Sweep-UC: Swapping Coins Privately

Lucjan Hanzlik [1]   Julian Loss [1]   Sri AravindaKrishnan Thyagarajan [2]
Benedikt Wagner [1,3]

July 18, 2023

[1] CISPA Helmholtz Center for Information Security
{hanzlik,loss,benedikt.wagner}@cispa.de
[2] NTT Research
t.srikrishnan@gmail.com
[3] Saarland University

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.

Keywords: Atomic Swap, Unlinkable exchange, Coin Mixing, Blind Signatures

[1] Read as *Sweep Ur Coins*.

This is the full version of a paper with the same title that will appear at IEEE S&P 2024.

Contents

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) [Her18, ato19, MMK+17, uni20, rai22, BDF21, BJZ+19]. 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 [ato19, MMK+17] 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 [BJZ+19] or smart contracts [Her18, uni20, rai22, BDF21] 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 [LRR+19].
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 [fun22].

Existing Constructions. To overcome these issues, Thyagarajan, Malavolta and Moreno-Sanchez proposed *universal swaps* [TMM22]. 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 signatures [EFH+21]. This strongly limits the applicability to important systems including Monero or the Chia network [chi22]. Due to the result of Erwig et al. [EFH+21], Chia (and any other system based on unique signatures) *provably* lacks support for adaptor signatures.

Tumblebit [HAB+17], A^2L [TMM21], and BlindHub [QPM+22] 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^2L). 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* [ato19, MMK+17, Her18, uni20, rai22, BDF21], 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 [OKH13, SO13, RH13, RS13, MSH+17] have shown that mere pseudonyms do not guarantee privacy or anonymity for the users and their coins. Many instances [fre22] have showcased the importance of privacy and anonymity of coins and there has been considerable effort like CoinJoin [Coi13], CoinShuffle [RMK14, RMK17], among many others to improve coin privacy. Even new currencies with enhanced privacy were developed from scratch [LRR+19, BCG+14].

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^2L, it was found in a later work [GMM+22] that there was a gap in their security model that allowed for key recovery attacks on specific instantiations. The authors of [GMM+22] also propose fixes to A^2L called A^2L^+, but only prove security in an idealized model (the linear-only encryption (LOE) model) [Gro04] with game-based security guarantees. They also propose a version called A^2L^{UC} in the *Universal Composability (UC)* framework [Can01], which unfortunately requires heavy cryptographic tools like general-purpose two party computation (GP-2PC). This makes A^2L^{UC} inefficient for immediate use. Moreover, both A^2L^+ and A^2L^{UC} do not offer compatibility with systems lacking adaptor signature support. A more recent protocol, BlindHub [QPM+22], 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 non-interactive zero-knowledge proofs (GP-NIZK), introducing a similar efficiency penalty as for A^2L^{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?

Protocol	Scripts	Signature	UC	Amounts	Comments
Tumblebit [HAB+17]	HTLC	ECDSA	P	Fixed	
A^2L [TMM21]	SV	Adaptor	✗	Fixed	Gap in proof
A^2L^+ [GMM+22]	SV	Adaptor	✗	Fixed	need LOE model
A^2L^{UC} [GMM+22]	SV	Adaptor	✓	Fixed	need GP-2PC
BlindHub [QPM+22]	SV	Adaptor	P	Variable	need GP-NIZK, 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 [TBM+20].

1.1 Our Contribution

We answer the above question positively by presenting Sweep-UC. Like Tumblebit and A^2L (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 [TMM22, GMM+22], it does not rely on any heavy cryptographic machinery such as general-purpose 2PC. In particular, we thereby solve the challenge raised in [GMM+22]. 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 [EFH+21], and BLS [BLS01][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 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.

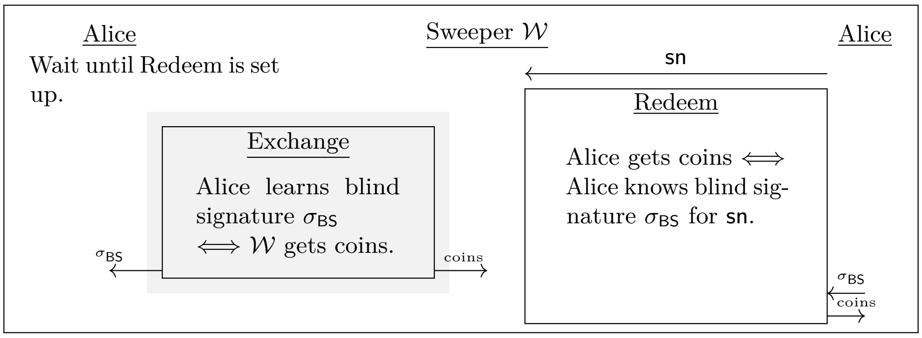

Figure 1: Informal overview of the high level structure of protocol Sweep-UC. The protocol is run between the sweeper $\mathcal{W}$ and a user Alice, using a redeem protocol and an exchange protocol as sub-protocols. The gray area represents an anonymous channel. The sweeper acts as a signer in the blind 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 $\mathsf{pk}_{\mathrm{in}}$ and $\mathsf{pk}_{\mathrm{out}}$ and the sweeper owns $\mathsf{pk}_{\mathcal{W}}$. Our goal is to coordinate two payments $\mathsf{tx}_{\mathrm{out}} = \mathsf{pk}_{a,\mathrm{out}} \to \mathsf{pk}_{\mathcal{W}}$ in currency A and $\mathsf{tx}_{\mathrm{in}} = \mathsf{pk}_{\mathcal{W}} \to \mathsf{pk}_{a,\mathrm{in}}$ in currency B[5]. To coordinate these payments, Sweep-UC implements a form of Chaum's E-Cash [Cha82], which is also the common high level structure of previous protocols [HAB+17, TMM21, GMM+22, HBG16]. In this E-Cash approach, Alice signs $\mathsf{tx}_{\mathrm{out}}$ using her secret key $\mathsf{sk}_{a,\mathrm{out}}$ (associated with $\mathsf{pk}_{a,\mathrm{out}}$) and obtains a

[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 [TBM+20].

[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.

[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.

voucher in exchange. Then, Alice can use that voucher to get a signature (valid with respect to $\mathsf{pk}_{\mathcal{W}}$) for $\mathsf{tx}_{\mathrm{in}}$. 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 $\mathsf{tx}_{\mathrm{in}}$. 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 [HBG16, MMS+19]. In exchange, the signed payment $\mathsf{tx}_{\mathrm{out}}$ 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 $\mathsf{tx}_{\mathrm{in}}$ 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.

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 $\mathsf{pk}_{\mathrm{in}}$ (right-hand side), (2) add a payment (left-hand side) by specifying $\mathsf{pk}_{\mathrm{out}}$ and refering to the registered $\mathsf{pk}_{\mathrm{in}}$, and (3) get the payment for $\mathsf{pk}_{\mathrm{in}}$ (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 [GMM+22], arises from the blindness of blind signatures. It is the reason why the UC proof of $\mathrm{A}^2\mathrm{L}$ [TMM21] 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 $\mathsf{pk}_{\mathrm{in}}$. If blindness of BS is unconditional, the simulator can not do that, as it can not extract the correct $\mathsf{pk}_{\mathrm{in}}$. For the case of computational blindness, we refer the reader to [GMM+22] for a detailed discussion. Namely, based on the common structure of known blind signature schemes, the authors of [GMM+22] elaborate that there is only little hope to get a secure system if blindness is computational. The solution taken in [GMM+22] 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 $\mathsf{pk}_{\mathrm{in}}$ 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 $\mathsf{pk}_{\mathrm{in}}$ to $\mathsf{pk}'_{\mathrm{in}}$ afterwards, this does steal coins from the sweeper, as long as she can not redeem coins (interface (3)) for *both* $\mathsf{pk}_{\mathrm{in}}$ and $\mathsf{pk}'_{\mathrm{in}}$. With this in mind, we add an additional interface `ChangePayment`, that allows the simulator to change $\mathsf{pk}_{\mathrm{in}}$ to $\mathsf{pk}'_{\mathrm{in}}$ in case Alice is corrupted and both $\mathsf{pk}_{\mathrm{in}}, \mathsf{pk}'_{\mathrm{in}}$ 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 $\mathsf{pk}_{\mathrm{in}}$, and call `ChangePayment` with the correct $\mathsf{pk}'_{\mathrm{in}}$ 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 [GMM+22].

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

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

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

(1) $\mathcal{W}$ receives ℓ coins on the left $\implies \mathcal{R}$ uses signer oracle ℓ times,

(2) $\mathcal{W}$ pays $\ell + 1$ coins on the right $\implies \mathcal{R}$ outputs $\ell + 1$ blind signatures.

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 $\mathsf{tx}_{\mathrm{in}} = \mathsf{pk}_{\mathcal{W}} \to \mathsf{pk}_{a,\mathrm{in}}$ 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

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

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 $\mathsf{tx}_{\mathrm{out}}$ 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.

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

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

[8]This can be implemented using a multi-signature address.
[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 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 [Cha82] 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: $(\mathsf{bsm}_1, St) \leftarrow \mathsf{U}_1(\mathsf{pk}, \mathsf{m})$, $\mathsf{bsm}_2 \leftarrow \mathsf{S}(\mathsf{sk}, \mathsf{bsm}_1)$, $\sigma \leftarrow \mathsf{U}_2(St, \mathsf{bsm}_2)$. 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 [Sha79] 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.

Informal 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 [Can01] 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 [HBG16, MMS+19]. Also, we consider a synchronous model of communication. This means that we implicitly assume a global clock functionality [KMTZ13], 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 [DEF18, MMS+19, TM21, TMM21, TMM22, GMM+22]. 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 [CJS14, CDG+18]. This heuristic is common practice in the area [HAB+17, MMS+19, TM21, TMM21, TMM22], 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 [DEF18, TMM22], 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 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 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 [TMM22] 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 15. 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 $x \in \{r = \mathtt{Register}, a = \mathtt{AddPayment}, g = \mathtt{GetPayment}\}$ 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 Block on the Left: Exchange Protocol

In this section, we define the first building block used for our protocol Sweep-UC, namely, an exchange protocol. We also give different instantiations of it. In the next section, we define our second building block, 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 Figures 1 and 14.

5.1 Definition of Exchange Protocols

Here, we formally define the syntax and security of the exchange protocol on the left.

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.

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). An overview of this can be found in Figure 9.

Definition 5.1 (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

$$\Pr\left[\begin{array}{c|l} & (\mathsf{bsm}_1, St) \leftarrow \mathsf{U}_1(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sn}), \\ & \mathsf{xpar} := (\mathsf{pk}_{\mathsf{BS}}, \mathsf{bsm}_1, \mathsf{pk}_b, \mathsf{pk}_s, \mathsf{tx}), \\ & (\mathsf{xm}_1, St) \leftarrow \mathsf{Setup}(\mathsf{xpar}, \mathsf{sk}_{\mathsf{BS}}, \mathsf{sk}_s), \\ b_1 = 1 & \mathsf{xm}_2 \leftarrow \mathsf{Buy}(\mathsf{xpar}, \mathsf{sk}_b, \mathsf{xm}_1), \\ \wedge \quad b_2 = 1 & \sigma_b := \mathsf{Sell}(St, \mathsf{xm}_2),\ \sigma_s \leftarrow \mathsf{Sig}(\mathsf{sk}_s, \mathsf{tx}) \\ & \mathsf{bsm}_2 := \mathsf{Get}(\mathsf{xpar}, \mathsf{xm}_1, \mathsf{xm}_2, \sigma_b, \sigma_s), \\ & \sigma_{\mathsf{BS}} \leftarrow \mathsf{U}_2(St, \mathsf{bsm}_2), \\ & b_1 := \mathsf{SIG.Ver}(\mathsf{pk}_b, \mathsf{tx}, \sigma_b), \\ & b_2 := \mathsf{BS.Ver}(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sn}, \sigma_{\mathsf{BS}}) \end{array}\right] = 1.$$

[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.

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.

Definition 5.2 (Well Distributed Signatures). Let $\mathsf{EXC} = (\mathsf{Setup}, \mathsf{Buy}, \mathsf{Sell}, \mathsf{Get})$ be an exchange for SIG and BS as in Definition 5.1. We say that EXC has well distributed signatures, if for all transactions tx, all messages sn, all keys $(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sk}_{\mathsf{BS}}) \in \mathsf{BS.Gen}(1^\lambda)$, all $(\mathsf{pk}_b, \mathsf{sk}_b) \in \mathsf{SIG.Gen}(1^\lambda)$, all $(\mathsf{pk}_s, \mathsf{sk}_s) \in \mathsf{SIG.Gen}(1^\lambda)$, we have, the following distributions $\mathcal{D}_1$ and $\mathcal{D}_2$ are the same:

$$\mathcal{D}_1 := \left\{ \begin{pmatrix} \mathsf{pk}_{\mathsf{BS}}, \mathsf{sk}_{\mathsf{BS}}, \\ \mathsf{pk}_b, \mathsf{pk}_s, \\ \mathsf{tx}, \mathsf{sn}, \sigma_b \end{pmatrix} \middle| \begin{array}{l} (\mathsf{bsm}_1, St) \leftarrow \mathsf{U}_1(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sn}), \\ \mathsf{xpar} := (\mathsf{pk}_{\mathsf{BS}}, \mathsf{bsm}_1, \mathsf{pk}_b, \mathsf{pk}_s, \mathsf{tx}), \\ (\mathsf{xm}_1, St) \leftarrow \mathsf{Setup}(\mathsf{xpar}, \mathsf{sk}_{\mathsf{BS}}, \mathsf{sk}_s), \\ \mathsf{xm}_2 \leftarrow \mathsf{Buy}(\mathsf{xpar}, \mathsf{sk}_b, \mathsf{xm}_1), \\ \sigma_b := \mathsf{Sell}(St, \mathsf{xm}_2) \end{array} \right\},$$

$$\mathcal{D}_2 := \left\{ \begin{pmatrix} \mathsf{pk}_{\mathsf{BS}}, \mathsf{sk}_{\mathsf{BS}}, \\ \mathsf{pk}_b, \mathsf{pk}_s, \\ \mathsf{tx}, \mathsf{sn}, \sigma_b \end{pmatrix} \middle| \; \sigma_b \leftarrow \mathsf{Sig}(\mathsf{sk}_b, \mathsf{tx}) \right\}.$$

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 5.3 (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 5.1. 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 5.4 (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 5.1. 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.

5.2 Toy Constructions

Here, 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. The reader may use these constructions as a starting point introducing the overall strategy.

