsf{Setup}(\mathsf{xpar}, \mathsf{sk}_{\mathsf{BS}}, \mathsf{sk}_s)$. *If* $b = 1$*, run* $\mathsf{xm}_1 \leftarrow \mathsf{Sim}_1(\mathsf{xpar}, \mathsf{sk}_s)$. *Return* xm_1.
   c) *Upon receiving* xm_2*, run* $\sigma_s \leftarrow \mathsf{SIG.Sig}(\mathsf{sk}_s, \mathsf{tx})$. *If* $b = 0$*, run* $\sigma_b := \mathsf{Sell}(St, \mathsf{xm}_2)$*, and abort if* $\mathsf{SIG.Ver}(\mathsf{pk}_b, \mathsf{tx}, \sigma_b) = 0$. *If* $b = 1$*, abort if* $\mathsf{Sim}_2(\mathsf{xm}_2) = 0$. *Otherwise, run* $\mathsf{bsm}_2 \leftarrow \mathsf{BS.S}(\mathsf{sk}_{\mathsf{BS}}, \mathsf{bsm}_1)$ *and* $\sigma_b \leftarrow \mathsf{Sim}_3(\mathsf{xm}_2, \mathsf{bsm}_2)$. *Return* σ_b, σ_s.
4) *Obtain a bit* b' *from* $\mathcal{A}$. *Output* b'.

We say that EXC *is secure against malicious buyers, if there are PPT algorithms* $\mathsf{Sim}_1, \mathsf{Sim}_{RO}, \mathsf{Sim}_2, \mathsf{Sim}_3$ *as above, such that for all PPT algorithms* $\mathcal{A}$ *the probability that the game with* $b = 0$ *outputs* 1 *and the probability that the game with* $b = 1$ *outputs* 1 *are negligibly close.*

Toy Constructions. In our full version [34], Section 5.2, we give two simple constructions of an exchange protocol. The first construction is for any unique signature scheme. The second construction is for any signature scheme supporting adaptor signatures. Their drawback is that we treat a random oracle as a circuit, which has unclear security implications.

Constructions using Cut-and-Choose. We give two concrete constructions of an exchange protocol using a cut-and-choose technique, avoiding the need to treat a random oracle as a circuit.In the first construction, the signature scheme $\mathsf{SIG} = (\mathsf{SIG.Gen}, \mathsf{SIG.Sig}, \mathsf{SIG.Ver})$ is the BLS signature scheme [30]. The second construction uses adaptor signatures for a discrete logarithm relation. It is given in our full version [34], Section 5.3. In both cases, the blind signature scheme BS is the BLS blind signature scheme . It is defined over cyclic groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ of prime order p with respective generators $g_1 \in \mathbb{G}_1, g_2 \in \mathbb{G}_2$, and $e(g_1, g_2) \in \mathbb{G}_T$, where $e\colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is a pairing. Let $\ell = \ell(\lambda)$ denote an upper bound on the bit length of messages bsm_2 sent in signing interactions of BS. We make use of random oracles $\mathsf{H}\colon \{0,1\}^* \to \{0,1\}^\ell$ and $\mathsf{H}_c\colon \{0,1\}^* \to \{0,1\}^\lambda$. The scheme is called $\mathsf{EXC}^{\mathsf{cc}}_{\mathsf{BLS}}[\mathsf{SIG}, \mathsf{BS}]$ and given in Figure 2. The security proofs are given in our full version [34], Appendix B.

Lemma 1. *Assuming the BLS signature scheme* SIG *is* EUF-CMA *secure, the exchange protocol* $\mathsf{EXC}^{\mathsf{cc}}_{\mathsf{BLS}}[\mathsf{SIG}, \mathsf{BS}]$ *is secure against malicious sellers.*

Lemma 2. *Assuming the BLS signature scheme* SIG *is* EUF-CMA *secure, the exchange protocol* $\mathsf{EXC}^{\mathsf{cc}}_{\mathsf{BLS}}[\mathsf{SIG}, \mathsf{BS}]$ *is secure against malicious buyers.*

5.2. Redeem Protocol

We define the syntax and security of the redeem protocol on the right. Later, we give concrete instantiations of it.

Setting. Informally, we consider the following scenario. Assume that a service and a user[12] are aware of a public key $\mathsf{pk}_{\mathsf{BS}}$ for a blind signature scheme BS. The service holds the corresponding secret key $\mathsf{sk}_{\mathsf{BS}}$. Further, the service published a public key pk_s for signature scheme SIG, for which it

12. In the context of Sweep-UC, the user Alice will act as the user of the redeem protocol, and the sweeper will have the role of the service.

$\underline{\mathsf{Setup}(\mathsf{xpar} = (\mathsf{pk}_{\mathsf{BS}}, \mathsf{bsm}_1, \mathsf{pk}_b, \mathsf{pk}_s, \mathsf{tx}), \mathsf{sk}_{\mathsf{BS}}, \mathsf{sk}_s)}$

// *Share* $\mathsf{bsm}_2 = \mathsf{bsm}_1^{\mathsf{sk}_{\mathsf{BS}}}$ *and* σ_s

$r_1, \ldots, r_\lambda \leftarrow\!\!\$\, \mathbb{Z}_p,\ r'_1, \ldots, r'_\lambda \leftarrow\!\!\$\, \mathbb{Z}_p$

$f(X) = \mathsf{sk}_{\mathsf{BS}} + \sum_{j=1}^{\lambda} r_j \cdot X^j \in \mathbb{Z}_p[X]$

$f'(X) = \mathsf{sk}_s + \sum_{j=1}^{\lambda} r'_j \cdot X^j \in \mathbb{Z}_p[X]$

for $j \in [2\lambda] : \mathsf{sk}_{\mathsf{BS},j} := f(j),\ \mathsf{bsm}_{2,j} \leftarrow \mathsf{S}(\mathsf{sk}_{\mathsf{BS},j}, \mathsf{bsm}_1)$

for $j \in [2\lambda] : \mathsf{sk}_{s,j} := f'(j),,\ \sigma_j \leftarrow \mathsf{SIG.Sig}(\mathsf{sk}_{s,j}, \mathsf{tx})$

for $j \in [\lambda] : \mathsf{coeff}_j := g_2^{r_j},\ \mathsf{coeff}'_j := g_2^{r'_j}$

// *Encrypt* $\mathsf{bsm}_{2,j}$ *with* σ_j

for $j \in [2\lambda] : \mathsf{ct}_j := \mathsf{H}(\sigma_j) \oplus \mathsf{bsm}_{2,j}$

// *Cut-and-choose*

$\mathsf{xm}_{1,1} := ((\mathsf{ct}_j)_{j\in[2\lambda]}, (\mathsf{coeff}_j, \mathsf{coeff}'_j)_{j\in[\lambda]})$

$b_0 \ldots b_{\lambda-1} := \mathsf{H}_c(\mathsf{xm}_{1,1}),\quad$ **for** $j \in [\lambda] : k_j := 2j - b_{j-1}$

return $(\mathsf{xm}_1 := (\mathsf{xm}_{1,1}, \mathsf{xm}_{1,2} := (\sigma_{k_j})_{j\in[\lambda]}), St := \perp)$

$\underline{\mathsf{Buy}(\mathsf{xpar}, \mathsf{sk}_b, \mathsf{xm}_1 = (\mathsf{xm}_{1,1}, \mathsf{xm}_{1,2}))}$

// *Verify cut-and-choose*

$b_0 \ldots b_{\lambda-1} := \mathsf{H}_c(\mathsf{xm}_{1,1})$

for $j \in [\lambda] :$

  $k_j := 2j - b_{j-1},\ \mathsf{pk}_{\mathsf{BS},k_j} := \mathsf{pk}_{\mathsf{BS}} \cdot \prod_{i=1}^{\lambda} (\mathsf{coeff}_i)^{k_j^i}$

  $\mathsf{pk}_{s,k_j} := \mathsf{pk}_s \cdot \prod_{i=1}^{\lambda} (\mathsf{coeff}'_i)^{k_j^i}$

  $\mathsf{bsm}_{2,k_j} := \mathsf{ct}_{k_j} \oplus \mathsf{H}(\sigma_{k_j})$

  if $e(\mathsf{bsm}_1, \mathsf{pk}_{\mathsf{BS},k_j}) \neq e(\mathsf{bsm}_{2,k_j}, g_2)$: **return** $\perp$

  if $\mathsf{SIG.Ver}(\mathsf{pk}_{s,k_j}, \mathsf{tx}, \sigma_{k_j}) = 0$: **return** $\perp$

// *Return a signature*

return $\mathsf{xm}_2 := \sigma_b \leftarrow \mathsf{SIG.Sig}(\mathsf{sk}_b, \mathsf{tx})$

$\underline{\mathsf{Sell}(St, \mathsf{xm}_2 = \sigma_b)}$

if $\mathsf{SIG.Ver}(\mathsf{pk}_b, \mathsf{tx}, \sigma_b) = 0$: **return** $\perp$

return σ_b

$\underline{\mathsf{Get}(\mathsf{xpar} = (\mathsf{pk}_{\mathsf{BS}}, \mathsf{bsm}_1, \mathsf{pk}_b, \mathsf{pk}_s, \mathsf{tx}), \mathsf{xm}_1, \mathsf{xm}_2, \sigma_b, \sigma_s)}$

$b_0 \ldots b_{\lambda-1} := \mathsf{H}_c(\mathsf{xm}_{1,1})$

// *Reconstruct all shares*

for $j \in [\lambda] :$

  $k_j := 2j - b_{j-1},\ \bar{k}_j := 2j - (1 - b_{j-1})$

  $\mathsf{bsm}_{2,k_j} := \mathsf{ct}_{k_j} \oplus \mathsf{H}(\sigma_{k_j})$

// *Find a valid share*

$w := 0$

for $j \in [\lambda] :$

  $\sigma_{\bar{k}_j} := \mathsf{reconst}_{g_1,\bar{k}_j}((0, \sigma_s), (k_i, \sigma_{k_i})_{i\in[\lambda]})$