Toy Construction for Unique Signatures. Let $\mathsf{SIG} = (\mathsf{SIG.Gen}, \mathsf{SIG.Sig}, \mathsf{SIG.Ver})$ be a signature scheme and $\mathsf{BS} = (\mathsf{BS.Gen}, \mathsf{BS.S}, \mathsf{BS.U}, \mathsf{BS.Ver})$ be a two-move blind signature scheme. We assume that SIG has unique signatures, and give a generic construction of an exchange protocol $\mathsf{EXC_u}[\mathsf{SIG}, \mathsf{BS}, \mathsf{PS}] = (\mathsf{Setup}, \mathsf{Buy}, \mathsf{Sell}, \mathsf{Get})$ for SIG and BS. The drawback of this scheme is that we have to treat a random oracle as a circuit. To this end, let $\ell_1 = \ell_1(\lambda)$ denote an upper bound on the bit length of messages bsm_2 sent in signing interactions of BS. Further, let $\ell_2 = \ell_2(\lambda)$ denote an upper bound on the number of random bits that algorithm S uses. We make use of a random oracle $\mathsf{H}\colon \{0,1\}^* \to \{0,1\}^{\ell_1}$ and a NIZK $\mathsf{PS} = (\mathsf{PProve}, \mathsf{PVer})$ with zero-knowledge simulator $\mathsf{PS.Sim}$ for the relation

$$\mathcal{R} := \left\{ (\mathsf{stmt}, \mathsf{witn}) \;\middle|\; \begin{array}{l} \mathsf{stmt} = (\mathsf{pk}_{\mathsf{BS}}, \mathsf{pk}_s, \mathsf{tx}, \mathsf{bsm}_1, \mathsf{ct}),\ \mathsf{witn} = (\sigma_s, \mathsf{sk}_{\mathsf{BS}}, \rho), \\ (\mathsf{pk}_{\mathsf{BS}}, \mathsf{sk}_{\mathsf{BS}}) \in \mathsf{BS.Gen}(1^\lambda) \wedge \mathsf{SIG.Ver}(\mathsf{pk}_s, \mathsf{tx}, \sigma_s) = 1 \\ \wedge\ \mathsf{ct} = \mathsf{H}(\sigma_s) \oplus \mathsf{BS.S}(\mathsf{sk}_{\mathsf{BS}}, \mathsf{bsm}_1; \rho) \end{array} \right\}.$$

The scheme $\mathsf{EXC_u}[\mathsf{SIG}, \mathsf{BS}, \mathsf{PS}]$ is formally presented in Figure 2. Completeness follows by inspection. As SIG has unique signatures, $\mathsf{EXC_u}[\mathsf{SIG}, \mathsf{BS}, \mathsf{PS}]$ has well distributed signatures. Security proofs are given in Appendix B.

Lemma 5.5 *If* SIG *has unique signatures,* SIG *is* $\mathsf{EUF\text{-}CMA}$ *secure, and* PS *is sound, then* $\mathsf{EXC_u}[\mathsf{SIG}, \mathsf{BS}, \mathsf{PS}]$ *is secure against malicious sellers.*

Lemma 5.6 *If* SIG *has unique signatures,* SIG *is* $\mathsf{EUF\text{-}CMA}$ *secure, and* PS *is zero-knowledge, then* $\mathsf{EXC_u}[\mathsf{SIG}, \mathsf{BS}, \mathsf{PS}]$ *is secure against malicious buyers.*

Toy Construction for Adaptor Signatures. We give a construction of an exchange protocol $\mathsf{EXC_a}[\mathsf{SIG}, \mathsf{aSIG}, \mathsf{BS}, \mathsf{PS}]$ for a signature scheme SIG supporting adaptor signatures. Concretely, let $\mathcal{R}'$ be a unique **NP**-relation that is hard relative to $\mathcal{R}'.\mathsf{Gen}$. Let $\mathsf{aSIG} = (\mathsf{PreSig}, \mathsf{Adapt}, \mathsf{PreVer}, \mathsf{Ext})$ be an adaptor signature for SIG and $\mathcal{R}'$. Let $\ell_1 = \ell_1(\lambda)$ denote an upper bound on the bit length of messages

$\underline{\mathsf{Setup}(\mathsf{xpar}, \mathsf{sk}_{\mathsf{BS}}, \mathsf{sk}_s)}$

$\rho \leftarrow_\$ \{0,1\}^{\ell_2}$
$\mathsf{bsm}_2 := \mathsf{S}(\mathsf{sk}_{\mathsf{BS}}, \mathsf{bsm}_1; \rho)$
$\sigma_s \leftarrow \mathsf{SIG.Sig}(\mathsf{sk}_s, \mathsf{tx})$
$\mathsf{ct} := \mathsf{H}(\sigma_s) \oplus \mathsf{bsm}_2$
$\mathsf{stmt} := (\mathsf{pk}_{\mathsf{BS}}, \mathsf{pk}_s, \mathsf{tx}, \mathsf{bsm}_1, \mathsf{ct})$
$\mathsf{witn} := (\sigma_s, \mathsf{sk}_{\mathsf{BS}}, \rho)$
$\pi \leftarrow \mathsf{PProve}(\mathsf{stmt}, \mathsf{witn})$
$\mathsf{xm}_1 := (\mathsf{ct}, \pi)$
return $(\mathsf{xm}_1, St := \mathsf{xpar})$

$\underline{\mathsf{Buy}(\mathsf{xpar}, \mathsf{sk}_b, \mathsf{xm}_1 = (\mathsf{ct}, \pi))}$

$\mathsf{stmt} := (\mathsf{pk}_{\mathsf{BS}}, \mathsf{pk}_s, \mathsf{tx}, \mathsf{bsm}_1, \mathsf{ct})$
if $\mathsf{PVer}(\mathsf{stmt}, \pi) = 0$: **return** $\perp$
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

$\underline{\mathsf{Get}(\mathsf{xpar}, \mathsf{xm}_1, \mathsf{xm}_2, \sigma_b, \sigma_s)}$

return $\mathsf{bsm}_2 := \mathsf{ct} \oplus \mathsf{H}(\sigma_s)$

Figure 2: The exchange protocol $\mathsf{EXC_u}[\mathsf{SIG}, \mathsf{BS}, \mathsf{PS}] = (\mathsf{Setup}, \mathsf{Buy}, \mathsf{Sell}, \mathsf{Get})$ for a unique signature scheme SIG and a blind signature scheme BS, where $\mathsf{PS} = (\mathsf{PProve}, \mathsf{PVer})$ is a NIZK for $\mathcal{R}$, and $\mathsf{H} : \{0,1\}^* \to \{0,1\}^{\ell_1}$ is a random oracle.

bsm_2 sent in signing interactions of BS. Further, let $\ell_2 = \ell_2(\lambda)$ denote an upper bound on the number of random bits that algorithm S uses. We make use of a random oracle $\mathsf{H}\colon \{0,1\}^* \to \{0,1\}^{\ell_1}$ and a NIZK $\mathsf{PS} = (\mathsf{PProve}, \mathsf{PVer})$ with zero-knowledge simulator $\mathsf{PS.Sim}$ for the relation

$$\mathcal{R} := \left\{ (\mathsf{stmt}, \mathsf{witn}) \;\middle|\; \begin{array}{l} \mathsf{stmt} = (\mathsf{pk}_{\mathsf{BS}}, \mathsf{bsm}_1, \mathsf{stmt}', \mathsf{ct}),\ \mathsf{witn} = (\mathsf{sk}_{\mathsf{BS}}, \mathsf{witn}', \rho), \\ (\mathsf{stmt}', \mathsf{witn}') \in \mathcal{R}' \wedge (\mathsf{pk}_{\mathsf{BS}}, \mathsf{sk}_{\mathsf{BS}}) \in \mathsf{BS.Gen}(1^\lambda) \\ \wedge\ \mathsf{ct} \oplus \mathsf{H}(\mathsf{witn}') = \mathsf{BS.S}(\mathsf{sk}_{\mathsf{BS}}, \mathsf{bsm}_1; \rho) \end{array} \right\}.$$

The scheme $\mathsf{EXC_a}[\mathsf{SIG}, \mathsf{aSIG}, \mathsf{BS}, \mathsf{PS}]$ is presented formally in Figure 3. Completeness follows by the uniqueness of $\mathcal{R}'$. The scheme has well distributed signatures if aSIG has well adapted signatures. We give the security proofs in Appendix B.

Lemma 5.7 *If* aSIG *is witness extractable and* $\mathsf{aEUF\text{-}CMA}$ *secure,* $\mathcal{R}'$ *is unique, and* PS *is sound, then* $\mathsf{EXC_a}[\mathsf{SIG}, \mathsf{aSIG}, \mathsf{BS}, \mathsf{PS}]$ *is secure against malicious sellers.*

Lemma 5.8 *If* aSIG *satisfies adaptability,* $\mathcal{R}'$ *is hard relative to* $\mathcal{R}'.\mathsf{Gen}$*, and* PS *is zero-knowledge, then* $\mathsf{EXC_a}[\mathsf{SIG}, \mathsf{aSIG}, \mathsf{BS}, \mathsf{PS}]$ *is secure against malicious buyers.*

$\underline{\mathsf{Setup}(\mathsf{xpar}, \mathsf{sk}_{\mathsf{BS}}, \mathsf{sk}_s)}$

$\rho \leftarrow_\$ \{0,1\}^{\ell_2}$
$\mathsf{bsm}_2 := \mathsf{S}(\mathsf{sk}_{\mathsf{BS}}, \mathsf{bsm}_1; \rho)$
$(\mathsf{stmt}', \mathsf{witn}') \leftarrow \mathcal{R}'.\mathsf{Gen}(1^\lambda)$
$\mathsf{ct} := \mathsf{H}(\mathsf{witn}') \oplus \mathsf{bsm}_2$
$\mathsf{stmt} := (\mathsf{pk}_{\mathsf{BS}}, \mathsf{bsm}_1, \mathsf{stmt}', \mathsf{ct})$
$\mathsf{witn} := (\mathsf{sk}_{\mathsf{BS}}, \mathsf{witn}', \rho)$
$\pi \leftarrow \mathsf{PProve}(\mathsf{stmt}, \mathsf{witn})$
$\mathsf{xm}_1 := (\mathsf{stmt}', \mathsf{ct}, \pi)$
$St := \mathsf{witn}'$
return (xm_1, St)

$\underline{\mathsf{Buy}(\mathsf{xpar}, \mathsf{sk}_b, \mathsf{xm}_1 = (\mathsf{stmt}', \mathsf{ct}, \pi))}$

$\mathsf{stmt} := (\mathsf{pk}_{\mathsf{BS}}, \mathsf{bsm}_1, \mathsf{stmt}', \mathsf{ct})$
if $\mathsf{PVer}(\mathsf{stmt}, \pi) = 0$: **return** $\perp$
return $\mathsf{xm}_2 := \tilde{\sigma}_b \leftarrow \mathsf{PreSig}(\mathsf{sk}_b, \mathsf{tx}, \mathsf{stmt}')$

$\underline{\mathsf{Sell}(St = \mathsf{witn}', \mathsf{xm}_2 = \tilde{\sigma}_b)}$

if $\mathsf{PreVer}(\mathsf{pk}_b, \mathsf{tx}, \mathsf{stmt}', \tilde{\sigma}_b) = 0$: **return** $\perp$
return $\sigma_b := \mathsf{Adapt}(\mathsf{pk}_b, \tilde{\sigma}_b, \mathsf{witn}')$

$\underline{\mathsf{Get}(\mathsf{xpar}, \mathsf{xm}_1, \mathsf{xm}_2, \sigma_b, \sigma_s)}$

let $\mathsf{xm}_1 = (\mathsf{stmt}', \mathsf{ct}, \pi),\ \mathsf{xm}_2 = \tilde{\sigma}_b$
$\mathsf{witn}' := \mathsf{Ext}(\tilde{\sigma}_b, \sigma_b)$
return $\mathsf{bsm}_2 := \mathsf{ct} \oplus \mathsf{H}(\mathsf{witn}')$

Figure 3: The exchange protocol $\mathsf{EXC_a}[\mathsf{SIG}, \mathsf{aSIG}, \mathsf{BS}, \mathsf{PS}] = (\mathsf{Setup}, \mathsf{Buy}, \mathsf{Sell}, \mathsf{Get})$ for a signature scheme SIG and an associated adaptor signature scheme aSIG, and a blind signature scheme BS. Here, $\mathsf{PS} = (\mathsf{PProve}, \mathsf{PVer})$ is a NIZK for $\mathcal{R}$, and $\mathsf{H} : \{0,1\}^* \to \{0,1\}^{\ell_1}$ is a random oracle.

5.3 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 both constructions, the blind signature scheme BS is the BLS blind signature scheme. For completess, we recall BLS (blind) signatures in Appendix E. 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.

Construction for BLS. In the first construction, the signature scheme $\mathsf{SIG} = (\mathsf{SIG.Gen}, \mathsf{SIG.Sig}, \mathsf{SIG.Ver})$ is the BLS signature scheme [BLS01]. 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 4. The security proofs are given in Appendix B.

Lemma 5.9 *Assume that the BLS signature scheme* SIG *is* $\mathsf{EUF\text{-}CMA}$ *secure. Then the exchange protocol* $\mathsf{EXC}^{\mathsf{cc}}_{\mathsf{BLS}}[\mathsf{SIG}, \mathsf{BS}]$ *is secure against malicious sellers.*

Lemma 5.10 *Assume that the BLS signature scheme* SIG *is* $\mathsf{EUF\text{-}CMA}$ *secure. Then the exchange protocol* $\mathsf{EXC}^{\mathsf{cc}}_{\mathsf{BLS}}[\mathsf{SIG}, \mathsf{BS}]$ *is secure against malicious buyers.*

Construction for Adaptor Signatures. We assume that the signature scheme SIG has an associated adaptor signature scheme $\mathsf{aSIG} = (\mathsf{PreSig}, \mathsf{Adapt}, \mathsf{PreVer}, \mathsf{Ext})$ for relation $\{(g^x, x) \mid x \in \mathbb{Z}_q\}$, where g is the generator of a cyclic prime order group $\mathbb{G}$ of order q. 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 $\mathsf{EXC}^{\mathsf{cc}}_{\mathsf{a}}[\mathsf{SIG}, \mathsf{aSIG}, \mathsf{BS}]$ and given in Figure 5. The security proofs are given in Appendix B.

Lemma 5.11 *Assume that* aSIG *is witness extractable and* $\mathsf{aEUF\text{-}CMA}$ *secure. Then the exchange protocol* $\mathsf{EXC}^{\mathsf{cc}}_{\mathsf{a}}[\mathsf{SIG}, \mathsf{aSIG}, \mathsf{BS}]$ *is secure against malicious sellers.*

Lemma 5.12 *Assume that* aSIG *satisfies adaptability and the* DLOG *assumption holds in* $\mathbb{G}$*. Then the exchange protocol* $\mathsf{EXC}^{\mathsf{cc}}_{\mathsf{a}}[\mathsf{SIG}, \mathsf{aSIG}, \mathsf{BS}]$ *is secure against malicious buyers.*

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

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

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

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

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

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

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

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

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

$//$ *Cut-and-choose*

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

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

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

$\underline{\mathsf{Buy}(\mathsf{xpar} = (\mathsf{pk}_{\mathsf{BS}}, \mathsf{bsm}_1, \mathsf{pk}_b, \mathsf{pk}_s, \mathsf{tx}), \mathsf{sk}_b, \mathsf{xm}_1 = (\mathsf{xm}_{1,1}, \mathsf{xm}_{1,2}))}$

$//$ *Verify cut-and-choose*

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

$\textbf{for } j \in [\lambda] :$

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

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

$\quad \textbf{if } e(\mathsf{bsm}_1, \mathsf{pk}_{\mathsf{BS},k_j}) \neq e(\mathsf{bsm}_{2,k_j}, g_2) \vee \mathsf{SIG.Ver}(\mathsf{pk}_{s,k_j}, \mathsf{tx}, \sigma_{k_j}) = 0 : \textbf{return } \bot$

$//$ *Return a signature*

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

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

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

$\textbf{return } \sigma_b$

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

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

$//$ *Reconstruct all shares*

$\textbf{for } j \in [\lambda] :\ k_j := 2j - b_{j-1},\ \bar{k}_j := 2j - (1 - b_{j-1}),\ \mathsf{bsm}_{2,k_j} := \mathsf{ct}_{k_j} \oplus \mathsf{H}(\sigma_{k_j})$

$//$ *Find a valid share*

$w := 0$

$\textbf{for } j \in [\lambda] :$

$\quad \sigma_{\bar{k}_j} := \mathsf{reconst}_{g_1,\bar{k}_j}((0, \sigma_s), (k_i, \sigma_{k_i})_{i\in[\lambda]}),\ \mathsf{bsm}_{2,\bar{k}_j} := \mathsf{ct}_{\bar{k}_j} \oplus \mathsf{H}(\sigma_{\bar{k}_j})$

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

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

$\textbf{if } w = 0 : \textbf{return } \bot$

$//$ *Reconstruct* bsm_2

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

Figure 4: The exchange protocol $\mathsf{EXC}^{\mathsf{cc}}_{\mathsf{BLS}}[\mathsf{SIG}, \mathsf{BS}] = (\mathsf{Setup}, \mathsf{Buy}, \mathsf{Sell}, \mathsf{Get})$ for BLS signature scheme SIG, and blind BLS signature scheme BS. Here, $\mathsf{H} : \{0,1\}^* \to \{0,1\}^\ell$ and $\mathsf{H}_c : \{0,1\}^* \to \{0,1\}^\lambda$ are random oracles and $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is a pairing.