  $\mathsf{bsm}_{2,\bar{k}_j} := \mathsf{ct}_{\bar{k}_j} \oplus \mathsf{H}(\sigma_{\bar{k}_j})$

  $\mathsf{pk}_{\mathsf{BS},\bar{k}_j} := \mathsf{pk}_{\mathsf{BS}} \cdot \prod_{i\in[\lambda]} (\mathsf{coeff}_i)^{\bar{k}_j^i}$

  if $e(\mathsf{bsm}_1, \mathsf{pk}_{\mathsf{BS},\bar{k}_j}) = e(\mathsf{bsm}_{2,\bar{k}_j}, g_2)$: $w := \bar{k}_j$

if $w = 0$: **return** $\perp$

// *Reconstruct* bsm_2

$\mathsf{bsm}_2 := \mathsf{reconst}_{g_1,0}((w, \mathsf{bsm}_{2,w}), (k_j, \mathsf{bsm}_{2,k_j})_{j\in[\lambda]})$

return bsm_2

$\underline{\mathsf{Promise}(\mathsf{rpar}, \mathsf{sk}_{\mathsf{BS}}, \mathsf{sk}_s)}$

$\sigma_s := \mathsf{H}(\mathsf{tx})^{\mathsf{sk}_s},\quad h := \hat{\mathsf{H}}(\mathsf{sn}),\quad s_0 \leftarrow\!\!\$\, \mathbb{Z}_p,\ \mathsf{ct}_0 := h^{s_0} \cdot \sigma_s$

// *Share* σ_{BS} *and* h^{s_0}

$r_1, \ldots, r_\lambda \leftarrow\!\!\$\, \mathbb{Z}_p, r'_0, \ldots, r'_\lambda \leftarrow\!\!\$\, \mathbb{Z}_p,\ \mathsf{coeff}'_0 := g_1^{s_0}$

$f(X) := \mathsf{sk}_{\mathsf{BS}} + \sum_{j=1}^{\lambda} r_j \cdot X^j$

$f'(X) := s_0 + \sum_{j=1}^{\lambda} r'_j \cdot X^j \in \mathbb{Z}_p[X]$

for $j \in [2\lambda] :\ \mathsf{sk}_j := f(j),\ s_j := f'(j),\ \sigma_j := \mathsf{H}(\mathsf{sn})^{\mathsf{sk}_j}$

for $j \in [\lambda] :\ \mathsf{coeff}_j := g_2^{r_j},\ \mathsf{coeff}'_j := g_1^{r'_j}$

// *Encrypt* h^{s_j} *with* σ_j

for $j \in [2\lambda] : \mathsf{ct}_j := \hat{\mathsf{H}}(\mathsf{sn}, \sigma_j) \cdot h^{s_j}$

// *Prove that* ct_0 *is well-formed*

$t_0, t_1 \leftarrow\!\!\$\, \mathbb{Z}_p^*,\ T_0 := h^{t_0} \cdot \mathsf{H}(\mathsf{tx})^{t_1},\ T_1 := g_1^{t_0}, T_2 := g_2^{t_1}$

$e := \mathsf{H}_p(T_0, T_1, T_2, h, \mathsf{H}(\mathsf{tx}), \mathsf{ct}_0, \mathsf{coeff}'_0, \mathsf{pk}_s)$

$\pi_0 := t_0 + e \cdot s_0,\ \pi_1 := t_1 + e \cdot \mathsf{sk}_s$

// *Cut-and-choose*

$\mathsf{prom}_1 := (\mathsf{ct}_0, (\mathsf{ct}_j)_{j\in[2\lambda]},$
$(\pi_0, \pi_1, e), \mathsf{coeff}'_0, (\mathsf{coeff}_j, \mathsf{coeff}'_j)_{j\in[\lambda]})$

$b_0 \ldots b_{\lambda-1} := \mathsf{H}_c(\mathsf{prom}_1),\quad$ **for** $j \in [\lambda] : k_j := 2j - b_{j-1}$

return $\mathsf{prom} := (\mathsf{prom}_1, \mathsf{prom}_2 := (\sigma_{k_j}, s_{k_j})_{j\in[\lambda]})$

$\underline{\mathsf{VerPromise}(\mathsf{rpar}, \mathsf{prom} = (\mathsf{prom}_1, \mathsf{prom}_2 = (\sigma_{\mathsf{BS},k_j}, s_{k_j})_{j\in[\lambda]}))}$

$h := \hat{\mathsf{H}}(\mathsf{sn}),\ b_0 \ldots b_{\lambda-1} := \mathsf{H}_c(\mathsf{prom}_1)$

// *Verify cut-and-choose*

for $j \in [\lambda] :$

  $k_j := 2j - b_{j-1},\ \mathsf{pk}_{\mathsf{BS},k_j} := \mathsf{pk}_{\mathsf{BS}} \cdot \prod_{i=1}^{\lambda} (\mathsf{coeff}_j)^{k_j^i}$

  if $\mathsf{ct}_{k_j} \neq \hat{\mathsf{H}}(\mathsf{sn}, \sigma_{k_j}) \cdot h^{s_{k_j}} \vee g_1^{s_{k_j}} \neq \prod_{i=0}^{\lambda} (\mathsf{coeff}'_j)^{k_j^i}$:

    return 0

  if $\mathsf{BS.Ver}(\mathsf{pk}_{\mathsf{BS},k_j}, \mathsf{sn}, \sigma_{k_j}) = 0$: **return** 0

// *Verify that* ct_0 *is well-formed*

$\hat{T}_0 := h^{\pi_0} \cdot \mathsf{H}(\mathsf{tx})^{\pi_1} \cdot \mathsf{ct}_0^{-e}$

$\hat{T}_1 := g_1^{\pi_0} \cdot (\mathsf{coeff}'_0)^{-e},\ \hat{T}_2 := g_2^{\pi_1} \cdot (\mathsf{pk}_s)^{-e}$

if $e \neq \mathsf{H}_p(\hat{T}_0, \hat{T}_1, \hat{T}_2, h, \mathsf{H}(\mathsf{tx}), \mathsf{ct}_0, \mathsf{coeff}'_0, \mathsf{pk}_s)$: **return** 0

return 1

$\underline{\mathsf{Redeem}(\mathsf{rpar}, \mathsf{prom} = (\mathsf{prom}_1, \mathsf{prom}_2), \sigma_{\mathsf{BS}})}$

$h := \hat{\mathsf{H}}(\mathsf{sn}),\ b_0 \ldots b_{\lambda-1} := \mathsf{H}_c(\mathsf{prom}_1)$

// *Reconstruct all shares*

for $j \in [\lambda] :$

  $k_j := 2j - b_{j-1},\ \bar{k}_j := 2j - (1 - b_{j-1}),\ h_{k_j} := h^{s_{k_j}}$

  $\sigma_{\bar{k}_j} := \mathsf{reconst}_{g_1,\bar{k}_j}((0, \sigma_{\mathsf{BS}}), (k_j, \sigma_{k_i})_{i\in[\lambda]})$

$h_{\bar{k}_j} := \mathsf{ct}_{\bar{k}_j} / \hat{\mathsf{H}}(\mathsf{sn}, \sigma_{\bar{k}_j})$

// *Try to decrypt* ct_0

for $j \in [\lambda] :$

  $h_0 := \mathsf{reconst}_{g_1,0}((\bar{k}_j, h_{\bar{k}_j}), (k_i, h_{k_i})_{i\in[\lambda]}),\ \sigma_s := \mathsf{ct}_0 / h_0$

  if $\mathsf{SIG.Ver}(\mathsf{pk}_s, \mathsf{tx}, \sigma_s) = 1$: **return** σ_s

return $\perp$

Figure 2. *Left*: The exchange protocol $\mathsf{EXC}^{\mathsf{cc}}_{\mathsf{BLS}}[\mathsf{SIG}, \mathsf{BS}] = (\mathsf{Setup}, \mathsf{Buy}, \mathsf{Sell}, \mathsf{Get})$. Here, $\mathsf{H} : \{0,1\}^* \to \{0,1\}^\ell$ and $\mathsf{H}_c : \{0,1\}^* \to \{0,1\}^\lambda$ are random oracles and and $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is a pairing. *Right*: The redeem protocol $\mathsf{RP}^{\mathsf{cc}}_{\mathsf{BLS}}[\mathsf{SIG}, \mathsf{BS}] = (\mathsf{Promise}, \mathsf{VerPromise}, \mathsf{Redeem})$. Here, $\mathsf{H} : \{0,1\}^* \to \mathbb{G}_1$, $\mathsf{H}_c : \{0,1\}^* \to \{0,1\}^\lambda$, $\mathsf{H}_p : \{0,1\}^* \to \mathbb{Z}_p^*$ and $\hat{\mathsf{H}} : \{0,1\}^* \to \mathbb{G}_1$ are random oracles. In both: SIG is the BLS signature scheme, BLS signature scheme BS is the blind BLS signature scheme.

knows a secret key sk_s. Additionally, both parties agreed on a transaction tx and a message sn. Then, the goal of both parties is to move towards a state, in which the user can use a blind signature σ_{BS} that is valid for message sn and key $\mathsf{pk}_{\mathsf{BS}}$, to obtain a signature σ_s which is valid for tx under key pk_s. This transformation of σ_{BS} into σ_s should be possible without any further interaction with the service. Moreover, the service wants to ensure that without knowing the blind signature σ_{BS}, it should not be possible to obtain σ_s. In other words, both parties want to run a protocol such that afterwards, the user is able to turn in σ_{BS} non-interactively and get a signature σ_s on the transaction tx for it.