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

$y \leftarrow_\$ \mathbb{Z}_q,\ Y := g^y$

// *Share* bsm_2 *and* y

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

$f(X) := \mathsf{sk_{BS}} + \sum_{j=1}^{\lambda} r_j \cdot X^j \in \mathbb{Z}_p[X],\ f'(X) := y + \sum_{j=1}^{\lambda} r'_j \cdot X^j \in \mathbb{Z}_q[X]$

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

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

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

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

// *Cut-and-choose*

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

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

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

$\underline{\mathsf{Buy}(\mathsf{xpar} = (\mathsf{pk_{BS}}, \mathsf{bsm}_1, \mathsf{pk}_b, \mathsf{pk}_s, \mathsf{tx}), \mathsf{sk}_b, \mathsf{xm}_1 = (\mathsf{xm}_{1,1}, \mathsf{xm}_{1,2}))}$

// *Verify cut-and-choose*

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

$\textbf{for } j \in [\lambda] :$

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

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

$\quad \textbf{if } e(\mathsf{bsm}_1, \mathsf{pk}_{\mathsf{BS},k_j}) \neq e(\mathsf{bsm}_{2,k_j}, g_2) \vee Y_{k_j} \neq g^{y_{k_j}} : \textbf{return } \bot$

// *Return a pre-signature for* Y

$\textbf{return } \mathsf{xm}_2 := \tilde{\sigma}_b \leftarrow \mathsf{PreSig}(\mathsf{sk}_b, \mathsf{tx}, Y)$

$\underline{\mathsf{Sell}(St = y, \mathsf{xm}_2 = \tilde{\sigma}_b)}$

$\textbf{if } \mathsf{PreVer}(\mathsf{pk}_b, \mathsf{tx}, g^y, \tilde{\sigma}_b) = 0 : \textbf{return } \bot$

$\textbf{return } \sigma_b := \mathsf{Adapt}(\mathsf{pk}_b, \tilde{\sigma}_b, y)$

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

$y := \mathsf{Ext}(\tilde{\sigma}_b, \sigma_b),\ b_0 \ldots b_{\lambda-1} := \mathsf{H}_c(\mathsf{xm}_{1,1})$

// *Reconstruct all shares*

$\textbf{for } j \in [\lambda] :\ k_j := 2j - b_{j-1},\ \bar{k}_j := 2j - (1 - b_{j-1}),\ \mathsf{bsm}_{2,k_j} := \mathsf{ct}_{k_j} \oplus \mathsf{H}(y_{k_j})$

$f'(X) := \mathsf{reconst}_q((0, y), (k_j, y_{k_j})_{j \in [\lambda]})$

// *Find a valid share*

$w := 0$

$\textbf{for } j \in [\lambda] :$

$\quad y_{\bar{k}_j} := f'(\bar{k}_j),\ \mathsf{bsm}_{2,\bar{k}_j} := \mathsf{ct}_{\bar{k}_j} \oplus \mathsf{H}(y_{\bar{k}_j})$

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

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

$\textbf{if } w = 0 : \textbf{return } \bot$

// *Reconstruct* bsm_2

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

Figure 5: The exchange protocol $\mathsf{EXC}_{\mathsf{a}}^{\mathsf{cc}}[\mathsf{SIG}, \mathsf{aSIG}, \mathsf{BS}] = (\mathsf{Setup}, \mathsf{Buy}, \mathsf{Sell}, \mathsf{Get})$ for a signature scheme SIG and an associated adaptor signature scheme aSIG, and blind BLS signature scheme BS. Here, $\mathsf{H} : \{0,1\}^* \to \{0,1\}^\ell$ and $\mathsf{H}_c : \{0,1\}^* \to \{0,1\}^\lambda$ are random oracles and $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is a pairing.

6 Building Block on the Right: Redeem Protocol

In this section, we introduce the second building block for our protocol Sweep-UC, namely, a redeem protocol. With a redeem protocol, a user can turn a blind signature in to get a signed transaction from the sweeper.

6.1 Definition of Redeem Protocols

We define the syntax and security of the redeem protocol on the right.

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 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. An overview of this can be found in Figure 10.

Definition 6.1 (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

$$\Pr\left[\begin{array}{c} b_1 = 1 \\ \wedge \quad b_2 = 1 \end{array} \middle| \begin{array}{l} \mathsf{rpar} := (\mathsf{pk}_{\mathsf{BS}}, \mathsf{pk}_s, \mathsf{tx}, \mathsf{sn}), \mathsf{prom} \leftarrow \mathsf{Promise}(\mathsf{rpar}, \mathsf{sk}_{\mathsf{BS}}, \mathsf{sk}_s), \\ \sigma_{\mathsf{BS}} \leftarrow \mathsf{BS.Sig}(\mathsf{sk}_{\mathsf{BS}}, \mathsf{sn}), \ \sigma_s \leftarrow \mathsf{Redeem}(\mathsf{rpar}, \mathsf{prom}, \sigma_{\mathsf{BS}}), \\ b_1 := \mathsf{VerPromise}(\mathsf{rpar}, \mathsf{prom}), \ b_2 := \mathsf{SIG.Ver}(\mathsf{pk}_s, \mathsf{tx}, \sigma_s) \end{array}\right] = 1.$$

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}.

[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.

Definition 6.2 (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 6.1.
Simulatability. For any algorithm $\mathcal{A}$, and algorithms $\mathsf{Sim}, \mathsf{Sim}_{RO}$, 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 $\mathsf{Sim}, \mathsf{Sim}_{RO}$, 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 6.3 (Security Against Malicious Services). Let $\mathsf{RP} = (\mathsf{Promise}, \mathsf{VerPromise}, \mathsf{Redeem})$ be a redeem protocol for SIG and BS as in Definition 6.1. 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 $\mathsf{BS.Ver}(\mathsf{pk_{BS}}, \mathsf{sn}, \sigma_{\mathsf{BS}}) = 0$, 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.

6.2 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. Consider an arbitrary signature scheme $\mathsf{SIG} = (\mathsf{SIG.Gen}, \mathsf{SIG.Sig}, \mathsf{SIG.Ver})$ and a blind signature scheme $\mathsf{BS} = (\mathsf{BS.Gen}, \mathsf{BS.S}, \mathsf{BS.U}, \mathsf{BS.Ver})$ with unique signatures. From that, we construct a redeem protocol $\mathsf{RP}[\mathsf{SIG}, \mathsf{BS}, \mathsf{PS}] = (\mathsf{Promise}, \mathsf{VerPromise}, \mathsf{Redeem})$ for SIG and BS. To this end, assume that signatures of SIG are elements of $\{0,1\}^\ell$ for some $\ell = \ell(\lambda)$. Let $\mathsf{H} : \{0,1\}^* \to \{0,1\}^\ell$ be a random oracle. We make use of a NIZK $\mathsf{PS} = (\mathsf{PProve}, \mathsf{PVer})$ with zero-knowledge simulator PS.Sim for the relation

$$\mathcal{R} := \left\{ (\mathsf{stmt}, \mathsf{witn}) \;\middle|\; \begin{array}{l} \mathsf{stmt} = (\mathsf{pk_{BS}}, \mathsf{pk}_s, \mathsf{tx}, \mathsf{sn}, \mathsf{ct}),\ \mathsf{witn} = \sigma_{\mathsf{BS}}, \\ \mathsf{BS.Ver}(\mathsf{pk_{BS}}, \mathsf{sn}, \sigma_{\mathsf{BS}}) = 1 \\ \wedge\ \mathsf{SIG.Ver}(\mathsf{pk}_s, \mathsf{tx}, \mathsf{ct} \oplus \mathsf{H}(\mathsf{sn}, \sigma_{\mathsf{BS}})) = 1 \end{array} \right\}.$$

The protocol is presented in Figure 6. Completeness follows from the uniqueness of BS. Security proofs are given in Appendix C.

Lemma 6.4 *If* BS *has unique signatures,* SIG *is smooth and* PS *is sound, then* RP[SIG, BS, PS] *is secure against malicious services.*

Lemma 6.5 *Assume that* PS *is zero-knowledge and* SIG *is* EUF-CMA *secure. Then* RP[SIG, BS, PS] *is secure against malicious users.*

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

$\sigma_{\mathsf{BS}} \leftarrow \mathsf{BS.Sig}(\mathsf{sk_{BS}}, \mathsf{sn})$
$\sigma_s \leftarrow \mathsf{SIG.Sig}(\mathsf{sk}_s, \mathsf{tx})$
$\mathsf{ct} := \mathsf{H}(\mathsf{sn}, \sigma_{\mathsf{BS}}) \oplus \sigma_s$
$\mathsf{stmt} := (\mathsf{pk_{BS}}, \mathsf{pk}_s, \mathsf{tx}, \mathsf{sn}, \mathsf{ct})$
$\pi \leftarrow \mathsf{PProve}(\mathsf{stmt}, \sigma_{\mathsf{BS}})$
return $\mathsf{prom} := (\mathsf{ct}, \pi)$

$\mathsf{VerPromise}(\mathsf{rpar}, \mathsf{prom} = (\mathsf{ct}, \pi))$

$\mathsf{stmt} := (\mathsf{pk_{BS}}, \mathsf{pk}_s, \mathsf{tx}, \mathsf{sn}, \mathsf{ct})$
return $\mathsf{PVer}(\mathsf{stmt}, \pi)$

$\mathsf{Redeem}(\mathsf{rpar}, \mathsf{prom} = (\mathsf{ct}, \pi), \sigma_{\mathsf{BS}})$

return $\sigma_s := \mathsf{ct} \oplus \mathsf{H}(\mathsf{sn}, \sigma_{\mathsf{BS}})$

Figure 6: The redeem protocol $\mathsf{RP}[\mathsf{SIG}, \mathsf{BS}, \mathsf{PS}] = (\mathsf{Promise}, \mathsf{VerPromise}, \mathsf{Redeem})$ for a signature scheme SIG and a blind signature scheme BS, where $\mathsf{PS} = (\mathsf{PProve}, \mathsf{PVer})$ is a NIZK for $\mathcal{R}$ and $\mathsf{H} : \{0,1\}^* \to \{0,1\}^\ell$ and is a random oracle.

6.3 Constructions using Cut-and-Choose

We give two concrete constructions of a redeem protocol using a cut-and-choose technique. This avoids the need to treat a random oracle as a circuit. For the first construction we assume that the signature scheme is BLS [BLS01], and for the second construction we assume it is Schnorr [Sch91]. As for our exchange protocols, we assume that the blind signature scheme BS is the BLS blind signature scheme, see Appendix E. 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. Both constructions use the random oracle $\mathsf{H} \colon \{0,1\}^* \to \mathbb{G}_1$, as the oracle for the BLS and blind BLS signature.

Construction for BLS. For our first construction, we assume that the signature scheme is the BLS signature scheme SIG defined over the same groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ as the BLS blind signature scheme that is used. 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 7. The security proofs are given in Appendix C.

Lemma 6.6 *If* BS *has unique signatures, then* $\mathsf{RP}^{\mathsf{cc}}_{\mathsf{BLS}}[\mathsf{SIG}, \mathsf{BS}]$ *is secure against malicious services.*

Lemma 6.7 *If BLS signature scheme* SIG *is* $\mathsf{EUF\text{-}CMA}$ *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.*

Construction for Schnorr. We give a construction of a redeem protocol for a Schnorr signature SIG defined over cyclic group $\mathbb{G}$ with generator g of prime order q. To recall, the random oracle $\mathsf{H} : \{0,1\}^* \to \mathbb{G}_1$ is the oracle for the blind BLS signature. Moreover, we let $\mathsf{H}_c : \{0,1\}^* \to \{0,1\}^\lambda$, $\mathsf{H}_q : \{0,1\}^* \to \mathbb{Z}_q^*$ and $\hat{\mathsf{H}}_q : \{0,1\}^* \to \mathbb{Z}_q$ be random oracles. The resulting scheme $\mathsf{RP}^{\mathsf{cc}}_{\mathsf{Schn}}[\mathsf{SIG}, \mathsf{BS}]$ is given in Figure 8. The security proofs are given in Appendix C.

Lemma 6.8 *If* BS *has unique signatures, then* $\mathsf{RP}^{\mathsf{cc}}_{\mathsf{Schn}}[\mathsf{SIG}, \mathsf{BS}]$ *is secure against malicious services.*

Lemma 6.9 *If the Schnorr signature scheme* SIG *is* $\mathsf{sEUF\text{-}CMA}$ *secure, and the* DLOG *assumption holds in* $\mathbb{G}$*, then* $\mathsf{RP}^{\mathsf{cc}}_{\mathsf{Schn}}[\mathsf{SIG}, \mathsf{BS}]$ *is secure against malicious users.*

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

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

// *Share* σ_{BS} *and* h^{s_0}

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

$f(X) := \mathsf{sk}_{\mathsf{BS}} + \sum_{j=1}^{\lambda} r_j \cdot X^j,\quad f'(X) := s_0 + \sum_{j=1}^{\lambda} r'_j \cdot X^j \in \mathbb{Z}_p[X]$

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*

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

$b_0 \ldots b_{\lambda-1} := \mathsf{H}_c(\mathsf{prom}_1)$, **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 7: The redeem protocol $\mathsf{RP}^{\mathsf{cc}}_{\mathsf{BLS}}[\mathsf{SIG}, \mathsf{BS}] = (\mathsf{Promise}, \mathsf{VerPromise}, \mathsf{Redeem})$ for the BLS signature scheme SIG and the blind BLS signature scheme BS. Here, $\mathsf{H} : \{0,1\}^* \rightarrow \mathbb{G}_1$, $\mathsf{H}_c : \{0,1\}^* \rightarrow \{0,1\}^\lambda$, $\mathsf{H}_p : \{0,1\}^* \rightarrow \mathbb{Z}_p^*$ and $\hat{\mathsf{H}} : \{0,1\}^* \rightarrow \mathbb{G}_1$ are random oracles.