Syntax. In our syntax, we assume that the parameters $\mathsf{rpar} := (\mathsf{pk}_{\mathsf{BS}}, \mathsf{pk}_s, \mathsf{tx}, \mathsf{sn})$ are known to both parties. The service first sends a promise message prom. This message can be verified by the user using the public key $\mathsf{pk}_{\mathsf{BS}}$. Intuitively, this verification step should guarantee that the user can be sure to obtain a valid signature σ_s from prom as soon as it knows σ_{BS}. Finally, the user can use σ_{BS} and prom to derive the signature σ_s on the transaction tx.

Definition 4 (Redeem Protocol)**.** *Let* $\mathsf{SIG} = (\mathsf{SIG.Gen}, \mathsf{SIG.Sig}, \mathsf{SIG.Ver})$ *be a digital signature scheme and* $\mathsf{BS} = (\mathsf{BS.Gen}, \mathsf{BS.S}, \mathsf{BS.U}, \mathsf{BS.Ver})$ *be a two-move blind signature scheme. A redeem protocol for* SIG *and* BS *is a tuple* $\mathsf{RP} = (\mathsf{Promise}, \mathsf{VerPromise}, \mathsf{Redeem})$ *of PPT algorithms with the following syntax:*

- $\mathsf{Promise}(\mathsf{rpar}, \mathsf{sk}_{\mathsf{BS}}, \mathsf{sk}_s) \to \mathsf{prom}$ *takes as input redeem parameters* rpar*, a secret key* $\mathsf{sk}_{\mathsf{BS}}$*, a secret key* sk_s*, and outputs a promise message* prom*.*
- $\mathsf{VerPromise}(\mathsf{rpar}, \mathsf{prom}) \to b$ *is deterministic, takes as input redeem parameters* rpar*, and a promise message* prom*, and outputs a bit* $b \in \{0, 1\}$*.*
- $\mathsf{Redeem}(\mathsf{rpar}, \mathsf{prom}, \sigma_{\mathsf{BS}}) \to \sigma_s$ *takes as input redeem parameters* rpar*, a promise message* prom*, and a signature* σ_{BS}*, and outputs a signature* σ_s*.*

We require the following completeness property: For all transactions tx*, all messages* sn*, all keys* $(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sk}_{\mathsf{BS}}) \in \mathsf{BS.Gen}(1^\lambda)$*, all* $(\mathsf{pk}_s, \mathsf{sk}_s) \in \mathsf{SIG.Gen}(1^\lambda)$*, we have* $b_1 = 1$ *and* $b_2 = 1$ *with probability* 1 *in the following experiment:*

$$
\begin{aligned}
&\mathsf{rpar} := (\mathsf{pk}_{\mathsf{BS}}, \mathsf{pk}_s, \mathsf{tx}, \mathsf{sn}),\\
&\mathsf{prom} \leftarrow \mathsf{Promise}(\mathsf{rpar}, \mathsf{sk}_{\mathsf{BS}}, \mathsf{sk}_s),\\
&\sigma_{\mathsf{BS}} \leftarrow \mathsf{BS.Sig}(\mathsf{sk}_{\mathsf{BS}}, \mathsf{sn}),\\
&\sigma_s \leftarrow \mathsf{Redeem}(\mathsf{rpar}, \mathsf{prom}, \sigma_{\mathsf{BS}}),\\
&b_1 := \mathsf{VerPromise}(\mathsf{rpar}, \mathsf{prom}),\\
&b_2 := \mathsf{SIG.Ver}(\mathsf{pk}_s, \mathsf{tx}, \sigma_s)
\end{aligned}
$$

Security. Next, we define security of such a redeem protocol in a game-based fashion. Informally, security should ensure that the following two properties hold:

1) *Security Against Malicious Users:* If a user can turn prom into a valid signature σ_s, then it must have known a valid blind signature σ_{BS}. Further, the message prom should not reveal anything about $\mathsf{sk}_{\mathsf{BS}}$.
2) *Security Against Malicious Services:* If the user gets message prom and the verification of it outputs 1, it can be sure that it can also derive a valid signature σ_s from it, using a valid blind signature σ_{BS}.

Definition 5 (Security Against Malicious Users)**.** *Suppose that* $\mathsf{RP} = (\mathsf{Promise}, \mathsf{VerPromise}, \mathsf{Redeem})$ *is a redeem protocol for* SIG *and* BS *as in Definition 4.*

Simulatability. *For any algorithm* $\mathcal{A}$*, and algorithms* Sim*,* Sim_{RO}*, which may share state, and bit* $b \in \{0, 1\}$*, consider the following game:*

1) *Sample keys* $(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sk}_{\mathsf{BS}}) \leftarrow \mathsf{BS.Gen}(1^\lambda)$ *and initialize an empty list* DSpend*.*
2) *Let* O *be an oracle that on input* sn *does the following:*
   a) *If* $\mathsf{sn} \in \mathsf{DSpend}$*, abort. Otherwise, insert* sn *into* DSpend*.*
   b) *Sample keys* $(\mathsf{pk}_s, \mathsf{sk}_s) \leftarrow \mathsf{SIG.Gen}(1^\lambda)$*, output* pk_s*.*
   c) *Receive* tx *and set* $\mathsf{rpar} := (\mathsf{pk}_{\mathsf{BS}}, \mathsf{pk}_s, \mathsf{tx}, \mathsf{sn})$*.*
   d) *If* $b = 0$*, run* $\mathsf{prom} \leftarrow \mathsf{Promise}(\mathsf{rpar}, \mathsf{sk}_{\mathsf{BS}}, \mathsf{sk}_s)$*. If* $b = 1$*, run* $\mathsf{prom} \leftarrow \mathsf{Sim}(\mathsf{rpar}, \mathsf{sk}_s)$*.*
   e) *Return* prom*.*
3) *Run* $\mathcal{A}$ *on input* $\mathsf{pk}_{\mathsf{BS}}, \mathsf{sk}_{\mathsf{BS}}$ *with access to oracle* O *and obtain a bit* b'*. During* $\mathcal{A}$*'s execution, if* $b = 0$*, provide a random oracle to* $\mathcal{A}$ *honestly via lazy sampling. If* $b = 1$*, use algorithm* Sim_{RO} *to provide the random oracle.*
4) *Output* b'*.*

We say that $(\mathsf{Sim}, \mathsf{Sim}_{RO})$ *is a simulator against malicious users for* RP*, if for all PPT algorithms* $\mathcal{A}$ *the probability that the game with* $b = 0$ *outputs* 1 *and the probability that the game with* $b = 1$ *outputs* 1 *are negligibly close.*

Extractability. *Further, for any algorithm* $\mathcal{A}$*, and algorithms* $\mathsf{Sim}, \mathsf{Sim}_{RO}, \mathsf{Ext}$*, which may share state, consider the following game:*

1) *Sample keys* $(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sk}_{\mathsf{BS}}) \leftarrow \mathsf{BS.Gen}(1^\lambda)$ *and initialize an empty list* DSpend *and set* $\mathsf{bad} := 0$*.*
2) *Let* O *be an interactive oracle that on input* sn *does the following:*
   a) *If* $\mathsf{sn} \in \mathsf{DSpend}$*, abort. Otherwise, add* sn *to* DSpend*.*
   b) *Sample keys* $(\mathsf{pk}_s, \mathsf{sk}_s) \leftarrow \mathsf{SIG.Gen}(1^\lambda)$*, output* pk_s*.*
   c) *Receive* tx *and set* $\mathsf{rpar} := (\mathsf{pk}_{\mathsf{BS}}, \mathsf{pk}_s, \mathsf{tx}, \mathsf{sn})$*.*
   d) *Run* $\mathsf{prom} \leftarrow \mathsf{Sim}(\mathsf{rpar}, \mathsf{sk}_s)$ *and output* prom*.*
   e) *Get* σ_s *as input and run* $\sigma_{\mathsf{BS}} \leftarrow \mathsf{Ext}(\mathsf{rpar}, \mathsf{sk}_s, \sigma_s)$*.*
   f) *If* $\mathsf{BS.Ver}(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sn}, \sigma_{\mathsf{BS}}) = 0$ *and* $\mathsf{SIG.Ver}(\mathsf{pk}_s, \mathsf{tx}, \sigma_s) = 1$*, set* $\mathsf{bad} := 1$*.*
3) *Run* $\mathcal{A}$ *on input* $\mathsf{pk}_{\mathsf{BS}}, \mathsf{sk}_{\mathsf{BS}}$ *with access to oracle* O*. During* $\mathcal{A}$*'s execution, use algorithm* Sim_{RO} *to provide the random oracle.*
4) *Output* bad*.*

We say that Ext *is an extractor against malicious users for* RP *and* $(\mathsf{Sim}, \mathsf{Sim}_{RO})$*, if for all PPT algorithms* $\mathcal{A}$*, the probability that the game outputs* 1 *is negligible.*

Security. *Finally, we say that* RP *is secure against malicious users, if there are algorithms* $\mathsf{Sim}, \mathsf{Sim}_{RO}, \mathsf{Ext}$ *as above, such that the pair* $(\mathsf{Sim}, \mathsf{Sim}_{RO})$ *is a simulator against malicious users for* RP *and* Ext *is a an extractor against malicious users for* RP *and* $(\mathsf{Sim}, \mathsf{Sim}_{RO})$*.*