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

// Compute Schnorr signature

$k \leftarrow\!\!\$\ \mathbb{Z}_q^*,\ T := g^k,\ e := \mathsf{H}_q(T, \mathsf{tx}),\ s := k - e \cdot \mathsf{sk}_s$

// Share σ_{BS} and s

$r_1, \ldots, r_\lambda \leftarrow\!\!\$\ \mathbb{Z}_p, r'_1, \ldots, r'_\lambda \leftarrow\!\!\$\ \mathbb{Z}_q,\ \mathsf{coeff}'_0 := g^s$

$f(X) = \mathsf{sk}_{\mathsf{BS}} + \sum_{j=1}^{\lambda} r_j \cdot X^j \in \mathbb{Z}_p[X],\quad f'(X) = s + \sum_{j=1}^{\lambda} r'_j \cdot X^j \in \mathbb{Z}_q[X]$

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

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

// Encrypt s_j with σ_j

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

// Cut-and-choose

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

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

for $j \in [\lambda] : k_j := 2j - b_{j-1}$

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

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

// Verify cut-and-choose

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

for $j \in [\lambda]$:

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

  if $\mathsf{ct}_{k_j} \neq \hat{\mathsf{H}}_q(\mathsf{sn}, \sigma_{k_j}) \oplus s_{k_j} \vee g^{s_{k_j}} \neq \prod_{i=0}^{\lambda} (\mathsf{coeff}'_j)^{k_j^i}$: **return** 0

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

// Verify Schnorr signature in the exponent

$T := \mathsf{coeff}'_0 \cdot (\mathsf{pk}_s)^e$

if $e \neq \mathsf{H}_q(T, \mathsf{tx})$: **return** 0

return 1

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

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

// Reconstruct all shares

for $j \in [\lambda]$:

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

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

  $s_{\bar{k}_j} := \mathsf{ct}_{\bar{k}_j} \oplus \hat{\mathsf{H}}_q(\mathsf{sn}, \sigma_{\bar{k}_j})$

// Try to find correct s

for $j \in [\lambda]$:

  $s := \mathsf{reconst}_q((\bar{k}_j, s_{\bar{k}_j}), (k_i, s_{k_i})_{i \in [\lambda]})$

  if $\mathsf{coeff}'_0 = g^s$: **return** $\sigma_s := (s, e)$

return $\bot$

Figure 8: The cut-and-choose redeem protocol $\mathsf{RP}^{\mathsf{cc}}_{\mathsf{Schn}}[\mathsf{SIG}, \mathsf{BS}] = (\mathsf{Promise}, \mathsf{VerPromise}, \mathsf{Redeem})$ for Schnorr signature SIG and the blind BLS signature scheme BS. Here, $\mathsf{H} : \{0,1\}^* \to \mathbb{G}_1$, $\mathsf{H}_c : \{0,1\}^* \to \{0,1\}^\lambda$, $\mathsf{H}_q : \{0,1\}^* \to \mathbb{Z}_q^*$ and $\hat{\mathsf{H}}_q : \{0,1\}^* \to \mathbb{Z}_q$ are random oracles.

7 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 $(\mathsf{pk_{BS}}, \mathsf{sk_{BS}}) \leftarrow \mathsf{BS.Gen}(1^\lambda)$ is generated. The sweeper $\mathcal{W}$ is initialized with $\mathsf{sk_{BS}}$. All parties are initialized with the corresponding public key $\mathsf{pk_{BS}}$. Further, $\mathcal{W}$ holds a secret key $\mathsf{sk}_{\mathcal{W}}$ for public key $\mathsf{pk}_{\mathcal{W}}$ for signature scheme SIG, and lists $\mathsf{Reg}, \mathsf{DSpend}$, which are initially empty.

Protocol. We now verbally describe the protocol Sweep-UC. An overview of our protocol can be found in Figure 14. The sub-protocols are given in Figures 11,12, and 13. 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 $\mathsf{rpar} := (\mathsf{pk_{BS}}, \bar{\mathsf{pk}}_{r,\mathcal{W}}, \mathsf{tx}_r, \mathsf{sn})$. Then, $\mathcal{W}$ computes a promise message $\mathsf{prom} \leftarrow \mathsf{Promise}(\mathsf{rpar}, \mathsf{sk_{BS}}, \bar{\mathsf{sk}}_{r,\mathcal{W}})$ 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 $(\mathsf{bsm}_1, St) \leftarrow \mathsf{BS.U}_1(\mathsf{pk_{BS}}, \mathsf{sn})$. It sends bsm_1 to $\mathcal{W}$.
2. *Opening a Shared Address:* Then, $\mathcal{P}$ calls $\mathcal{F}_s.\mathtt{OpenSh}(T, \mathsf{pk}_a, \mathcal{W}, \mathsf{amt}, \mathsf{sk}_a)$. 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 $\mathsf{xpar} := (\mathsf{pk_{BS}}, \mathsf{bsm}_1, \bar{\mathsf{pk}}_{l,\mathcal{P}}, \bar{\mathsf{pk}}_{l,\mathcal{W}}, \mathsf{tx}_l)$. Then, the sweeper runs $(\mathsf{xm}_1, St) \leftarrow \mathsf{Setup}(\mathsf{xpar}, \mathsf{sk_{BS}}, \bar{\mathsf{sk}}_{l,\mathcal{W}})$. 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.

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

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 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.

8 Discussion

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

8.1 Practical Considerations

We discuss how to deal with and Denial-of-Service attacks and dynamic exchange rates.

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 [TMM21]. 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.

8.2 Efficiency Evaluation

Here, we focus on the efficiency of our protocol Sweep-UC.

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 $\mathsf{reconst}$ (see Figure 7, Line 25). We can significantly reduce this to $O(\lambda^2)$ operations using preprocessing techniques, as explained in 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 $\mathrm{A}^2\mathrm{L}$ [TMM21] and $\mathrm{A}^2\mathrm{L}^+$ [GMM+22] 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.

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.

Communication. Using a simple Python script (see Appendix G), we estimate the communication complexity of our protocol. The results are presented in Table 3. 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 `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.

8.3 Extensions and Future Work

Here, we discuss how to extend our protocol and possible directions for future research.

Redeem for Arbitrary Signatures Scheme. In Section 6 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

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

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 [BF01] and add IBE ciphertexts to the promise. This particular construction for BLS was recently proposed by Döttling et al. [DHMW22] and called signature witness encryption (SWE). The primitive they propose allows encrypting an arbitrary message, proving any statement about the message using Bulletproofs [BBB+18], 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.

References

[ato19] 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. (Cited on page 3.)

[BBB+18] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy*, pages 315–334. IEEE Computer Society Press, May 2018. (Cited on page 28.)

[BCG+14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474. IEEE Computer Society Press, May 2014. (Cited on page 4.)

[BDF21] Carsten Baum, Bernardo David, and Tore Kasper Frederiksen. P2DEX: Privacy-preserving decentralized cryptocurrency exchange. In Kazue Sako and Nils Ole Tippenhauer, editors, *ACNS 21, Part I*, volume 12726 of *LNCS*, pages 163–194. Springer, Heidelberg, June 2021. (Cited on page 3.)

[BF01] Dan Boneh and Matthew K. Franklin. Identity-based encryption from the Weil pairing. In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 213–229. Springer, Heidelberg, August 2001. (Cited on page 28.)

[BJZ+19] Iddo Bentov, Yan Ji, Fan Zhang, Lorenz Breidenbach, Philip Daian, and Ari Juels. Tesseract: Real-time cryptocurrency exchange using trusted hardware. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 1521–1538. ACM Press, November 2019. (Cited on page 3.)

[BLS01] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. In Colin Boyd, editor, *ASIACRYPT 2001*, volume 2248 of *LNCS*, pages 514–532. Springer, Heidelberg, December 2001. (Cited on page 5, 16, 21, 66.)

[Bol03] Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-Diffie-Hellman-group signature scheme. In Yvo Desmedt, editor, *PKC 2003*, volume 2567 of *LNCS*, pages 31–46. Springer, Heidelberg, January 2003. (Cited on page 66.)

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001. (Cited on page 4, 10, 36.)

[CDG+18] Jan Camenisch, Manu Drijvers, Tommaso Gagliardoni, Anja Lehmann, and Gregory Neven. The wonderful world of global random oracles. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part I*, volume 10820 of *LNCS*, pages 280–312. Springer, Heidelberg, April / May 2018. (Cited on page 10.)

[Cha82] David Chaum. Blind signatures for untraceable payments. In David Chaum, Ronald L. Rivest, and Alan T. Sherman, editors, *CRYPTO'82*, pages 199–203. Plenum Press, New York, USA, 1982. (Cited on page 5, 9.)

[chi22] Chia network faq. https://www.chia.net/faq/, 2022. Accessed: 2023-04-11. (Cited on page 3.)

[CJS14] Ran Canetti, Abhishek Jain, and Alessandra Scafuro. Practical UC security with a global random oracle. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 2014*, pages 597–608. ACM Press, November 2014. (Cited on page 10.)

[Coi13] CoinJoin - Bitcoin Forum. https://bitcointalk.org/?topic=279249, 2013. Accessed: 2023-04-11. (Cited on page 4.)

[DEF18] Stefan Dziembowski, Lisa Eckey, and Sebastian Faust. FairSwap: How to fairly exchange digital goods. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 967–984. ACM Press, October 2018. (Cited on page 10.)

[DHMW22] Nico Döttling, Lucjan Hanzlik, Bernardo Magri, and Stella Wohnig. McFly: Verifiable encryption to the future made practical. Cryptology ePrint Archive, Report 2022/433, 2022. https://eprint.iacr.org/2022/433. (Cited on page 28.)

[EFH+21] Andreas Erwig, Sebastian Faust, Kristina Hostáková, Monosij Maitra, and Siavash Riahi. Two-party adaptor signatures from identification schemes. In Juan Garay, editor, *PKC 2021, Part I*, volume 12710 of *LNCS*, pages 451–480. Springer, Heidelberg, May 2021. (Cited on page 3, 5, 34, 35.)

[fre22] 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. (Cited on page 4.)

[FS10] Marc Fischlin and Dominique Schröder. On the impossibility of three-move blind signature schemes. In Henri Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 197–215. Springer, Heidelberg, May / June 2010. (Cited on page 32.)

[fun22] Understanding bitcoin fungibility. https://river.com/learn/bitcoin-fungibility/, 2022. Accessed: 2023-04-11. (Cited on page 3.)

[GMM+22] Noemi Glaeser, Matteo Maffei, Giulio Malavolta, Pedro Moreno-Sanchez, Erkan Tairi, and Sri Aravinda Krishnan Thyagarajan. Foundations of coin mixing services. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 1259–1273. ACM Press, November 2022. (Cited on page 4, 5, 6, 10, 26.)

[Gro04] Jens Groth. Rerandomizable and replayable adaptive chosen ciphertext attack secure cryptosystems. In Moni Naor, editor, *TCC 2004*, volume 2951 of *LNCS*, pages 152–170. Springer, Heidelberg, February 2004. (Cited on page 4.)

[HAB+17] Ethan Heilman, Leen Alshenibr, Foteini Baldimtsi, Alessandra Scafuro, and Sharon Goldberg. TumbleBit: An untrusted bitcoin-compatible anonymous payment hub. In *NDSS 2017*. The Internet Society, February / March 2017. (Cited on page 3, 4, 5, 10.)

[HBG16] Ethan Heilman, Foteini Baldimtsi, and Sharon Goldberg. Blindly signed contracts: Anonymous on-blockchain and off-blockchain bitcoin transactions. In Jeremy Clark, Sarah Meiklejohn, Peter Y. A. Ryan, Dan S. Wallach, Michael Brenner, and Kurt Rohloff, editors, *FC 2016 Workshops*, volume 9604 of *LNCS*, pages 43–60. Springer, Heidelberg, February 2016. (Cited on page 5, 6, 10.)

[Her18] Maurice Herlihy. Atomic cross-chain swaps, 2018. (Cited on page 3.)

[KMTZ13] Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. Universally composable synchronous computation. In Amit Sahai, editor, *TCC 2013*, volume 7785 of *LNCS*, pages 477–498. Springer, Heidelberg, March 2013. (Cited on page 10.)

[LRR+19] Russell W. F. Lai, Viktoria Ronge, Tim Ruffing, Dominique Schröder, Sri Aravinda Krishnan Thyagarajan, and Jiafan Wang. Omniring: Scaling private payments without trusted setup. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 31–48. ACM Press, November 2019. (Cited on page 3, 4.)

[MMK+17] Giulio Malavolta, Pedro Moreno-Sanchez, Aniket Kate, Matteo Maffei, and Srivatsan Ravi. Concurrency and privacy with payment-channel networks. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 455–471. ACM Press, October / November 2017. (Cited on page 3.)

[MMS+19] Giulio Malavolta, Pedro Moreno-Sanchez, Clara Schneidewind, Aniket Kate, and Matteo Maffei. Anonymous multi-hop locks for blockchain scalability and interoperability. In *NDSS 2019*. The Internet Society, February 2019. (Cited on page 6, 10.)

[MSH+17] Malte Möser, Kyle Soska, Ethan Heilman, Kevin Lee, Henry Heffan, Shashvat Srivastava, Kyle Hogan, Jason Hennessey, Andrew Miller, Arvind Narayanan, et al. An empirical analysis of traceability in the monero blockchain. *arXiv preprint arXiv:1704.04299*, 2017. (Cited on page 4.)

[OKH13] Micha Ober, Stefan Katzenbeisser, and Kay Hamacher. Structure and anonymity of the bitcoin transaction graph. *Future internet*, 5(2):237–250, 2013. (Cited on page 4.)

[QPM+22] Xianrui Qin, Shimin Pan, Arash Mirzaei, Zhimei Sui, Oğuzhan Ersoy, Amin Sakzad, Muhammed F. Esgin, Joseph K. Liu, Jiangshan Yu, and Tsz Hon 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. (Cited on page 3, 4.)

[rai22] Raiden network. https://raiden.network/, 2022. Accessed: 2023-04-11. (Cited on page 3.)

[RH13] Fergal Reid and Martin Harrigan. An analysis of anonymity in the bitcoin system. In *Security and privacy in social networks*, pages 197–223. Springer, 2013. (Cited on page 4.)

[RMK14] Tim Ruffing, Pedro Moreno-Sanchez, and Aniket Kate. CoinShuffle: Practical decentralized coin mixing for bitcoin. In Miroslaw Kutylowski and Jaideep Vaidya, editors, *ESORICS 2014, Part II*, volume 8713 of *LNCS*, pages 345–364. Springer, Heidelberg, September 2014. (Cited on page 4.)

[RMK17] Tim Ruffing, Pedro Moreno-Sanchez, and Aniket Kate. P2P mixing and unlinkable bitcoin transactions. In *NDSS 2017*. The Internet Society, February / March 2017. (Cited on page 4.)

[RS13] Dorit Ron and Adi Shamir. Quantitative analysis of the full bitcoin transaction graph. In *International Conference on Financial Cryptography and Data Security*, pages 6–24. Springer, 2013. (Cited on page 4.)

[Sch91] Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, January 1991. (Cited on page 21.)

[Sha79] Adi Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, November 1979. (Cited on page 9.)

[SO13] Marc Santamaria Ortega. The bitcoin transaction graph anonymity. 2013. (Cited on page 4.)