Definition 6 (Security Against Malicious Services)**.** *Let* $\mathsf{RP} = (\mathsf{Promise}, \mathsf{VerPromise}, \mathsf{Redeem})$ *be a redeem protocol*

for SIG *and* BS *as in Definition 4. For any algorithm* $\mathcal{A}$ *and any algorithm* Ext*, consider the following game:*

1) *Run* $\mathcal{A}$ *and obtain* $\mathsf{pk}_s, \mathsf{tx}, \mathsf{sn}, \mathsf{pk}_{\mathsf{BS}}$ *and a message* prom *in return. Set* $\mathsf{rpar} := (\mathsf{pk}_{\mathsf{BS}}, \mathsf{pk}_s, \mathsf{tx}, \mathsf{sn})$.
2) *If* $\mathsf{VerPromise}(\mathsf{rpar}, \mathsf{prom}) = 0$*, return* 0.
3) *Run* $\sigma_{\mathsf{BS}} \leftarrow \mathsf{Ext}(\mathsf{rpar}, \mathsf{prom}, \mathcal{Q})$*, where* $\mathcal{Q}$ *is the list of random oracle queries that* $\mathcal{A}$ *made.*
4) *If* $\mathsf{BS.Ver}(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sn}, \sigma_{\mathsf{BS}}) = 0$*, return* 1.
5) *Compute* $\sigma_s \leftarrow \mathsf{Redeem}(\mathsf{rpar}, \mathsf{prom}, \sigma_{\mathsf{BS}})$.
6) *If* $\mathsf{SIG.Ver}(\mathsf{pk}_s, \mathsf{tx}, \sigma_s) = 0$*, return* 1 *and* 0 *otherwise.*

We say that RP *is secure against malicious services, if there is a PPT algorithm* Ext*, such that for all PPT algorithms* $\mathcal{A}$*, the probability that the game outputs* 1 *is negligible.*

Toy Construction. We generically construct a redeem protocol for any signature scheme and any unique blind signature scheme. The drawback of this scheme is that it uses proofs about relations defined by random oracles. We postpone the details to our full version [34], Section 6.2.

Constructions using Cut-and-Choose. We give two constructions of a redeem protocol without relying on proof systems that argue about the random oracle. For the first construction we assume that the signature scheme is the BLS signature scheme SIG defined over cyclic groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ of prime order p with respective generators $g_1 \in \mathbb{G}_1, g_2 \in \mathbb{G}_2$, and $e(g_1, g_2) \in \mathbb{G}_T$. The second construction works with a Schnorr signature SIG and is postponed to our full version [34], Section 6.3. In both cases we use the BLS blind signature scheme. Both signature schemes use the random oracle $\mathsf{H}\colon \{0,1\}^* \to \mathbb{G}_1$, as the oracle for the BLS and blind BLS signature. We let $\mathsf{H}_c\colon \{0,1\}^* \to \{0,1\}^\lambda$, $\hat{\mathsf{H}} : \{0,1\}^* \to \mathbb{G}_1$, and $\mathsf{H}_p\colon \{0,1\}^* \to \mathbb{Z}_p^*$ be random oracles. The resulting scheme $\mathsf{RP}^{\mathsf{cc}}_{\mathsf{BLS}}[\mathsf{SIG}, \mathsf{BS}]$ is given in Figure 2. The security proofs are given in our full version [34], Appendix C.

Lemma 3. *If* BS *has unique signatures, then* $\mathsf{RP}^{\mathsf{cc}}_{\mathsf{BLS}}[\mathsf{SIG}, \mathsf{BS}]$ *is secure against malicious services.*

Lemma 4. *If BLS signature scheme* SIG *is* EUF-CMA *secure, the* DDH *assumption holds in* $\mathbb{G}_1$*, then* $\mathsf{RP}^{\mathsf{cc}}_{\mathsf{BLS}}[\mathsf{SIG}, \mathsf{BS}]$ *is secure against malicious users.*

6. Sweep-UC: The Complete Protocol

Here, we formally present our protocol Sweep-UC that realizes $\mathcal{F}_{\mathsf{ux}}$ for a ledger functionality $\mathcal{L}^{\mathsf{SIG}}$ for signature scheme $\mathsf{SIG} = (\mathsf{SIG.Gen}, \mathsf{SIG.Sig}, \mathsf{SIG.Ver})$. The protocol is parameterized by $\mathsf{amt} \in \mathbb{N}$ and $T \in \mathbb{N}$.

Setup. Assume that $\mathsf{BS} = (\mathsf{BS.Gen}, \mathsf{BS.S}, \mathsf{BS.U}, \mathsf{BS.Ver})$ is a two-move[13] blind signature scheme. Let $\mathsf{EXC} = (\mathsf{Setup}, \mathsf{Buy}, \mathsf{Sell}, \mathsf{Get})$ be an exchange protocol and $\mathsf{RP} = (\mathsf{Promise}, \mathsf{VerPromise}, \mathsf{Redeem})$ be a redeem protocol for SIG and BS. Our protocol makes use of the functionality $\mathcal{F}_s$. Accordingly, we describe our protocol in the $(\mathcal{L}^{\mathsf{SIG}}, \mathcal{F}_s)$-hybrid model. At setup time, a key pair $(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sk}_{\mathsf{BS}}) \leftarrow \mathsf{BS.Gen}(1^\lambda)$ is generated. The sweeper $\mathcal{W}$ is initialized with $\mathsf{sk}_{\mathsf{BS}}$. All parties are initialized with the corresponding public key $\mathsf{pk}_{\mathsf{BS}}$. Further, $\mathcal{W}$ holds a secret key $\mathsf{sk}_{\mathcal{W}}$ for public key $\mathsf{pk}_{\mathcal{W}}$ for signature scheme SIG, and lists Reg, DSpend, which are initially empty.

13. We only assume two moves for simplicity of exposition. The construction can naturally be generalized to more moves.

Protocol. We now verbally describe the protocol Sweep-UC. An overview of our protocol can be found in Figure 7. The sub-protocols are given in Figures 4,5, and 6. We assume that the three parts of the protocol are executed in the correct order, i.e. first a party $\mathcal{P}$ registers, then a payment is added and then $\mathcal{P}$ gets the payment. If the parts of the protocol are called in any different order, then the execution aborts. Also, if any party expects to receive a certain message and does not receive it, the execution aborts. Finally, we assume that communication between $\mathcal{W}$ and $\mathcal{P}$ is done via a secure channel. Furthermore, we assume that EXC and RP make use of different random oracles. This can easily be achieved using proper prefixing for domain separation.

$\mathtt{Register}(\mathsf{pk}_b)$: We describe the sub-protocol as an interaction between a party $\mathcal{P}$ and the sweeper $\mathcal{W}$.

1) *Sampling a Random Nonce:* Party $\mathcal{P}$ samples a random nonce $\mathsf{sn} \leftarrow\!\!\$\, \{0,1\}^\lambda$ and sends $\mathsf{sn}, \mathsf{pk}_b$ to $\mathcal{W}$.
2) *Opening a Shared Address:* Then, $\mathcal{W}$ aborts if $\mathsf{sn} \in \mathsf{DSpend}$ or $\mathsf{pk}_b \in \mathsf{Reg}$. Otherwise, it adds these entries to the respective lists. Then, it calls $\mathcal{F}_s.\mathtt{OpenSh}(T, \mathsf{pk}_{\mathcal{W}}, \mathcal{P}, \mathsf{amt}, \mathsf{sk}_{\mathcal{W}})$. As a result, $\mathcal{W}$ obtains $(\bar{\mathsf{pk}}_{r,\mathcal{W}}, \bar{\mathsf{pk}}_{r,\mathcal{P}}, \bar{\mathsf{sk}}_{r,\mathcal{W}})$ from $\mathcal{F}_s$ and $\mathcal{P}$ obtains $(\bar{\mathsf{pk}}_{r,\mathcal{W}}, \bar{\mathsf{pk}}_{r,\mathcal{P}}, \bar{\mathsf{sk}}_{r,\mathcal{P}})$ from $\mathcal{F}_s$.
3) *Making a Promise:* Both parties $\mathcal{P}$ and $\mathcal{W}$ set $\mathsf{tx}_r := (\bar{\mathsf{pk}}_{r,\mathcal{W}}, \bar{\mathsf{pk}}_{r,\mathcal{P}}, \mathsf{pk}_b, \mathsf{amt})$. Also, both set the redeem parameters $\mathsf{rpar} := (\mathsf{pk}_{\mathsf{BS}}, \bar{\mathsf{pk}}_{r,\mathcal{W}}, \mathsf{tx}_r, \mathsf{sn})$. Then, $\mathcal{W}$ computes a promise message $\mathsf{prom} \leftarrow \mathsf{Promise}(\mathsf{rpar}, \mathsf{sk}_{\mathsf{BS}}, \bar{\mathsf{sk}}_{r,\mathcal{W}})$ and sends prom to $\mathcal{P}$.
4) *Verifying the Promise:* $\mathcal{P}$ runs $b := \mathsf{VerPromise}(\mathsf{rpar}, \mathsf{prom})$. If $b = 0$, it aborts the entire execution.

$\mathtt{AddPayment}(\mathsf{pk}_a, \mathsf{sk}_a, \mathsf{pk}_b)$: We describe the sub-protocol as an interaction between a party $\mathcal{P}$ and the sweeper $\mathcal{W}$. In this sub-protocol, $\mathcal{P}$ uses an anonymous secure channel to communicate with $\mathcal{W}$.