[TBM+20] Sri Aravinda Krishnan Thyagarajan, Adithya Bhat, Giulio Malavolta, Nico Döttling, Aniket Kate, and Dominique Schröder. Verifiable timed signatures made practical. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1733–1750. ACM Press, November 2020. (Cited on page 4, 5.)

[TM21] Sri Aravinda Krishnan Thyagarajan and Giulio Malavolta. Lockable signatures for blockchains: Scriptless scripts for all signatures. In *2021 IEEE Symposium on Security and Privacy*, pages 937–954. IEEE Computer Society Press, May 2021. (Cited on page 10.)

[TMM21] Erkan Tairi, Pedro Moreno-Sanchez, and Matteo Maffei. A^2L: Anonymous atomic locks for scalability in payment channel hubs. In *2021 IEEE Symposium on Security and Privacy*, pages 1834–1851. IEEE Computer Society Press, May 2021. (Cited on page 3, 4, 5, 6, 10, 26.)

[TMM22] Sri Aravinda Krishnan Thyagarajan, Giulio Malavolta, and Pedro Moreno-Sanchez. Universal atomic swaps: Secure exchange of coins across all blockchains. In *2022 IEEE Symposium on Security and Privacy*, pages 1299–1316. IEEE Computer Society Press, May 2022. (Cited on page 3, 4, 10.)

[uni20] Uniswap. https://uniswap.org/whitepaper.pdf, 2020. Accessed: 2023-04-11. (Cited on page 3.)

Appendix

A Detailed Preliminaries

Digital Signatures.

Definition A.1 (Signature Scheme). A signature scheme SIG is a tuple $\mathsf{SIG} = (\mathsf{Gen}, \mathsf{Sig}, \mathsf{Ver})$ of PPT algorithms with the following syntax:

- $\mathsf{Gen}(1^\lambda) \to (\mathsf{pk}, \mathsf{sk})$ takes as input the security parameter 1^λ and outputs a public key pk and a secret key sk.
- $\mathsf{Sig}(\mathsf{sk}, \mathsf{m}) \to \sigma$ takes as input a secret key sk and a message m, and outputs a signature σ.
- $\mathsf{Ver}(\mathsf{pk}, \mathsf{m}, \sigma) \to b$ is deterministic, takes as input a public key pk, a message m, and a signature σ and outputs a bit $b \in \{0, 1\}$.

We require that SIG is complete in the following sense: For all keys $(\mathsf{pk}, \mathsf{sk}) \in \mathsf{Gen}(1^\lambda)$ and all messages m, we have

$$\Pr[\mathsf{Ver}(\mathsf{pk}, \mathsf{m}, \sigma) = 1 \mid \sigma \leftarrow \mathsf{Sig}(\mathsf{sk}, \mathsf{m})] = 1.$$

Definition A.2 (Unique Signatures). Let $\mathsf{SIG} = (\mathsf{Gen}, \mathsf{Sig}, \mathsf{Ver})$ be a signature scheme. We say that SIG has unique signatures, if for every public key pk (not necessarily output by Gen) and every message m, there is exactly one signature σ such that $\mathsf{Ver}(\mathsf{pk}, \mathsf{m}, \sigma) = 1$.

Definition A.3 (Smoothness). Let $\mathsf{SIG} = (\mathsf{Gen}, \mathsf{Sig}, \mathsf{Ver})$ be a signature scheme. Assume that signatures have length $\ell = \ell(\lambda)$ bits. We say that SIG is smooth, if for every public key pk (not necessarily output by Gen) and every message m, the following probability is negligible:

$$\Pr\left[\mathsf{Ver}(\mathsf{pk}, \mathsf{m}, \sigma) = 1 \mid \sigma \leftarrow\!\!\$\, \{0, 1\}^\ell\right].$$

Definition A.4 (Public Key Entropy). Let $\mathsf{SIG} = (\mathsf{Gen}, \mathsf{Sig}, \mathsf{Ver})$ be a signature scheme and $f : \mathbb{N} \to \mathbb{R}$ be a function. We say that SIG is has public key entropy f, if for all public keys pk_0 the following holds

$$\Pr\left[\mathsf{pk} = \mathsf{pk}_0 \mid (\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Gen}(1^\lambda)\right] \leq 2^{-f(\lambda)}.$$

Definition A.5 (Unforgeability). Consider a signature scheme $\mathsf{SIG} = (\mathsf{Gen}, \mathsf{Sig}, \mathsf{Ver})$. For any algorithm $\mathcal{A}$, consider the following game:

1. Generate a key pair $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Gen}(1^\lambda)$ and initialize $\mathcal{Q} := \emptyset$.
2. Let $\textsc{Sig}$ be an oracle that on input m sets $\mathcal{Q} := \mathcal{Q} \cup \{\mathsf{m}\}$ and returns $\mathsf{Sig}(\mathsf{sk}, \mathsf{m})$.
3. Run $\mathcal{A}$ with access to oracle $\textsc{Sig}$ and on input pk. Obtain a pair (m^*, σ^*) in return.
4. If $\mathsf{m}^* \in \mathcal{Q}$ or $\mathsf{Ver}(\mathsf{pk}, \mathsf{m}^*, \sigma^*) = 0$, return 0. Otherwise, return 1.

We say that SIG is $\mathsf{EUF\text{-}CMA}$ secure, if for all PPT algorithms $\mathcal{A}$, the probability that the above game outputs 1 is negligible.

Definition A.6 (Strong Unforgeability). Let $\mathsf{SIG} = (\mathsf{Gen}, \mathsf{Sig}, \mathsf{Ver})$ be a signature scheme. For any algorithm $\mathcal{A}$, consider the following game:

1. Generate a key pair $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Gen}(1^\lambda)$ and initialize $\mathcal{Q} := \emptyset$.
2. Let $\textsc{Sig}$ be an oracle that takes as input a message m, computes $\sigma \leftarrow \mathsf{Sig}(\mathsf{sk}, \mathsf{m})$, sets $\mathcal{Q} := \mathcal{Q} \cup \{(\mathsf{m}, \sigma)\}$ and returns σ.
3. Run $\mathcal{A}$ with access to oracle $\textsc{Sig}$ and on input pk. Obtain a pair (m^*, σ^*) in return.
4. If $(\mathsf{m}^*, \sigma^*) \in \mathcal{Q}$ or $\mathsf{Ver}(\mathsf{pk}, \mathsf{m}^*, \sigma^*) = 0$, return 0. Otherwise, return 1.

We say that SIG is $\mathsf{sEUF\text{-}CMA}$ secure, if for all PPT algorithms $\mathcal{A}$, the probability that the above game outputs 1 is negligible.

Blind Signatures.

Definition A.7 (Blind Signature Scheme). A (two-move) blind signature scheme $\mathsf{BS} = (\mathsf{Gen}, \mathsf{S}, \mathsf{U}, \mathsf{Ver})$ is a quadruple of PPT algorithms with the following syntax:

- $\mathsf{Gen}(1^\lambda) \to (\mathsf{pk}, \mathsf{sk})$ takes as input the security parameter 1^λ and outputs a public key pk and a secret key sk.
- $\mathsf{U} = (\mathsf{U}_1, \mathsf{U}_2)$ is split into two algorithms: $\mathsf{U}_1(\mathsf{pk}, \mathsf{m}) \to (\mathsf{bsm}_1, St)$ takes as input a public key pk and a message m and outputs a message bsm_1 and a state St; $\mathsf{U}_2(St, \mathsf{bsm}_2) \to \sigma$ takes as input a state St and a message bsm_2, and outputs a signature σ.
- $\mathsf{S}(\mathsf{sk}, \mathsf{bsm}_1) \to \mathsf{bsm}_2$ takes as input a secret key sk and a message bsm_1, and outputs a message bsm_2.
- $\mathsf{Ver}(\mathsf{pk}, \mathsf{m}, \sigma) \to b$ is deterministic, takes as input a public key pk, a message m, and a signature σ, and returns $b \in \{0, 1\}$.

Given BS, we define algorithm $\mathsf{BS.Sig}(\mathsf{sk}, \mathsf{m})$ for $(\mathsf{pk}, \mathsf{sk}) \in \mathsf{Gen}(1^\lambda)$ and a messages m as running the following steps and outputting σ:

$$(\mathsf{bsm}_1, St) \leftarrow \mathsf{U}_1(\mathsf{pk}, \mathsf{m}), \quad \mathsf{bsm}_2 \leftarrow \mathsf{S}(\mathsf{sk}, \mathsf{bsm}_1), \quad \sigma \leftarrow \mathsf{U}_2(St, \mathsf{bsm}_2).$$

We require that BS is complete in the following sense: For all $(\mathsf{pk}, \mathsf{sk}) \in \mathsf{Gen}(1^\lambda)$ and all messages m, we have

$$\Pr\left[\mathsf{Ver}(\mathsf{pk}, \mathsf{m}, \sigma) = 1 \mid \sigma \leftarrow \mathsf{BS.Sig}(\mathsf{sk}, \mathsf{m})\right] = 1.$$

In this work, we only consider signature schemes and blind signature schemes for which one can efficiently decide if $(\mathsf{pk}, \mathsf{sk}) \in \mathsf{Gen}(1^\lambda)$ for given $(\mathsf{pk}, \mathsf{sk})$. This holds true for all schemes used in practice.

Definition A.8 (Unique Blind Signatures). Let $\mathsf{BS} = (\mathsf{Gen}, \mathsf{S}, \mathsf{U}, \mathsf{Ver})$ be a blind signature scheme. We say that BS has unique signatures, if for every public key pk (not necessarily output by Gen) and every message m, there is exactly one signature σ such that $\mathsf{Ver}(\mathsf{pk}, \mathsf{m}, \sigma) = 1$.

We define a weak form of blindness against malicious signers, where the signer does not get signatures in the end. If a scheme has so called signature-derivation checks [FS10], this is implied by the standard notion of blindness. It is sufficient for our purposes[15].

Definition A.9 (Weak Blindness). Let $\mathsf{BS} = (\mathsf{Gen}, \mathsf{S}, \mathsf{U}, \mathsf{Ver})$ be a blind signature scheme. For any algorithm $\mathcal{A}$ and bit $b \in \{0, 1\}$, consider the following game:

1. Run $\mathcal{A}$ and get a key pk and messages $\mathsf{m}_0, \mathsf{m}_1$.
2. Run $(\mathsf{bsm}_1, St) \leftarrow \mathsf{U}_1(\mathsf{pk}, \mathsf{m}_b)$ and give bsm_1 to $\mathcal{A}$.

[15] We require unique blind signatures for our construction. For unique blind signatures with signature-derivation checks this notion and the standard blindness notion are equivalent.

3. Get bsm_2 from $\mathcal{A}$ and run $\sigma \leftarrow \mathsf{U}_2(St, \mathsf{bsm}_2)$.

4. Give $\mathsf{Ver}(\mathsf{pk}, \mathsf{m}_b, \sigma)$ to $\mathcal{A}$ and obtain a bit b' in return.

5. Output b'.

We say that BS is weakly blind, 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.

Definition A.10 (One-More Unforgeability). Let $\mathsf{BS} = (\mathsf{Gen}, \mathsf{S}, \mathsf{U}, \mathsf{Ver})$ be a blind signature scheme. For any algorithm $\mathcal{A}$, consider the following game:

1. Generate keys $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Gen}(1^\lambda)$.

2. Let O be an oracle that on input bsm_1 returns $\mathsf{bsm}_2 \leftarrow \mathsf{BS.S}(\mathsf{sk}, \mathsf{bsm}_1)$.

3. Run $\mathcal{A}$ on input pk with access to oracle O and obtain $(\mathsf{m}_1, \sigma_1), \ldots, (\mathsf{m}_k, \sigma_k)$.

4. Let ℓ denote the number of queries that $\mathcal{A}$ made to O. Output 1 if the following three conditions hold. Otherwise, output 0:
   (a) We have $k > \ell$.
   (b) For all $i, j \in [k]$ with $i \neq j$ we have $\mathsf{m}_i \neq \mathsf{m}_j$.
   (c) For all $i \in [k]$ we have $\mathsf{Ver}(\mathsf{pk}, \mathsf{m}_i, \sigma_i) = 1$.

We say that BS satisfies one-more unforgeability, if for all PPT algorithms $\mathcal{A}$, the probability that the above game outputs 1 is negligible.

NP-Relations.

Definition A.11 (**NP**-Relation). Let $\mathcal{R} = (\mathcal{R}_\lambda)_\lambda$ be a family of binary relations $\mathcal{R}_\lambda \subseteq \{0,1\}^* \times \{0,1\}^*$. We define the language of yes-instances $\mathcal{L}_\lambda$ via

$$\mathcal{L}_\lambda := \left\{\mathsf{stmt} \in \{0,1\}^* \;\middle|\; \exists \mathsf{witn} \in \{0,1\}^* : (\mathsf{stmt}, \mathsf{witn}) \in \mathcal{R}_\lambda\right\}.$$

We say that $\mathcal{R}$ is an **NP**-relation, if the following properties hold:

- There exists a polynomial poly, such that for any $\mathsf{stmt} \in \mathcal{L}_\lambda$, we have $|\mathsf{stmt}| \leq \mathsf{poly}(\lambda)$.
- Membership in $\mathcal{R}_\lambda$ is efficiently decidable, i.e. there exists a deterministic polynomial time algorithm that decides $\mathcal{R}_\lambda$.
- There is a polynomial poly' such that for all $(\mathsf{stmt}, \mathsf{witn}) \in \mathcal{R}_\lambda$ we have $|\mathsf{witn}| \leq \mathsf{poly}'(|\mathsf{stmt}|)$.

Definition A.12 (Hard **NP**-Relation). Let $\mathcal{R} = (\mathcal{R}_\lambda)_\lambda$ be an **NP**-relation. Assume that there is a PPT algorithm $\mathcal{R}.\mathsf{Gen}$ that on input 1^λ outputs tuples $(\mathsf{stmt}, \mathsf{witn}) \in \mathcal{R}_\lambda$. We say that $\mathcal{R}$ is hard relative to $\mathcal{R}.\mathsf{Gen}$ if for any PPT algorithm $\mathcal{A}$ the following probability is negligible:

$$\Pr\left[(\mathsf{stmt}, \mathsf{witn}') \in \mathcal{R}_\lambda \;\middle|\; \begin{array}{l} (\mathsf{stmt}, \mathsf{witn}) \leftarrow \mathcal{R}.\mathsf{Gen}(1^\lambda), \\ \mathsf{witn}' \leftarrow \mathcal{A}(\mathsf{stmt}) \end{array}\right].$$

Definition A.13 (Unique **NP**-Relation). Let $\mathcal{R} = (\mathcal{R}_\lambda)_\lambda$ be an **NP**-relation. We say that $\mathcal{R}$ is unique if for any $\mathsf{stmt} \in \mathcal{L}_\lambda$ there is exactly one witn such that $(\mathsf{stmt}, \mathsf{witn}) \in \mathcal{R}_\lambda$.

Adaptor Signatures.

Definition A.14 (Adaptor Signature). Let SIG be a signature scheme and $\mathcal{R}$ an **NP**-relation. An adaptor signature scheme for SIG and $\mathcal{R}$ is a tuple $\mathsf{aSIG} = (\mathsf{PreSig}, \mathsf{Adapt}, \mathsf{PreVer}, \mathsf{Ext})$ of PPT algorithms with the following syntax:

- $\mathsf{PreSig}(\mathsf{sk}, \mathsf{m}, \mathsf{stmt}) \rightarrow \tilde{\sigma}$ takes as input a secret key sk, a message m, and a statement stmt, and outputs a pre-signature $\tilde{\sigma}$.