1) *Challenge:* Party $\mathcal{P}$ runs $(\mathsf{bsm}_1, St) \leftarrow \mathsf{BS.U}_1(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sn})$. It sends bsm_1 to $\mathcal{W}$.
2) *Opening a Shared Address:* Then, $\mathcal{P}$ calls $\mathcal{F}_s.\mathtt{OpenSh}(T, \bar{\mathsf{pk}}_a, \mathcal{W}, \mathsf{amt}, \mathsf{sk}_a)$. As a result, $\mathcal{W}$ obtains $(\bar{\mathsf{pk}}_{l,\mathcal{P}}, \bar{\mathsf{pk}}_{l,\mathcal{W}}, \bar{\mathsf{sk}}_{l,\mathcal{W}})$ and $\mathcal{P}$ obtains $(\bar{\mathsf{pk}}_{l,\mathcal{P}}, \bar{\mathsf{pk}}_{l,\mathcal{W}}, \bar{\mathsf{sk}}_{l,\mathcal{P}})$.
3) *Running the Exchange:* Both parties define a transaction $\mathsf{tx}_l := (\bar{\mathsf{pk}}_{l,\mathcal{P}}, \bar{\mathsf{pk}}_{l,\mathcal{W}}, \mathsf{pk}_{\mathcal{W}}, \mathsf{amt})$ and exchange parameters $\mathsf{xpar} := (\mathsf{pk}_{\mathsf{BS}}, \mathsf{bsm}_1, \bar{\mathsf{pk}}_{l,\mathcal{P}}, \bar{\mathsf{pk}}_{l,\mathcal{W}}, \mathsf{tx}_l)$. Then, the sweeper runs $(\mathsf{xm}_1, St) \leftarrow \mathsf{Setup}(\mathsf{xpar}, \mathsf{sk}_{\mathsf{BS}}, \bar{\mathsf{sk}}_{l,\mathcal{W}})$. It sends xm_1 to $\mathcal{P}$. Then, $\mathcal{P}$ runs $\mathsf{xm}_2 \leftarrow \mathsf{Buy}(\mathsf{xpar}, \bar{\mathsf{sk}}_{l,\mathcal{P}}, \mathsf{xm}_1)$ and sends xm_2 to $\mathcal{W}$. Then, $\mathcal{W}$ runs $\sigma_{l,\mathcal{P}} := \mathsf{Sell}(St, \mathsf{xm}_2)$. Additionally, $\mathcal{W}$ computes $\sigma_{l,\mathcal{W}} \leftarrow \mathsf{SIG.Sig}(\bar{\mathsf{sk}}_{l,\mathcal{W}}, \mathsf{tx}_l)$.
4) *Closing the Shared Address:* $\mathcal{W}$ calls $\mathcal{F}_s.\mathtt{CloseSh}(\bar{\mathsf{pk}}_{l,\mathcal{P}}, \bar{\mathsf{pk}}_{l,\mathcal{W}}, \mathsf{pk}_{\mathcal{W}}, \mathsf{amt}, \sigma_{l,\mathcal{P}}, \sigma_{l,\mathcal{W}})$. Then, $\mathcal{P}$ receives ("closedSharedAddress", $\bar{\mathsf{pk}}_{l,\mathcal{P}}, \bar{\mathsf{pk}}_{l,\mathcal{W}}, \mathsf{pk}_{\mathcal{W}}, \mathsf{amt}$,

$\sigma_{l,\mathcal{P}}, \sigma_{l,\mathcal{W}})$ from $\mathcal{F}_s$. Finally, it computes message $\mathsf{bsm}_2 := \mathsf{Get}(\mathsf{xpar}, \mathsf{xm}_1, \mathsf{xm}_2, \sigma_{l,\mathcal{P}}, \sigma_{l,\mathcal{W}})$ and the blind signature $\sigma_{\mathsf{BS}} \leftarrow \mathsf{BS.U}_2(St, \mathsf{bsm}_2)$.

$\mathtt{GetPayment}(\mathsf{pk}_b)$: With the variable names from $\mathtt{Register}(\mathsf{pk}_b)$, party $\mathcal{P}$ runs $\sigma_{r,\mathcal{W}} \leftarrow \mathsf{Redeem}(\mathsf{rpar}, \mathsf{prom}, \sigma_{\mathsf{BS}})$, where σ_{BS} was computed in $\mathtt{AddPayment}(\mathsf{pk}_a, \mathsf{sk}_a, \mathsf{pk}_b)$. It also computes $\sigma_{r,\mathcal{P}} \leftarrow \mathsf{SIG.Sig}(\bar{\mathsf{sk}}_{r,\mathcal{P}}, \mathsf{tx}_r)$. Then, it closes the shared address by calling the interface $\mathcal{F}_s.\mathtt{CloseSh}(\bar{\mathsf{pk}}_{r,\mathcal{W}}, \bar{\mathsf{pk}}_{r,\mathcal{P}}, \mathsf{pk}_b, \mathsf{amt}, \sigma_{r,\mathcal{W}}, \sigma_{r,\mathcal{P}})$. As a result, $\mathcal{W}$ receives ("closedSharedAddress", $\bar{\mathsf{pk}}_{r,\mathcal{W}}, \bar{\mathsf{pk}}_{r,\mathcal{P}}, \mathsf{pk}_b, \mathsf{amt}, \sigma_{r,\mathcal{W}}, \sigma_{r,\mathcal{P}}$) from $\mathcal{F}_s$. It removes pk_b from Reg.

Security. We informally argue why Sweep-UC satisfies security for users, security for the sweeper, and user unlinkability. A formal analysis in the UC model can be found in the full version [34], Appendix D. Security for users follows from the security of the exchange protocol and the security of the redeem protocol. Namely, there are two ways the user can lose coins when interacting with $\mathcal{W}$. First, assume the user does not get a blind signature from the interaction in the exchange protocol, although $\mathcal{W}$ is able to close the shared address. This means that the sweeper broke the security of the exchange protocol. Second, assume the user did obtain a valid blind signature using the exchange protocol but can not derive a valid signature to close the shared address related to the redeem protocol. In this case, the sweeper broke the security of the redeem protocol. Security for the sweeper can be broken if users close more shared addresses related to the redeem protocol than the sweeper closes shared addresses related to the exchange protocol. The security of the exchange protocol guarantees that users only learn a blind signature if the sweeper closes the shared address. The security of the redeem protocol guarantees that users need a blind signature to close the shared address. In a case where users stole coins from the sweeper, they would have learned more blind signatures than they obtained. Due to the usage of the list DSpend, all of these are valid for different messages, and users must have broken one-more unforgeability of BS. Finally, unlinkability follows from the blindness of BS and the use of an anonymous channel. We remark that without an anonymous channel, the atomicity of the swap would not be affected, but the sweeper could link a payment on the left and on the right simply because it is interacting with the same party, or IP address in practice.

7. Discussion

Here, we discuss practical aspects of the system, efficiency, and potential extensions of our protocol.

DoS Attacks. Note that for every user that registers anonymously, the sweeper has to freeze a certain amount of its coins for a while. Without additional measures, this can lead to a form of Denial-of-Service attacks, called griefing attacks. To mitigate these attacks, we can employ the standard blind registration based technique from [14]. This technique ensures that the sweeper only locks coins if there is a user locking the same amount of coins. This way, the attacker has to lock the same amount of coins as the sweeper to launch such a DoS attack.

Exchange Rates. We envisage a system running our protocol to announce exchange rates for every supported pair of currencies for a period of time. During this time, all transactions have to respect this rate. To ensure anonymity, the system only allows swaps of fixed denominations, e.g., one coin of currency A for x coins of currency B, where x is determined by the exchange rate.

Asymptotic Efficiency. Both the communication and computational complexity of our protocol are dominated by the exchange and redeem protocols. In terms of computation, naively looking at the pseudocode results in $O(\lambda)$ hash evaluations and pairings, but $O(\lambda^3)$ group operations in the worst-case. These are caused by λ evaluations of algorithm reconst (see Figure 2, Line 63). We can significantly reduce this to $O(\lambda^2)$ operations using preprocessing techniques, as explained in our full version [34], Appendix F. Further, cut-and-choose is naturally highly parallelizable. We are confident that there are other optimizations to further reduce the concrete number of operations. For communication, it is easy to see that $O(\lambda)$ group elements are sent over the network.

On-Chain Costs. To measure on-chain costs, the typical metric is to evaluate the transaction fee associated with all transactions that appear on-chain as part of the protocol. Sweep-UC's on-chain transactions are minimal and in line with the transactions of A^2L [14] and A^2L^+ [26] that are state of the art. On a successful execution, we have four transactions with standard signature verification scripts that go on chain. In contrast, Tumblebit (a prior work) uses HTLC scripts in its transactions, which are more expensive in terms of transaction fees than transactions with regular signature verification scripts.

Communication. In Table 3, we give estimations of the communication complexity of our protocol.For our estimation, we assume curve BLS12-381 for BLS and curve secp256k1 for Schnorr. The communication does not include communication with the chain, as this is specific to the currency that is used and should be negligible compared to the other costs. We see that for all combinations, communication cost is low, namely, less than 80 kilobytes.

Experimental Evaluation. To show efficiency in practice, we implemented an unoptimized prototype. As the running time of the protocol is dominated by the algorithms of the redeem and exchange protocols, we implemented those. Concretely, we focused on the Schnorr variant of our cut-and-choose approach with $\lambda = 128$ in combination with the BLS blind signature scheme. Other cut-and-choose variants of our algorithms should be equally practical. We based our prototype on the Chia-Network implementation of the BLS12-381 curve[14]. The Chia-Network BLS12-381 library uses C++-based shared libraries and Python binding. Additionally, we implemented the prototype to execute certain algorithm parts in parallel. We used the Python

14. See https://github.com/Chia-Network/bls-signatures

EXC.	Setup	Buy	Get (HC)	Get (WC)
	0.82	5.3	0.35	13.5
RP.	Promise	VerPromise	Redeem (HC)	Redeem (WC)
	0.53	5.16	0.21	25.5

TABLE 2. EXECUTION TIME IN SECONDS AVERAGED OVER 100 TESTS FOR OUR SCHNORR CUT-AND-CHOOSE VARIANT. FOR EXC.Get AND RP.Redeem, THE HONEST CASE (HC), AND THE WORST CASE FOR A MALICIOUS SWEEPER (WC) IS PRESENTED.

Left	Right	Comm. Left	Comm. Right	Comm. Total
BLS	BLS	43	31	74
Schnorr	BLS	33	31	64
BLS	Schnorr	43	35	78
Schnorr	Schnorr	33	35	68

TABLE 3. COMMUNICATION COMPLEXITY FOR OUR PROTOCOL Sweep-UC INSTANTIATED WITH OUR CUT-AND-CHOOSE CONSTRUCTIONS WITH PARAMETER $\lambda = 128$. COMMUNICATION IS GIVEN IN KILOBYTES.

`multiprocessing` module for this. We applied parallelism only to implement EXC.Buy and RP.VerPromise algorithms. Others can potentially only benefit from this, and we leave a further optimized implementation for future work. We evaluated our implementation on a MacbookPro with Intel i7@2.3 GHz and 16 GB RAM. The Intel i7 has four physical cores, so we used 16 workers at a time for the parallel execution. Our results are presented in Table 2. The table shows average running times over 100 tests. These results clearly indicate that our solution is practical. For example, the sweeper can set up an exchange (algorithm EXC.Setup) and create a promise (algorithm RP.Promise) in less than a second. Naturally, we expect that sweeper will be executed on a powerful server, significantly reducing this time. On the user side, the most time-consuming operations are algorithms EXC.Buy and RP.VerPromise, i.e., verifying the cut-and-choose. Both take around 5 seconds. We expect that this can be optimized further. Also, note that in the case of an honest sweeper, algorithms EXC.Get and RP.Redeem terminate early and take less time (less than a second) than for a malicious sweeper.

Redeem for Arbitrary Signatures Scheme. In Section 5.2 we presented two redeem protocols based on a cut-and-choose technique, where the signature scheme SIG was instantiated respectively using BLS and Schnorr. On the other hand, our generic redeem protocol supports any signature scheme. We will briefly discuss how to achieve the same for cut-and-choose. The idea is similar to hybrid encryption. In this regard, we will use the BLS-based redeem protocol. Recall that at the end of the protocol, one gets a BLS signature for tx that is valid with respect to public key pk_s. We will now treat this signature as a secret key for an identity-based encryption (IBE) scheme [41] and add IBE ciphertexts to the promise. This particular construction for BLS was recently proposed by Döttling et al. [42] and called signature witness encryption (SWE). The primitive they propose allows encrypting an arbitrary message, proving any statement about the message using Bulletproofs [43], and using a BLS signature as the secret witness that can be used to decrypt. Equipped with SWE we can encrypt a signature for SIG, prove that the ciphertext is consistent, and then use the BLS-based redeem protocol to redeem the witness used to decrypt the SWE.

Future Work. As our framework is modular, one can extend our results by providing exchange and redeem protocols. This includes efficiency improvements or supporting other transaction signature scheme, e.g. post-quantum schemes. Another direction for future work is to practically implement and further optimize the concrete efficiency of our protocol.

8. Conclusion

We introduced Sweep-UC, a UC secure protocol for unlinkable exchanges across different cryptocurrencies using a central, but untrusted, party called the sweeper. Our construction is modular using two new protocols as building blocks: An exchange protocol and a redeem protocol. We instantiate these protocols for currencies relying on adaptor signatures such as Schnorr or ECDSA and for currencies relying on BLS signatures, whereas previous protocols only supported the adaptor setting. In our instantiations, we avoid treating the random oracle as a circuit using a novel cut-and-choose approach. Finally, we demonstrated the practicality of our protocol.

References

[1] M. Herlihy, "Atomic cross-chain swaps," 2018.

[2] "What is atomic swap and how to implement it," https://www.axiomadev.com/blog/what-is-atomic-swap-and-how-to-implement-it/, 2019, accessed: 2023-04-11.

[3] G. Malavolta, P. Moreno-Sanchez, A. Kate, M. Maffei, and S. Ravi, "Concurrency and privacy with payment-channel networks," in *ACM CCS 2017*, B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds. ACM Press, Oct. / Nov. 2017, pp. 455–471.

[4] "Uniswap," https://uniswap.org/whitepaper.pdf, 2020, accessed: 2023-04-11.

[5] "Raiden network," https://raiden.network/, 2022, accessed: 2023-04-11.

[6] C. Baum, B. David, and T. K. Frederiksen, "P2DEX: Privacy-preserving decentralized cryptocurrency exchange," in *ACNS 21, Part I*, ser. LNCS, K. Sako and N. O. Tippenhauer, Eds., vol. 12726. Springer, Heidelberg, Jun. 2021, pp. 163–194.

[7] I. Bentov, Y. Ji, F. Zhang, L. Breidenbach, P. Daian, and A. Juels, "Tesseract: Real-time cryptocurrency exchange using trusted hardware," in *ACM CCS 2019*, L. Cavallaro, J. Kinder, X. Wang, and J. Katz, Eds. ACM Press, Nov. 2019, pp. 1521–1538.

[8] R. W. F. Lai, V. Ronge, T. Ruffing, D. Schröder, S. A. K. Thyagarajan, and J. Wang, "Omniring: Scaling private payments without trusted setup," in *ACM CCS 2019*, L. Cavallaro, J. Kinder, X. Wang, and J. Katz, Eds. ACM Press, Nov. 2019, pp. 31–48.

[9] "Understanding bitcoin fungibility," https://river.com/learn/bitcoin-fungibility/, 2022, accessed: 2023-04-11.

[10] S. A. K. Thyagarajan, G. Malavolta, and P. Moreno-Sanchez, "Universal atomic swaps: Secure exchange of coins across all blockchains," in *2022 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2022, pp. 1299–1316.

[11] A. Erwig, S. Faust, K. Hostáková, M. Maitra, and S. Riahi, "Two-party adaptor signatures from identification schemes," in *PKC 2021, Part I*, ser. LNCS, J. Garay, Ed., vol. 12710. Springer, Heidelberg, May 2021, pp. 451–480.

[12] "Chia network faq," https://www.chia.net/faq/, 2022, accessed: 2023-04-11.

[13] E. Heilman, L. Alshenibr, F. Baldimtsi, A. Scafuro, and S. Goldberg, "TumbleBit: An untrusted bitcoin-compatible anonymous payment hub," in *NDSS 2017*. The Internet Society, Feb. / Mar. 2017.

[14] E. Tairi, P. Moreno-Sanchez, and M. Maffei, "A^{2}L: Anonymous atomic locks for scalability in payment channel hubs," in *2021 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2021, pp. 1834–1851.

[15] X. Qin, S. Pan, A. Mirzaei, Z. Sui, O. Ersoy, A. Sakzad, M. F. Esgin, J. K. Liu, J. Yu, and T. H. Yuen, "BlindHub: Bitcoin-compatible privacy-preserving payment channel hubs supporting variable amounts," Cryptology ePrint Archive, Report 2022/1735, 2022, https://eprint.iacr.org/2022/1735.

[16] M. Ober, S. Katzenbeisser, and K. Hamacher, "Structure and anonymity of the bitcoin transaction graph," *Future internet*, vol. 5, no. 2, pp. 237–250, 2013.

[17] M. Santamaria Ortega, "The bitcoin transaction graph anonymity," 2013.

[18] F. Reid and M. Harrigan, "An analysis of anonymity in the bitcoin system," in *Security and privacy in social networks*. Springer, 2013, pp. 197–223.

[19] D. Ron and A. Shamir, "Quantitative analysis of the full bitcoin transaction graph," in *International Conference on Financial Cryptography and Data Security*. Springer, 2013, pp. 6–24.

[20] M. Möser, K. Soska, E. Heilman, K. Lee, H. Heffan, S. Srivastava, K. Hogan, J. Hennessey, A. Miller, A. Narayanan *et al.*, "An empirical analysis of traceability in the monero blockchain," *arXiv preprint arXiv:1704.04299*, 2017.

[21] "Digital currency donations for freedom convoy evading seizure by authorities," https://www.cbc.ca/news/canada/ottawa/freedom-convoy-cryptocurrency-asset-seizure-1.6389601, 2022, accessed: 2023-04-11.

[22] "CoinJoin - Bitcoin Forum," https://bitcointalk.org/?topic=279249, 2013, accessed: 2023-04-11.

[23] T. Ruffing, P. Moreno-Sanchez, and A. Kate, "CoinShuffle: Practical decentralized coin mixing for bitcoin," in *ESORICS 2014, Part II*, ser. LNCS, M. Kutylowski and J. Vaidya, Eds., vol. 8713. Springer, Heidelberg, Sep. 2014, pp. 345–364.

[24] ——, "P2P mixing and unlinkable bitcoin transactions," in *NDSS 2017*. The Internet Society, Feb. / Mar. 2017.

[25] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from bitcoin," in *2014 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2014, pp. 459–474.

[26] N. Glaeser, M. Maffei, G. Malavolta, P. Moreno-Sanchez, E. Tairi, and S. A. K. Thyagarajan, "Foundations of coin mixing services," in *ACM CCS 2022*, H. Yin, A. Stavrou, C. Cremers, and E. Shi, Eds. ACM Press, Nov. 2022, pp. 1259–1273.

[27] J. Groth, "Rerandomizable and replayable adaptive chosen ciphertext attack secure cryptosystems," in *TCC 2004*, ser. LNCS, M. Naor, Ed., vol. 2951. Springer, Heidelberg, Feb. 2004, pp. 152–170.