- $\mathsf{Adapt}(\mathsf{pk}, \tilde{\sigma}, \mathsf{witn}) \to \sigma$ is deterministic, takes as input a public key pk, a pre-signature $\tilde{\sigma}$, and a witness witn, and outputs a signature σ.
- $\mathsf{PreVer}(\mathsf{pk}, \mathsf{m}, \mathsf{stmt}, \tilde{\sigma}) \to b$ is deterministic, takes as input a public key pk, a message m, a statement stmt, and a pre-signature $\tilde{\sigma}$, and returns $b \in \{0, 1\}$.
- $\mathsf{Ext}(\tilde{\sigma}, \sigma) \to \mathsf{witn}$ is deterministic, takes as input a pre-signature $\tilde{\sigma}$, a signature σ, and outputs a witness witn.

We require that aSIG is complete in the following sense: For all $(\mathsf{pk}, \mathsf{sk}) \in \mathsf{Gen}(1^\lambda)$, all messages m, and all $(\mathsf{stmt}, \mathsf{witn}) \in \mathcal{R}_\lambda$, we have

$$\Pr\left[\begin{array}{l|l} \mathsf{Ver}(\mathsf{pk}, \mathsf{m}, \sigma) = 1 \wedge & \tilde{\sigma} \leftarrow \mathsf{PreSig}(\mathsf{sk}, \mathsf{m}, \mathsf{stmt}), \\ (\mathsf{stmt}, \mathsf{witn}') \in \mathcal{R}_\lambda \wedge & \sigma := \mathsf{Adapt}(\mathsf{pk}, \tilde{\sigma}, \mathsf{witn}), \\ \mathsf{PreVer}(\mathsf{pk}, \mathsf{m}, \mathsf{stmt}, \tilde{\sigma}) = 1 & \mathsf{witn}' := \mathsf{Ext}(\tilde{\sigma}, \sigma) \end{array}\right] = 1.$$

Definition A.15 (Adaptability). Let SIG be a signature scheme, $\mathcal{R}$ an **NP**-relation, and $\mathsf{aSIG} = (\mathsf{PreSig}, \mathsf{Adapt}, \mathsf{PreVer}, \mathsf{Ext})$ be an adaptor signature scheme for SIG and $\mathcal{R}$. We say that aSIG satisfies adaptability, if for all messages m, pairs $(\mathsf{stmt}, \mathsf{witn}) \in \mathcal{R}_\lambda$, keys pk and pre-signatures $\tilde{\sigma}$ the following implication holds:

$$\mathsf{PreVer}(\mathsf{pk}, \mathsf{m}, \mathsf{stmt}, \tilde{\sigma}) = 1 \Rightarrow \mathsf{Ver}(\mathsf{pk}, \mathsf{m}, \mathsf{Adapt}(\mathsf{pk}, \tilde{\sigma}, \mathsf{witn})) = 1.$$

Definition A.16 (Witness Extractability). Let SIG be a signature scheme, $\mathcal{R}$ an **NP**-relation, and $\mathsf{aSIG} = (\mathsf{PreSig}, \mathsf{Adapt}, \mathsf{PreVer}, \mathsf{Ext})$ be an adaptor signature scheme for SIG and $\mathcal{R}$. For any algorithm $\mathcal{A}$ consider the following game:

1. Sample keys $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Gen}(1^\lambda)$ and initialize $\mathcal{Q} := \emptyset$.
2. Let $\textsc{Sig}, \textsc{PreSig}$ be oracles, defined as follows:
   - $\textsc{Sig}(\mathsf{m})$: Set $\mathcal{Q} := \mathcal{Q} \cup \{\mathsf{m}\}$ and return $\mathsf{Sig}(\mathsf{sk}, \mathsf{m})$.
   - $\textsc{PreSig}(\mathsf{m}, \mathsf{stmt})$: Set $\mathcal{Q} := \mathcal{Q} \cup \{\mathsf{m}\}$. Then, return $\mathsf{PreSig}(\mathsf{sk}, \mathsf{m}, \mathsf{stmt})$.
3. Run $\mathcal{A}$ on input pk with access to $\textsc{Sig}, \textsc{PreSig}$. Obtain $(\mathsf{m}^*, \mathsf{stmt}^*)$ in return.
4. Compute $\tilde{\sigma} \leftarrow \mathsf{PreSig}(\mathsf{sk}, \mathsf{m}^*, \mathsf{stmt}^*)$ and give $\tilde{\sigma}$ to $\mathcal{A}$. Obtain σ^* in return.
5. Run $\mathsf{witn} := \mathsf{Ext}(\tilde{\sigma}, \sigma^*)$.
6. Output 1 if $\mathsf{Ver}(\mathsf{pk}, \mathsf{m}^*, \sigma^*)$, $\mathsf{m}^* \notin \mathcal{Q}$, and $(\mathsf{stmt}^*, \mathsf{witn}) \notin \mathcal{R}_\lambda$. Otherwise, output 0.

We say that aSIG satisfies witness extractability, if for all PPT algorithms $\mathcal{A}$, the probability that the above game outputs 1 is negligible.

Our definition of $\mathsf{aEUF\text{-}CMA}$ is weaker than the standard notion (e.g. in [EFH+21]) in a sense that we do not give the adversary a pre-signature on the message m^*.

Definition A.17 (Adaptor Unforgeability). Let SIG be a signature scheme, $\mathcal{R}$ an **NP**-relation, and $\mathsf{aSIG} = (\mathsf{PreSig}, \mathsf{Adapt}, \mathsf{PreVer}, \mathsf{Ext})$ be an adaptor signature scheme for SIG and $\mathcal{R}$. For any algorithm $\mathcal{A}$ consider the following game:

1. Sample keys $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Gen}(1^\lambda)$ and initialize $\mathcal{Q} := \emptyset$.
2. Let $\textsc{Sig}, \textsc{PreSig}$ be oracles, defined as follows:
   - $\textsc{Sig}(\mathsf{m})$: Set $\mathcal{Q} := \mathcal{Q} \cup \{\mathsf{m}\}$ and return $\mathsf{Sig}(\mathsf{sk}, \mathsf{m})$.
   - $\textsc{PreSig}(\mathsf{m}, \mathsf{stmt})$: Set $\mathcal{Q} := \mathcal{Q} \cup \{\mathsf{m}\}$. Then, return $\mathsf{PreSig}(\mathsf{sk}, \mathsf{m}, \mathsf{stmt})$.
3. Run $\mathcal{A}$ on input pk with access to oracles $\textsc{Sig}, \textsc{PreSig}$. Obtain a pair (m^*, σ^*) in return.
4. Output 1 if $\mathsf{m}^* \notin \mathcal{Q}$ and $\mathsf{Ver}(\mathsf{pk}, \mathsf{m}^*, \sigma^*) = 1$. Otherwise, output 0.

We say that aSIG is $\mathsf{aEUF\text{-}CMA}$ secure, if for all PPT algorithms $\mathcal{A}$, the probability that the above game outputs 1is negligible.

We also define a notion capturing that adapted signatures look like standard signatures. It is easy to see that this notion is satisfied by known constructions, e.g. in [EFH+21].

Definition A.18 (Well Adapted Signatures). Let SIG be a signature scheme, $\mathcal{R}$ an **NP**-relation, and $\mathsf{aSIG} = (\mathsf{PreSig}, \mathsf{Adapt}, \mathsf{PreVer}, \mathsf{Ext})$ be an adaptor signature scheme for SIG and $\mathcal{R}$. We say that aSIG has well adapted signatures, if for all keys $(\mathsf{pk}, \mathsf{sk}) \in \mathsf{Gen}(1^\lambda)$, all messages m, and all pairs $(\mathsf{stmt}, \mathsf{witn}) \in \mathcal{R}_\lambda$, the following distributions $\mathcal{D}_1$ and $\mathcal{D}_2$ are the same:

$$\mathcal{D}_1 := \left\{ (\mathsf{pk}, \mathsf{sk}, \mathsf{m}, \sigma) \;\middle|\; \begin{array}{l} \tilde{\sigma} \leftarrow \mathsf{PreSig}(\mathsf{sk}, \mathsf{m}, \mathsf{stmt}), \\ \sigma := \mathsf{Adapt}(\mathsf{pk}, \tilde{\sigma}, \mathsf{witn}) \end{array} \right\},$$
$$\mathcal{D}_2 := \left\{ (\mathsf{pk}, \mathsf{sk}, \mathsf{m}, \sigma) \;\middle|\; \sigma \leftarrow \mathsf{Sig}(\mathsf{sk}, \mathsf{m}) \right\}.$$

Non-Interactive Proofs. We define non-interactive zero-knowledge proofs. For simplicity, we define proofs in the random oracle model. However, other formalizations, e.g. in the common reference string model, would also be applicable for our purposes. Without loss of generality, we assume that inputs to random oracles that are used in proof systems are prefixed with the statement. This domain separation allows to use the simulator PSim multiple times without introducing conflicts due to random oracle programming.

Definition A.19 (Non-Interactive Proof System). Let $\mathcal{R}$ be an **NP**-relation. A non-interactive proof system for $\mathcal{R}$ is a tuple $\mathsf{PS} = (\mathsf{PProve}, \mathsf{PVer})$ of PPT algorithms with the following syntax:

- $\mathsf{PProve}(\mathsf{stmt}, \mathsf{witn}) \to \pi$ takes as input a statement stmt and a witness witn, and outputs a proof π.
- $\mathsf{PVer}(\mathsf{stmt}, \pi) \to b$ is deterministic, takes as input a statement stmt, a proof π, and outputs a bit $b \in \{0, 1\}$.

We require that PS is complete in the following sense: For all $(\mathsf{stmt}, \mathsf{witn}) \in \mathcal{R}_\lambda$, we have

$$\Pr\left[\mathsf{PVer}(\mathsf{stmt}, \pi) = 1 \mid \pi \leftarrow \mathsf{PProve}(\mathsf{stmt}, \mathsf{witn})\right] = 1.$$

Definition A.20 (Soundness). Let $\mathcal{R}$ be an **NP**-relation and $\mathsf{PS} = (\mathsf{PProve}, \mathsf{PVer})$ be a non-interactive proof system for $\mathcal{R}$. We say that PS is sound, if for any algorithm $\mathcal{A}$, the following probability is negligible:

$$\Pr\left[\mathsf{PVer}(\mathsf{stmt}, \pi) = 1 \land \mathsf{stmt} \notin \mathcal{L}_\lambda \mid (\mathsf{stmt}, \pi) \leftarrow \mathcal{A}(1^\lambda)\right].$$

Definition A.21 (Zero-Knowledge). Consider an **NP**-relation $\mathcal{R}$ and a non-interactive proof system $\mathsf{PS} = (\mathsf{PProve}, \mathsf{PVer})$ for $\mathcal{R}$. We say that PS is zero-knowledge, if there exists a PPT algorithm PSim, that is allowed to program random oracles, such that for any $(\mathsf{stmt}, \mathsf{witn}) \in \mathcal{R}_\lambda$, the following distributions $\mathcal{D}_1$ and $\mathcal{D}_2$ are statistically close:

$$\mathcal{D}_1 := \{\pi \leftarrow \mathsf{PProve}(\mathsf{stmt}, \mathsf{witn})\},\ \mathcal{D}_2 := \{\pi \leftarrow \mathsf{PSim}(\mathsf{stmt})\}$$

If a non-interactive proof system PS for an **NP**-relation $\mathcal{R}$ is both sound and zero-knowledge, we also refer to it as a NIZK.

Computational Assumptions.

Definition A.22 (DLOG Assumption). Let $\mathbb{G}$ be a (family of) cyclic group(s) of prime order $p > 2^\lambda$ with generator $g \in \mathbb{G}$. We say that the DLOG assumption holds in $\mathbb{G}$ if for all PPT algorithms $\mathcal{A}$ the following is negligible:

$$\Pr\left[\mathcal{A}(g, g^x) = x \mid x \leftarrow\!\!\$\, \mathbb{Z}_p\right].$$

Definition A.23 (DDH Assumption). Let $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ be (families of) cyclic groups of prime order $p > 2^\lambda$ with generators $g_1 \in \mathbb{G}_1, g_2 \in \mathbb{G}_2$ and $g_T := e(g_1, g_2) \in \mathbb{G}_T$, where $e : \mathbb{G}_1 \times \mathbb{G}_2$ is a pairing. For $i \in \{1, 2\}$, we say that the DDH assumption holds in $\mathbb{G}_i$ if for all PPT algorithms $\mathcal{A}$ the following is negligible:

$$\begin{aligned} &|\Pr\left[\mathcal{A}(g_1, g_2, e, X, Y, Z) = 1 \;\middle|\; x, y \leftarrow\!\!\$\, \mathbb{Z}_p, X := g_i^x, Y := g_i^y, Z := g_i^{xy}\right] \\ &- \Pr\left[\mathcal{A}(g_1, g_2, e, X, Y, Z) = 1 \;\middle|\; x, y, z \leftarrow\!\!\$\, \mathbb{Z}_p, X := g_i^x, Y := g_i^y, Z := g_i^z\right]|. \end{aligned}$$

Universal Composability Framework. In the universal composability (UC) framework [Can01], all parties are modelled as interactive Turing machines. For an environment $\mathcal{Z}$, an adversary $\mathcal{A}$, a protocol π, and a functionality $\mathcal{G}$, we write $Hybrid^{\mathcal{G}}_{\mathcal{Z},\mathcal{A},\pi}$ to denote the output distribution of $\mathcal{Z}$ in the execution with protocol π and adversary $\mathcal{A}$. Here, π is given access to ideal functionality $\mathcal{G}$. In the execution, the environment communicates with all parties that interact in the protocol via the interfaces of the protocol. At setup time, $\mathcal{A}$ is allowed to corrupt a number of parties. For an ideal functionality $\mathcal{F}$, we write $Ideal_{\mathcal{Z},\mathcal{S},\mathcal{F}}$ to denote the output distribution of $\mathcal{Z}$ when it interacts with functionality $\mathcal{F}$ via dummy parties that forward messages between $\mathcal{Z}$ and $\mathcal{F}$, and a simulator $\mathcal{S}$.

Definition A.24 (UC Security). A protocol π realizes functionality $\mathcal{F}$ in the $\mathcal{G}$-hybrid model, if for all PPT adversaries $\mathcal{A}$, there is a simulator $\mathcal{S}$, such that for any environment $\mathcal{Z}$, the distributions $Hybrid^{\mathcal{G}}_{\mathcal{Z},\mathcal{A},\pi}$ and $Ideal_{\mathcal{Z},\mathcal{S},\mathcal{F}}$ are computationally indistinguishable.

B Security Proofs of Exchange Protocols

Remark. The key ideas and many steps of our proofs for exchange protocols are very similar, which is why we reuse parts verbatim in different proofs. It is recommended to understand the proofs for the generic constructions first, before reading the proofs for the cut-and-choose construction.

B.1 Proofs for the Construction for Adaptor Signatures

Proof of Lemma 5.7 (Mal. Seller - Adaptor Signature). Consider an adversary $\mathcal{A}$ against the security of $\mathsf{EXC_a}[\mathsf{SIG}, \mathsf{aSIG}, \mathsf{BS}, \mathsf{PS}]$ against malicious sellers. We define three events in the security game, following the three possible ways $\mathcal{A}$ can win.

- win_1: This occurs if the security game outputs 1 and $\mathsf{tx} \neq \mathsf{tx}'$.
- win_2: This occurs if the security game outputs 1, $\mathsf{tx} = \mathsf{tx}'$ and $\mathsf{xm}_2 = \perp$.
- win_3: This occurs if the security game outputs 1, $\mathsf{tx} = \mathsf{tx}', \mathsf{xm}_2 \neq \perp$, and $\mathsf{BS.Ver}(\mathsf{pk_{BS}}, \mathsf{sn}, \sigma_{\mathsf{BS}}) = 0$.

First, we bound the probability of $\mathsf{win}_1 \vee \mathsf{win}_2$. Intuitively if one of the events occurs, the adversary came up with a valid signature σ_b for a message tx', for which the game did not compute a signature or pre-signature before. Formally, we give a reduction that runs in the $\mathsf{aEUF\text{-}CMA}$ security game of aSIG. The reduction gets pk as input and access to a signing oracle $\textsc{Sig}$ and a pre-signing oracle $\textsc{PreSig}$. It runs $\mathcal{A}$ and obtains a public key $\mathsf{pk_{BS}}$ and a message sn from $\mathcal{A}$. Then, it runs $(\mathsf{bsm}_1, St) \leftarrow \mathsf{U}_1(\mathsf{pk_{BS}}, \mathsf{sn})$. It sets $\mathsf{pk}_b := \mathsf{pk}$. Next, it gives pk_b and bsm_1 to $\mathcal{A}$, which outputs a key pk_s, a transaction tx and a message xm_1. If $\mathsf{xm}_1 = \perp$ or $\mathsf{xm}_1 = (\mathsf{stmt}', \mathsf{ct}, \pi)$ but $\mathsf{PVer}(\mathsf{stmt}, \pi) = 0$ for $\mathsf{stmt} := (\mathsf{pk_{BS}}, \mathsf{bsm}_1, \mathsf{stmt}', \mathsf{ct})$, the reduction sets $\mathsf{xm}_2 := \perp$. Otherwise, it uses the oracle $\textsc{PreSig}$ as $\tilde{\sigma}_b \leftarrow \textsc{PreSig}(\mathsf{tx}, \mathsf{stmt}')$ and sets $\mathsf{xm}_2 := \tilde{\sigma}_b$. The reduction gives xm_2 to $\mathcal{A}$ and obtains tx', σ_b, and σ_s in return. If $\mathsf{win}_1 \vee \mathsf{win}_2$ occurs, it returns (tx', σ_b) to its game. Otherwise, it aborts. It is clear that the reduction perfectly simulates the game for $\mathcal{A}$. Also, note that the pair (tx', σ_b) that the reduction outputs in the end is valid, i.e. $\mathsf{SIG.Ver}(\mathsf{pk}, \mathsf{tx}', \sigma_b) = 1$, by definition of $\mathsf{win}_1 \vee \mathsf{win}_2$. Further, note that if win_1 occurs, the reduction did only query oracle $\textsc{PreSig}$ on input $\mathsf{tx} \neq \mathsf{tx}'$, and not on input tx'. Similarly, if win_2 occurs, the reduction did not query $\textsc{PreSig}$ at all. In both cases, the reduction did never query oracle $\textsc{Sig}$. Therefore, the probability of $\mathsf{win}_1 \vee \mathsf{win}_2$ can be upper bounded by the probability that the reduction wins the $\mathsf{aEUF\text{-}CMA}$ game. This is negligible by assumption.

It remains to bound the probability of event win_3. To do so, we partition win_3 into two events. Let $\mathsf{xm}_1 = (\mathsf{stmt}', \mathsf{ct}, \pi)$ and $\mathsf{xm}_2 = \tilde{\sigma}_b$ be as in the security game against malicious sellers.

- $\mathsf{win}_{3,1}$: This event occurs, if win_3 occurs and for $\mathsf{witn}' := \mathsf{Ext}(\tilde{\sigma}_b, \sigma_b)$ we have $(\mathsf{stmt}', \mathsf{witn}') \notin \mathcal{R}'$.
- $\mathsf{win}_{3,2}$: This event occurs, if win_3 occurs and for $\mathsf{witn}' := \mathsf{Ext}(\tilde{\sigma}_b, \sigma_b)$ we have $(\mathsf{stmt}', \mathsf{witn}') \in \mathcal{R}'$.

Clearly, it is sufficient to bound the probability of both $\mathsf{win}_{3,1}$ and $\mathsf{win}_{3,2}$.

We start with event $\mathsf{win}_{3,1}$. Intuitively, if this event occurs, then the adversary managed to turn the pre-signature $\tilde{\sigma}_b$ into a valid signature, but we can not extract a witness, contradicting the witness extractability of aSIG. Formally, we give a reduction against the witness extractability of aSIG. The reduction gets pk as input and access to oracles $\textsc{Sig}$ and $\textsc{PreSig}$. It runs $\mathcal{A}$ and obtains a public key $\mathsf{pk_{BS}}$ and a message sn from $\mathcal{A}$. Next, it runs $(\mathsf{bsm}_1, St) \leftarrow \mathsf{U}_1(\mathsf{pk_{BS}}, \mathsf{sn})$, sets $\mathsf{pk}_b := \mathsf{pk}$, and gives pk_b and bsm_1 to $\mathcal{A}$, which outputs a key pk_s, a transaction tx and a message xm_1. If $\mathsf{xm}_1 = \perp$ or π does not verify, the reduction aborts. Otherwise, it parses $\mathsf{xm}_1 = (\mathsf{stmt}', \mathsf{ct}, \pi)$ and outputs $(\mathsf{tx}, \mathsf{stmt}')$ to its game. It obtains a pre-signature $\tilde{\sigma}$ in return and sets $\mathsf{xm}_2 := \tilde{\sigma}_b := \tilde{\sigma}$. Then, the reduction passes xm_2 to $\mathcal{A}$ and obtains tx', σ_b, and σ_s in return. If $\mathsf{win}_{3,1}$ occurs, it outputs σ_b to its game. It is easy to see that the witness extractability game outputs 1 if event $\mathsf{win}_{3,1}$ occurs. Especially, the reduction did not use the oracles $\textsc{Sig}$ and $\textsc{PreSig}$ at all.

Finally, we bound the probability of event $\mathsf{win}_{3,2}$. This follows from soundness of PS and uniqueness of $\mathcal{R}'$. Namely, assume towards contradiction that $\mathsf{win}_{3,2}$ occurs and the statement $\mathsf{stmt} = (\mathsf{pk_{BS}}, \mathsf{bsm}_1, \mathsf{stmt}', \mathsf{ct})$ is a yes-instance, i.e. there is some $\mathsf{witn} = (\mathsf{sk_{BS}}, \mathsf{witn}'', \rho)$ such that $(\mathsf{stmt}, \mathsf{witn}) \in \mathcal{R}$. Then, by definition of $\mathcal{R}$, we have $(\mathsf{stmt}', \mathsf{witn}'') \in \mathcal{R}'$ and

$$\mathsf{ct} \oplus \mathsf{H}(\mathsf{witn}'') = \mathsf{BS.S}(\mathsf{sk_{BS}}, \mathsf{bsm}_1; \rho).$$

Uniqueness of $\mathcal{R}'$ implies that $\mathsf{witn}' = \mathsf{witn}''$, where witn' is as in the definition of event $\mathsf{win}_{3,2}$. This implies that

$$\mathsf{Get}(\mathsf{xpar}, \mathsf{xm}_1, \mathsf{xm}_2, \sigma_b, \sigma_s) = \mathsf{BS.S}(\mathsf{sk}_{\mathsf{BS}}, \mathsf{bsm}_1; \rho).$$

Completeness of BS implies that σ_{BS}, as computed in the security game, is a valid blind signature, i.e. $\mathsf{BS.Ver}(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sn}, \sigma_{\mathsf{BS}}) = 1$, contradicting the assumption that $\mathsf{win}_{3,2}$ occurs. In summary, we showed that stmt is not a yes-instance, violating the soundness of PS. □

Proof of Lemma 5.8 (Mal. Buyer - Adaptor Signature). We give algorithms $\mathsf{Sim}_1, \mathsf{Sim}_{RO}, \mathsf{Sim}_2, \mathsf{Sim}_3$, and then we show indistinguishability. The algorithms keep a list L that holds tuples $(\mathsf{tx}, \mathsf{stmt}', \mathsf{witn}', \mathsf{pk}_s, \mathsf{ct})$. Algorithm $\mathsf{Sim}_1(\mathsf{xpar}, \mathsf{sk}_s)$ is as follows:

1. Sample $(\mathsf{stmt}', \mathsf{witn}') \leftarrow \mathcal{R}'.\mathsf{Gen}(1^\lambda)$ and $\mathsf{ct} \leftarrow_{\$} \{0,1\}^{\ell_1}$.
2. Abort if $\mathsf{H}(\mathsf{witn}')$ already defined.
3. Set $\mathsf{stmt} := (\mathsf{pk}_{\mathsf{BS}}, \mathsf{bsm}_1, \mathsf{stmt}', \mathsf{ct})$ and compute $\pi \leftarrow \mathsf{PSim}(\mathsf{stmt})$.
4. Insert $(\mathsf{tx}, \mathsf{stmt}', \mathsf{witn}', \mathsf{pk}_s, \mathsf{ct})$ into L.
5. Return $\mathsf{xm}_1 := (\mathsf{stmt}', \mathsf{ct}, \pi)$.

Algorithm Sim_{RO} simulates the random oracle honestly. However, on a random oracle query $\mathsf{H}(Z)$, it aborts if there is an entry $(\mathsf{tx}, \mathsf{stmt}', \mathsf{witn}', \mathsf{pk}_s, \mathsf{ct})$ in L such that $Z = \mathsf{witn}'$, i.e. $(\mathsf{stmt}', Z) \in \mathcal{R}'$. Algorithm $\mathsf{Sim}_2(\mathsf{xm}_2)$ first parses $\mathsf{xm}_2 = \tilde{\sigma}_b$, and then returns the result of $\mathsf{aSIG.PreVer}(\mathsf{pk}_b, \mathsf{tx}, \mathsf{stmt}', \tilde{\sigma}_b)$. Algorithm $\mathsf{Sim}_3(\mathsf{xm}_2 = \tilde{\sigma}_b, \mathsf{bsm}_2)$ removes entry $(\mathsf{tx}, \mathsf{stmt}', \mathsf{witn}', \mathsf{pk}_s, \mathsf{ct})$ from L, defines $\mathsf{H}(\mathsf{witn}') := \mathsf{bsm}_2 \oplus \mathsf{ct}$, and returns $\sigma_b := \mathsf{Adapt}(\mathsf{pk}_b, \tilde{\sigma}_b, \mathsf{witn}')$.

It remains to show that algorithms $\mathsf{Sim}_1, \mathsf{Sim}_{RO}, \mathsf{Sim}_2, \mathsf{Sim}_3$ satisfy the indistinguishability that is required by the security definition. We show this via a sequence of games.

Game $\mathbf{G}_0$: This game is the security game against malicious buyers with $b = 0$. Recall that in this game, a key pair $(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sk}_{\mathsf{BS}})$ is sampled. Then, the adversary $\mathcal{A}$ gets access to a signer oracle O and an oracle O^*. When $\mathcal{A}$ queries oracle O^*, it samples a key pair $(\mathsf{pk}_s, \mathsf{sk}_s) \leftarrow \mathsf{SIG.Gen}(1^\lambda)$, gives pk_s to $\mathcal{A}$ and obtains a key pk_b, a transaction tx, and a message bsm_1 in return. Then, it runs algorithm Setup. Concretely, it computes bsm_2, samples witn' and stmt', defines ciphertext ct, and computes a proof π as in the scheme. Then, it sets $\mathsf{xm}_1 := (\mathsf{stmt}', \mathsf{ct}, \pi)$ and sends xm_1 to $\mathcal{A}$. The adversary responds with a message xm_2. If $\mathsf{xm}_2 = \tilde{\sigma}_b$ satisfies $\mathsf{PreVer}(\mathsf{pk}_b, \mathsf{tx}, \mathsf{stmt}', \tilde{\sigma}_b) = 1$, the game computes σ_s using sk_s and σ_b via $\sigma_b := \mathsf{Adapt}(\mathsf{pk}_b, \tilde{\sigma}_b, \mathsf{witn}')$. Otherwise, it aborts. Finally, the game outputs whatever $\mathcal{A}$ outputs.

Game $\mathbf{G}_1$: This game is as $\mathbf{G}_0$, but we change how the proof π in message xm_1 is computed by oracle O^*. Recall that before, it was computed via $\pi \leftarrow \mathsf{PProve}(\mathsf{stmt}, \mathsf{witn})$, where stmt and witn are as in algorithm Setup. In game $\mathbf{G}_1$, we simulate it using the zero-knowledge simulator $\mathsf{PS.PSim}$ via $\pi \leftarrow \mathsf{PSim}(\mathsf{stmt})$. Games $\mathbf{G}_0$ and $\mathbf{G}_1$ are indistinguishable by the zero-knowledge property of PS.

Game $\mathbf{G}_2$: In this game, we introduce two bad events bad_1 and bad_2 and let the game abort if one of these occurs. Further, we introduce a list L that contains tuples $(\mathsf{tx}, \mathsf{stmt}', \mathsf{witn}', \mathsf{pk}_s, \mathsf{ct})$. Whenever the values $(\mathsf{stmt}', \mathsf{witn}')$ are sampled using $\mathcal{R}'.\mathsf{Gen}$ by oracle O^* as part of algorithm Setup, the game sets $\mathsf{bad}_1 := 1$ and aborts if $\mathsf{H}(\mathsf{witn}')$ is already defined. Otherwise, it continues the execution of Setup and inserts $(\mathsf{tx}, \mathsf{stmt}', \mathsf{pk}_s, \mathsf{ct})$ into L. Later, as soon as the oracle O^* returns the signatures σ_b, σ_s, it removes this entry $(\mathsf{tx}, \mathsf{stmt}', \mathsf{witn}', \mathsf{pk}_s, \mathsf{ct})$ from L. Further, we introduce an event bad_2 that occurs if in a random oracle query $\mathsf{H}(Z)$ there is an entry $(\mathsf{tx}, \mathsf{stmt}', \mathsf{witn}', \mathsf{pk}_s, \mathsf{ct})$ in L such that $(\mathsf{stmt}', Z) \in \mathcal{R}'$. If this event occurs, the game aborts. To show indistinguishability of $\mathbf{G}_2$ and $\mathbf{G}_3$, it is sufficient to bound the probability of event $\mathsf{bad}_1 \vee \mathsf{bad}_2$. To do this, we write

$$\mathsf{bad}_1 \vee \mathsf{bad}_2 = \bigvee_{i \in [Q]} \mathsf{bad}_{1,i} \vee \mathsf{bad}_{2,i}.$$

Here, Q denotes the number of queries to oracles O^*, and $\mathsf{bad}_{1,i}$ (resp. $\mathsf{bad}_{2,i}$) denotes the event that bad_1 (resp. bad_2) occurs for the entry in L that is inserted in the ith query to O^*. As Q is polynomially bounded, it is sufficient to bound the probability of event $\mathsf{bad}_{1,i} \vee \mathsf{bad}_{2,i}$ for all $i \in [Q]$. To do so, we give a reduction from the hardness of $\mathcal{R}'$ relative to $\mathcal{R}'.\mathsf{Gen}$.