[28] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," in *42nd FOCS*. IEEE Computer Society Press, Oct. 2001, pp. 136–145.

[29] S. A. K. Thyagarajan, A. Bhat, G. Malavolta, N. Döttling, A. Kate, and D. Schröder, "Verifiable timed signatures made practical," in *ACM CCS 2020*, J. Ligatti, X. Ou, J. Katz, and G. Vigna, Eds. ACM Press, Nov. 2020, pp. 1733–1750.

[30] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the Weil pairing," in *ASIACRYPT 2001*, ser. LNCS, C. Boyd, Ed., vol. 2248. Springer, Heidelberg, Dec. 2001, pp. 514–532.

[31] D. Chaum, "Blind signatures for untraceable payments," in *CRYPTO'82*, D. Chaum, R. L. Rivest, and A. T. Sherman, Eds. Plenum Press, New York, USA, 1982, pp. 199–203.

[32] E. Heilman, F. Baldimtsi, and S. Goldberg, "Blindly signed contracts: Anonymous on-blockchain and off-blockchain bitcoin transactions," in *FC 2016 Workshops*, ser. LNCS, J. Clark, S. Meiklejohn, P. Y. A. Ryan, D. S. Wallach, M. Brenner, and K. Rohloff, Eds., vol. 9604. Springer, Heidelberg, Feb. 2016, pp. 43–60.

[33] G. Malavolta, P. Moreno-Sanchez, C. Schneidewind, A. Kate, and M. Maffei, "Anonymous multi-hop locks for blockchain scalability and interoperability," in *NDSS 2019*. The Internet Society, Feb. 2019.

[34] L. Hanzlik, J. Loss, S. A. Thyagarajan, and B. Wagner, "Sweep-UC: Swapping coins privately," Cryptology ePrint Archive, Report 2022/1605, 2022, https://eprint.iacr.org/2022/1605.

[35] A. Shamir, "How to share a secret," *Communications of the Association for Computing Machinery*, vol. 22, no. 11, pp. 612–613, Nov. 1979.

[36] J. Katz, U. Maurer, B. Tackmann, and V. Zikas, "Universally composable synchronous computation," in *TCC 2013*, ser. LNCS, A. Sahai, Ed., vol. 7785. Springer, Heidelberg, Mar. 2013, pp. 477–498.

[37] S. Dziembowski, L. Eckey, and S. Faust, "FairSwap: How to fairly exchange digital goods," in *ACM CCS 2018*, D. Lie, M. Mannan, M. Backes, and X. Wang, Eds. ACM Press, Oct. 2018, pp. 967–984.

[38] S. A. K. Thyagarajan and G. Malavolta, "Lockable signatures for blockchains: Scriptless scripts for all signatures," in *2021 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2021, pp. 937–954.

[39] R. Canetti, A. Jain, and A. Scafuro, "Practical UC security with a global random oracle," in *ACM CCS 2014*, G.-J. Ahn, M. Yung, and N. Li, Eds. ACM Press, Nov. 2014, pp. 597–608.

[40] J. Camenisch, M. Drijvers, T. Gagliardoni, A. Lehmann, and G. Neven, "The wonderful world of global random oracles," in *EUROCRYPT 2018, Part I*, ser. LNCS, J. B. Nielsen and V. Rijmen, Eds., vol. 10820. Springer, Heidelberg, Apr. / May 2018, pp. 280–312.

[41] D. Boneh and M. K. Franklin, "Identity-based encryption from the Weil pairing," in *CRYPTO 2001*, ser. LNCS, J. Kilian, Ed., vol. 2139. Springer, Heidelberg, Aug. 2001, pp. 213–229.

[42] N. Döttling, L. Hanzlik, B. Magri, and S. Wohnig, "McFly: Verifiable encryption to the future made practical," Cryptology ePrint Archive, Report 2022/433, 2022, https://eprint.iacr.org/2022/433.

[43] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell, "Bulletproofs: Short proofs for confidential transactions and more," in *2018 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2018, pp. 315–334.

Functionality $\mathcal{F}_{\mathsf{ux}}$

The functionality interacts with parties $\mathcal{P}_1, \ldots, \mathcal{P}_n, \mathcal{W}$, ideal adversary $\mathcal{S}$ and functionality $\mathcal{L}^{\mathsf{SIG}}$. It is parameterized by a digital signature scheme $\mathsf{SIG} = (\mathsf{Gen}, \mathsf{Sig}, \mathsf{Ver})$ A key $\mathsf{pk}_{\mathcal{W}}$ for party $\mathcal{W}$ is given. It is parameterized by $\mathsf{amt} \in \mathbb{N}, T \in \mathbb{N}$. It holds lists $\mathsf{Reg}, \mathsf{Pay}$.

Interface $\mathtt{Register}(\mathsf{pk}_b)$, called by $\mathcal{P}_i$:
01 Send ("register", $\mathcal{P}_i, \mathsf{pk}_b$) to $\mathcal{S}$. If $\mathcal{W}$ is corrupted, receive message m_1 from $\mathcal{S}$.
02 If $m_1 =$ "abort", send "fail" and return.
03 If $(\mathcal{P}_i, \mathsf{pk}_b)$ is already in Reg, send "failDoubleRegister" and return.
04 Call $\mathcal{L}^{\mathsf{SIG}}.\mathtt{Freeze}(\mathsf{pk}_{\mathcal{W}}, \mathsf{amt})$ and receive m in return. If $m =$ ("nofunds", $\mathsf{pk}_{\mathcal{W}}, \mathsf{amt}$), send "failNoFunds" and return.
05 Append $(\mathcal{P}_i, \mathsf{pk}_b)$ to Reg.
06 Send ("registered", $\mathcal{P}_i, \mathsf{pk}_b$) to $\mathcal{S}$. If $\mathcal{W}$ is corrupted, obtain m_2 in return. If $m_2 =$ "abort", remove $(\mathcal{P}_i, \mathsf{pk}_b)$ from Reg, send "fail" and return.
07 After T clock cycles: If the entry $(\mathcal{P}_i, \mathsf{pk}_b)$ is still in Reg, then call $\mathcal{L}^{\mathsf{SIG}}.\mathtt{Unfreeze}(\mathsf{pk}_{\mathcal{W}}, \mathsf{amt})$ and delete the entry from Reg.

Interface $\mathtt{AddPayment}(\mathsf{pk}_a, \mathsf{sk}_a, \mathsf{pk}_b)$, called by $\mathcal{P}_i$:
01 If $\mathcal{P}_i$ is not corrupted, and $(\mathcal{P}_i, \mathsf{pk}_b)$ is not in Reg, send "failNotRegistered" and return.
02 If $(\mathsf{pk}_a, \mathsf{sk}_a) \notin \mathsf{SIG.Gen}(1^\lambda)$, send "failInvalidKey" and return.
03 Send ("addPayment", pk_a) to $\mathcal{S}$.
04 Call $\mathcal{L}^{\mathsf{SIG}}.\mathtt{Freeze}(\mathsf{pk}_a, \mathsf{amt})$ and receive m in return.
05 If $m =$ ("nofunds", $\mathsf{pk}_a, \mathsf{amt}$), send "failNoFunds" and return.
06 Send ("addPaymentFreeze", pk_a) to $\mathcal{S}$ and receive m_1 in return.
07 If $m_1 =$ "abort", send "fail" and return.
08 If the message m_1 is not yet received after T clock cycles, call $\mathcal{L}^{\mathsf{SIG}}.\mathtt{Unfreeze}(\mathsf{pk}_a, \mathsf{amt})$, send "fail" and return.
09 Call $\mathcal{L}^{\mathsf{SIG}}.\mathtt{Unfreeze}(\mathsf{pk}_{\mathcal{W}}, \mathsf{amt})$.
10 Append $(\mathcal{P}_i, \mathsf{pk}_a, \mathsf{pk}_b)$ to Pay.

Interface $\mathtt{ChangePayment}(\mathsf{pk}_a, \mathsf{pk}_b, \mathsf{pk}_c)$, called by $\mathcal{S}$:
01 Search for entry $(\mathcal{P}_i, \mathsf{pk}_a, \mathsf{pk}_b)$ in Pay. If no such entry is found, send "fail" and return.
02 If party $\mathcal{P}_i$ is not corrupted, send "fail" and return.
03 Replace the entry $(\mathcal{P}_i, \mathsf{pk}_a, \mathsf{pk}_b)$ in Pay with $(\mathcal{P}_i, \mathsf{pk}_a, \mathsf{pk}_c)$.

Interface $\mathtt{GetPayment}(\mathsf{pk}_b)$, called by $\mathcal{P}_i$:
01 Send ("getPayment", $\mathcal{P}_i, \mathsf{pk}_b$) to $\mathcal{S}$.
02 If $(\mathcal{P}_i, \mathsf{pk}_b)$ is not in Reg, send "failNotRegistered" and return.
03 If there is no entry of the form $(\mathcal{P}_j, \mathsf{pk}_a, \mathsf{pk}_b)$ in Pay, send "failNoPayment" and return.
04 Remove the first entry of this form $(\mathcal{P}_j, \mathsf{pk}_a, \mathsf{pk}_b)$ from Pay and $(\mathcal{P}_i, \mathsf{pk}_b)$ from Reg.
05 Send ("gotPayment", $\mathcal{P}_i, \mathsf{pk}_b$) to $\mathcal{S}$.
06 Call $\mathcal{L}^{\mathsf{SIG}}.\mathtt{Unfreeze}(\mathsf{pk}_b, \mathsf{amt})$.

Figure 3. Ideal functionality $\mathcal{F}_{\mathsf{ux}}$ that models an unlinkable exchange.

$\mathcal{W}(\mathsf{sk}_{\mathsf{BS}}, \mathsf{sk}_{\mathcal{W}})$		$\mathcal{P}_i(\mathsf{pk}_b, \mathsf{pk}_{\mathsf{BS}})$
if $\mathsf{sn} \in \mathsf{DSpend}$: **abort**	$\xleftarrow{\mathsf{sn}, \mathsf{pk}_b}$	$\mathsf{sn} \leftarrow\!\!\$\, \{0,1\}^\lambda$
if $\mathsf{pk}_b \in \mathsf{Reg}$: **abort**		
$\mathsf{DSpend} := \mathsf{DSpend} \cup \{\mathsf{sn}\}$		
$\mathsf{Reg} := \mathsf{Reg} \cup \{\mathsf{pk}_b\}$		
$\mathcal{F}_s.\mathtt{OpenSh}(T, \mathsf{pk}_{\mathcal{W}}, \mathcal{P}, \mathsf{amt}, \mathsf{sk}_{\mathcal{W}})$		
Receive $(\bar{\mathsf{pk}}_{r,\mathcal{W}}, \bar{\mathsf{pk}}_{r,\mathcal{P}}, \bar{\mathsf{sk}}_{r,\mathcal{W}})$ **from** $\mathcal{F}_s$		**Receive** $(\bar{\mathsf{pk}}_{r,\mathcal{W}}, \bar{\mathsf{pk}}_{r,\mathcal{P}}, \bar{\mathsf{sk}}_{r,\mathcal{P}})$ **from** $\mathcal{F}_s$
$\mathsf{tx}_r := (\bar{\mathsf{pk}}_{r,\mathcal{W}}, \bar{\mathsf{pk}}_{r,\mathcal{P}}, \mathsf{pk}_b, \mathsf{amt})$		$\mathsf{tx}_r := (\bar{\mathsf{pk}}_{r,\mathcal{W}}, \bar{\mathsf{pk}}_{r,\mathcal{P}}, \mathsf{pk}_b, \mathsf{amt})$
$\mathsf{rpar} := (\mathsf{pk}_{\mathsf{BS}}, \bar{\mathsf{pk}}_{r,\mathcal{W}}, \mathsf{tx}_r, \mathsf{sn})$		$\mathsf{rpar} := (\mathsf{pk}_{\mathsf{BS}}, \bar{\mathsf{pk}}_{r,\mathcal{W}}, \mathsf{tx}_r, \mathsf{sn})$
$\mathsf{prom} \leftarrow \mathsf{Promise}(\mathsf{rpar}, \mathsf{sk}_{\mathsf{BS}}, \bar{\mathsf{sk}}_{r,\mathcal{W}})$	$\xrightarrow{\mathsf{prom}}$	$b := \mathsf{VerPromise}(\mathsf{rpar}, \mathsf{prom})$
		if $b = 0$: **abort**

Figure 4. Overview of the sub-protocol `Register` of protocol Sweep-UC. The protocol is run between the sweeper $\mathcal{W}$ and a party $\mathcal{P}_i$.

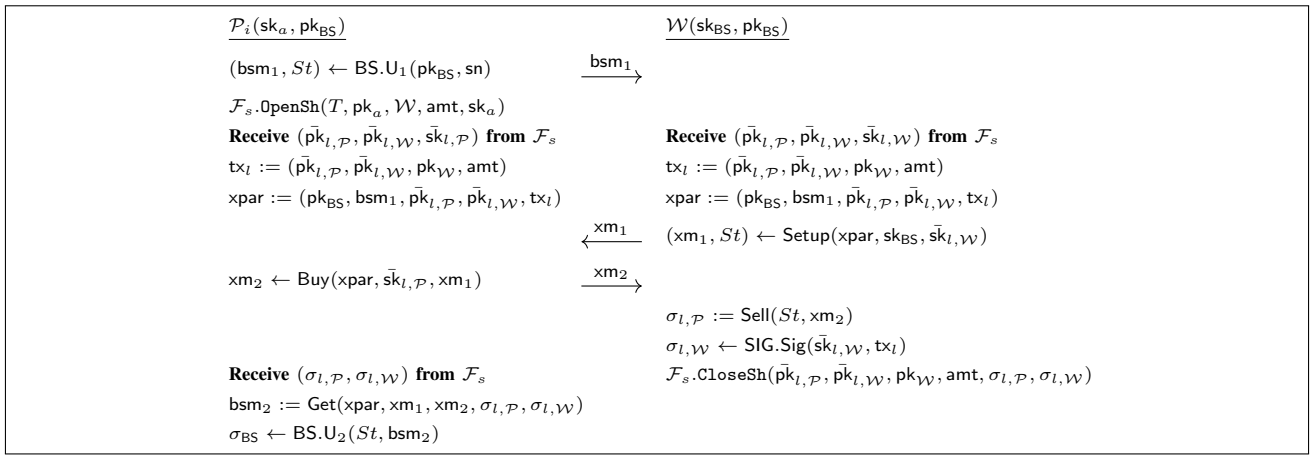

Figure 5. Overview of the sub-protocol `AddPayment` of protocol Sweep-UC. The protocol is run between the sweeper $\mathcal{W}$ and a party $\mathcal{P}_i$.

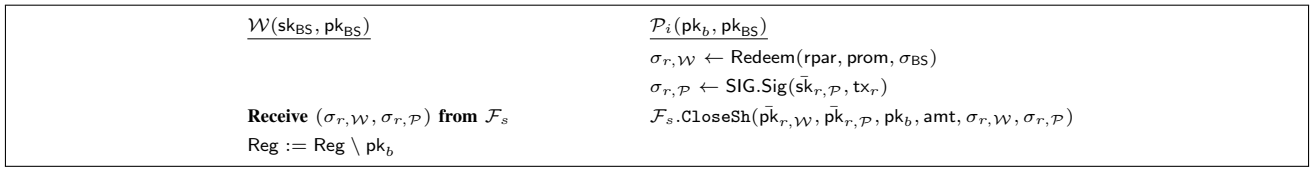

Figure 6. Overview of the sub-protocol `GetPayment` of protocol Sweep-UC. The protocol is run between the sweeper $\mathcal{W}$ and a party $\mathcal{P}_i$.

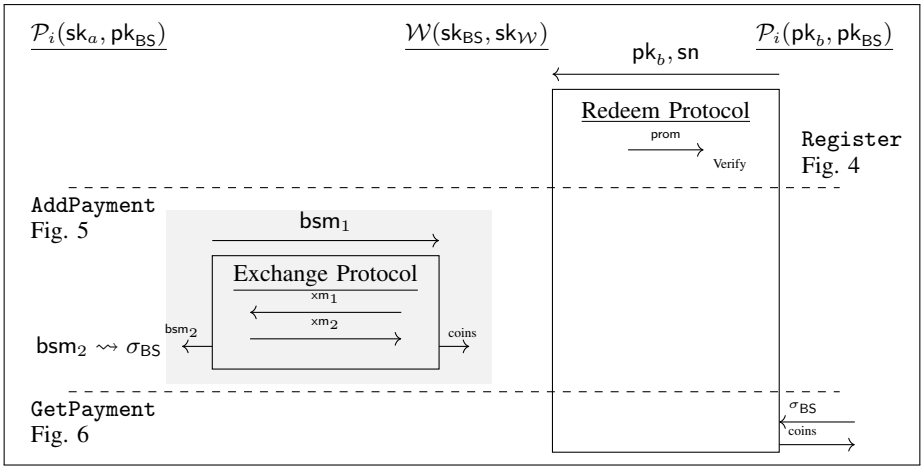

Figure 7. Overview of Sweep-UC, run between the sweeper $\mathcal{W}$ and a party $\mathcal{P}_i$. The gray area stands for an anonymous channel.

Appendix A.
Meta-Review

A.1. Summary

This paper studies the problem of coin swapping in a privacy-preserving manner. It presents two sub-protocols: 1) the exchange and 2) redeem protocols. The protocol achieves UC security without relying on general-purpose two-party computation and is universally compatible with both adaptor and unique signatures.

A.2. Scientific Contributions

- Addresses a Long-Known Issue
- Provides a Valuable Step Forward in an Established Field

A.3. Reasons for Acceptance

1) This paper presents an approach building on previous work that removes reliance on HTLCs (although still requires timelock scripts), provides privacy to users, and provides support for cryptocurrencies using unique signatures.
2) The approach is comparatively better than some of the existing schemes as referenced by the authors.
3) The paper is well organized and, for the most part, presents essential ideas clearly. Furthermore, the security models and protocols are properly explained.

A.4. Noteworthy Concerns

1) The paper discusses high on-chain costs as a limitation of previous approaches but does not evaluate how Sweep-UC compares to previous protocols in regard to on-chain costs.
2) The experimental results presented in the paper are based on simulations. While this approach has its merits, it may not accurately reflect the performance in real-world implementations. It would be beneficial to see results from a real-world implementation or at least a discussion on how the simulated results might translate to a real-world scenario. In line with that, also clarify the compatibility of Sweep-UC with popular blockchains including the discussion of how much change is required to use in for those networks.

Appendix B.
Response to the Meta-Review

We thank the reviewers of IEEE S&P 2024 for their valuable feedback, comments, and suggestions. We would like to respond to the two concerns raised in the meta-review:

1) As discussed in the paper, the on-chain costs are comparable to A^2L [14] and A^2L^+ [26], which are state of the art. How to give a more concrete estimation of the on-chain costs is not clear, as fees for transaction depend on the concrete currency at hand and vary continuously over time. For example, both works A^2L [14] and A^2L^+ [26] also do not give concrete numbers for on-chain costs.
2) We think a full-scale real-world implementation is beyond the scope of this paper, which is already quite long. However, we are happy to consider this direction for future work.