The reduction gets as input a statement stmt'^*. It simulates $\mathbf{G}_1$ as it is, except for the ith call to oracle O^*, and the random oracle H:

- In the ith call to oracle O^*, the reduction sets $\mathsf{stmt}' := \mathsf{stmt}^*$, instead of sampling $(\mathsf{stmt}', \mathsf{witn}') \leftarrow \mathcal{R}'.\mathsf{Gen}(1^\lambda)$. Then, if for one of the previous random oracle queries $\mathsf{H}(Z)$ it holds that $(\mathsf{stmt}^*, Z) \in \mathcal{R}'$, it outputs $\mathsf{witn}^* := Z$ and stops (cf. event $\mathsf{bad}_{1,i}$). Otherwise, it samples $\mathsf{ct} \leftarrow\$ \{0,1\}^{\ell_1}$. Note that it never needs the witness witn'.

- For random oracle queries $\mathsf{H}(Z)$ after the ith call to oracle O^*, the reduction checks if $(\mathsf{stmt}^*, Z) \in \mathcal{R}'$. If this holds, it outputs $\mathsf{witn}^* := Z$ and stops (cf. event $\mathsf{bad}_{2,i}$).

First, if $\mathsf{bad}_{1,i}$ occurs, it is clear that the reduction simulates $\mathbf{G}_1$ perfectly until it stops. Also, if $\mathsf{bad}_{1,i}$, it outputs a valid witness witn^* for stmt^*. Similarly, we see that if event $\mathsf{bad}_{2,i}$ occurs, then the reduction simulates $\mathbf{G}_1$ perfectly until it stops and outputs a valid witness witn^* for stmt^*. We obtain that the probability of $\mathsf{bad}_{1,i} \vee \mathsf{bad}_{2,i}$ is upper bounded by the advantage of the reduction against the hardness of $\mathcal{R}'$ relative to $\mathcal{R}'.\mathsf{Gen}$, which is negligible by assumption.
Game $\mathbf{G}_3$: This game is as game $\mathbf{G}_2$, but we change how values ct contained in messages xm_1 are computed in executions of O^*. Namely, we sample $\mathsf{ct} \leftarrow\$ \{0,1\}^{\ell_1}$. Later, before returning signatures σ_b, σ_s, we define $\mathsf{H}(\mathsf{witn}') := \mathsf{ct} \oplus \mathsf{bsm}_2$, where bsm_2 is computed using algorithm $\mathsf{BS.U}_2$ as in algorithm Setup. The bad events that we ruled out in our sequence of games imply that this does not change the view of $\mathcal{A}$. Finally, we note that the only difference between $\mathbf{G}_3$ and the security game against malicious buyers with $b = 1$, using algorithms $\mathsf{Sim}_1, \mathsf{Sim}_{RO}, \mathsf{Sim}_2, \mathsf{Sim}_3$, is the following: In game $\mathbf{G}_3$, the oracle O^* aborts if $\mathsf{SIG.Ver}(\mathsf{pk}_b, \mathsf{tx}, \sigma_b) = 0$ for $\sigma_b := \mathsf{Sell}(St, \mathsf{xm}_2)$. This check is not given in the security game with $b = 1$. However, one can observe that by adaptability of aSIG, this check is redundant. □

B.2 Proofs for the Construction for Unique Signatures

Proof of Lemma 5.5 (Mal. Seller - Unique Signature). We consider an adversary $\mathcal{A}$ against the security of $\mathsf{EXC_u}[\mathsf{SIG}, \mathsf{BS}, \mathsf{PS}]$ against malicious sellers. We define three events in the security game, according to the three possible ways $\mathcal{A}$ can win.

- win_1: This occurs if the security game outputs 1 and $\mathsf{tx} \neq \mathsf{tx}'$.
- win_2: This occurs if the security game outputs 1, $\mathsf{tx} = \mathsf{tx}'$ and $\mathsf{xm}_2 = \perp$.
- win_3: This occurs if the security game outputs 1, $\mathsf{tx} = \mathsf{tx}', \mathsf{xm}_2 \neq \perp$, and $\mathsf{BS.Ver}(\mathsf{pk_{BS}}, \mathsf{sn}, \sigma_{\mathsf{BS}}) = 0$.

First, we bound the probability of $\mathsf{win}_1 \vee \mathsf{win}_2$. Intuitively, this follows from $\mathsf{EUF\text{-}CMA}$ security of SIG, because if one of the events occurs, the adversary came up with a valid signature σ_b for a message tx', for which the game did not compute a signature before. Formally, we give a reduction that runs in the $\mathsf{EUF\text{-}CMA}$ security game. The reduction gets as input a public key pk, and it gets access to a signing oracle $\textsc{Sig}$. Then, the reduction runs $\mathcal{A}$ as in the security game for $\mathsf{EXC_u}[\mathsf{SIG}, \mathsf{BS}, \mathsf{PS}]$ against malicious sellers. Precisely, it runs $\mathcal{A}$, obtains a public key $\mathsf{pk_{BS}}$ and a nonce sn. Then, it runs $(\mathsf{bsm}_1, St) \leftarrow \mathsf{U}_1(\mathsf{pk_{BS}}, \mathsf{sn})$. It sets $\mathsf{pk}_b := \mathsf{pk}$, and passes $\mathsf{bsm}_1, \mathsf{pk}_b$ to $\mathcal{A}$. The adversary outputs $\mathsf{pk}_s, \mathsf{tx}$, and a message xm_1. If $\mathsf{xm}_1 = \perp$ or $\mathsf{xm}_1 = (\mathsf{ct}, \pi)$ and $\mathsf{PVer}(\mathsf{stmt}, \pi) = 0$ for stmt as in algorithm Buy, the reduction sends $\mathsf{xm}_2 := \perp$ to $\mathcal{A}$. Otherwise, it queries a signature $\sigma_b' \leftarrow \textsc{Sig}(\mathsf{tx})$ from the signing oracle and sets $\mathsf{xm}_2 := \sigma_b'$. The reduction passes xm_2 to $\mathcal{A}$ and obtains $\mathsf{tx}', \sigma_b, \sigma_s$ in return. If $\mathsf{win}_1 \vee \mathsf{win}_2$ occurs, it returns (tx', σ_b) to its game. Otherwise, it aborts.

It is clear that the reduction perfectly simulates the game for $\mathcal{A}$. Also, note that the pair (tx', σ_b) that the reduction outputs in the end is valid, i.e. $\mathsf{SIG.Ver}(\mathsf{pk}, \mathsf{tx}', \sigma_b) = 1$, by definition of $\mathsf{win}_1 \vee \mathsf{win}_2$. Further, note that if win_1 occurs, the reduction did only query oracle $\textsc{Sig}$ on input $\mathsf{tx} \neq \mathsf{tx}'$, and not on input tx'. Similarly, if win_2 occurs, the reduction did not query $\textsc{Sig}$ at all. Therefore, the probability of $\mathsf{win}_1 \vee \mathsf{win}_2$ can be upper bounded by the probability that the reduction wins the $\mathsf{EUF\text{-}CMA}$ game. This is negligible by assumption.

It remains to bound the probability of event win_3. Intuitively, this should follow from the soundness of PS. Recall that win_3 occurs, if $\mathsf{tx} = \mathsf{tx}', \mathsf{xm}_2 \neq \perp$, and $\mathsf{BS.Ver}(\mathsf{pk_{BS}}, \mathsf{sn}, \sigma_{\mathsf{BS}}) = 0$. In particular, if $\mathsf{xm}_2 \neq \perp$, we know that for $\mathsf{stmt} = (\mathsf{pk_{BS}}, \mathsf{pk}_s, \mathsf{tx}, \mathsf{bsm}_1, \mathsf{ct})$ and $\mathsf{xm}_1 = (\mathsf{ct}, \pi)$ we have $\mathsf{PVer}(\mathsf{stmt}, \pi) = 1$. We assume towards contradiction that there exists a witness witn such that $(\mathsf{stmt}, \mathsf{witn}) \in \mathcal{R}$, i.e. stmt is a yes-instance. Then, by definition of $\mathcal{R}$ and unique signatures, we know that the first component of witn

is σ_b. and that there is a string ρ such that $\mathsf{ct} = \mathsf{H}(\sigma_s) \oplus \mathsf{BS.S}(\mathsf{sk}_{\mathsf{BS}}, \mathsf{bsm}_1; \rho)$. In combination, we get

$$\begin{aligned}\mathsf{BS.S}(\mathsf{sk}_{\mathsf{BS}}, \mathsf{bsm}_1; \rho) &= \mathsf{ct} \oplus \mathsf{H}(\sigma_s)\\ &= \mathsf{Get}(\mathsf{xpar}, \mathsf{xm}_1, \mathsf{xm}_2, \sigma_b, \sigma_s),\end{aligned}$$

by definition of algorithm Get. Recall that

$$\begin{aligned}\sigma_{\mathsf{BS}} &\leftarrow \mathsf{BS.U}_2(St, \mathsf{Get}(\mathsf{xpar}, \mathsf{xm}_1, \mathsf{xm}_2, \sigma_b, \sigma_s))\\ &= \mathsf{BS.U}_2(St, \mathsf{BS.S}(\mathsf{sk}_{\mathsf{BS}}, \mathsf{bsm}_1; \rho)).\end{aligned}$$

Using completeness of BS, we see that $\mathsf{BS.Ver}(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sn}, \sigma_{\mathsf{BS}}) = 1$. A contradiction.

In summary, we showed that stmt is not a yes-instance, violating soundness of PS. Therefore, the probability of win_3 is negligible. □

Proof of Lemma 5.6 (Mal. Buyer - Unique Signature). We define algorithms $\mathsf{Sim}_1, \mathsf{Sim}_{RO}, \mathsf{Sim}_2, \mathsf{Sim}_3$, and then we show indistinguishability. The algorithms keep a list L containing tuples of the form $(\mathsf{tx}, \mathsf{pk}_s, \mathsf{ct})$. Algorithm $\mathsf{Sim}_1(\mathsf{xpar}, \mathsf{sk}_s)$ is as follows:

1. Compute $\sigma_s \leftarrow \mathsf{SIG.Sig}(\mathsf{sk}_s, \mathsf{tx})$, abort if $\mathsf{H}(\sigma_s)$ is already defined.
2. Sample $\mathsf{ct} \leftarrow_{\$} \{0,1\}^{\ell_1}$.
3. Set $\mathsf{stmt} := (\mathsf{pk}_{\mathsf{BS}}, \mathsf{pk}_s, \mathsf{tx}, \mathsf{bsm}_1, \mathsf{ct})$ and compute $\pi \leftarrow \mathsf{PSim}(\mathsf{stmt})$.
4. Insert $(\mathsf{tx}, \mathsf{pk}_s, \mathsf{ct})$ into L.
5. Return $\mathsf{xm}_1 := (\mathsf{ct}, \pi)$.

Algorithm Sim_{RO} simulates the random oracle honestly. However, on a random oracle query $\mathsf{H}(x)$, it aborts if there is an entry $(\mathsf{tx}, \mathsf{pk}_s, \mathsf{ct})$ in L such that $\mathsf{SIG.Ver}(\mathsf{pk}_s, \mathsf{tx}, x) = 1$. Algorithm $\mathsf{Sim}_2(\mathsf{xm}_2)$ parses $\mathsf{xm}_2 = \sigma_b$ and returns $\mathsf{SIG.Ver}(\mathsf{pk}_b, \mathsf{tx}, \sigma_b)$. Algorithm $\mathsf{Sim}_3(\mathsf{xm}_2, \mathsf{bsm}_2)$ removes the entry $(\mathsf{tx}, \mathsf{pk}_s, \mathsf{ct})$ from L and defines $\mathsf{H}(\sigma_s) := \mathsf{bsm}_2 \oplus \mathsf{ct}$.

Next, we present a sequence of games to show that algorithms $\mathsf{Sim}_1, \mathsf{Sim}_{RO}, \mathsf{Sim}_2, \mathsf{Sim}_3$ satisfy the indistinguishability that is required by the security definition.

Game $\mathbf{G}_0$: This is the security game against malicious buyers with $b = 0$. Recall that in this game, a key pair $(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sk}_{\mathsf{BS}})$ is sampled. Then, the adversary $\mathcal{A}$ gets access to a signer oracle O and an oracle O^*. When called by $\mathcal{A}$, oracle O^* samples a key pair $(\mathsf{pk}_s, \mathsf{sk}_s) \leftarrow \mathsf{SIG.Gen}(1^\lambda)$, gives pk_s to $\mathcal{A}$ and obtains a key pk_b, a transaction tx, and a message bsm_1 in return. Then, it runs algorithm Setup. Concretely, it computes bsm_2 and σ_s, defines ciphertext ct and computes a proof π as in the scheme. Then, it sets $\mathsf{xm}_1 := (\mathsf{ct}, \pi)$ and sends xm_1 to $\mathcal{A}$. The adversary responds with a message xm_2. If xm_2 is a valid signature σ_b for tx with respect to pk_b, the game outputs σ_b, σ_s. Otherwise, it aborts. Finally, the game outputs whatever $\mathcal{A}$ outputs.

Game $\mathbf{G}_1$: This game is as $\mathbf{G}_0$, but we change how the proof π contained in message xm_1 is computed by oracle O^*. Before, it was computed via $\pi \leftarrow \mathsf{PProve}(\mathsf{stmt}, \mathsf{witn})$, where stmt and witn are as in algorithm Setup. In game $\mathbf{G}_1$, we compute it using the zero-knowledge simulator $\mathsf{PS.PSim}$ via $\pi \leftarrow \mathsf{PSim}(\mathsf{stmt})$. By the zero-knowledge property of PS, games $\mathbf{G}_0$ and $\mathbf{G}_1$ are indistinguishable.

Game $\mathbf{G}_2$: In this game, we define bad events bad_1 and bad_2, and abort if one of the two occurs. To do so, we introduce a list L that contains tuples $(\mathsf{tx}, \mathsf{pk}_s, \mathsf{ct})$. Whenever oracle O^* computes the signature σ_s as part of algorithm Setup, and $\mathsf{H}(\sigma_s)$ is already defined, we say that event bad_1 occurs and the game aborts. Otherwise, the game continues the execution of algorithm Setup and inserts the entry $(\mathsf{tx}, \mathsf{pk}_s, \mathsf{ct})$ into L. Later, as soon as the oracle O^* returns the signatures σ_b, σ_s, it removes this entry $(\mathsf{tx}, \mathsf{pk}_s, \mathsf{ct})$ from L. Furthermore, we introduce an event bad_2 that occurs if in a random oracle query $\mathsf{H}(x)$ there is an entry $(\mathsf{tx}, \mathsf{pk}_s, \mathsf{ct})$ in L such that $\mathsf{SIG.Ver}(\mathsf{pk}_s, \mathsf{tx}, x) = 1$. If this event occurs, the game aborts. To show indistinguishability of $\mathbf{G}_1$ and $\mathbf{G}_2$, it is sufficient to bound the probability of event $\mathsf{bad}_1 \vee \mathsf{bad}_2$. To do this, we write

$$\mathsf{bad}_1 \vee \mathsf{bad}_2 = \bigvee_{i \in [Q]} \mathsf{bad}_{1,i} \vee \mathsf{bad}_{2,i},$$

where Q is the number of queries to oracle O^*, and $\mathsf{bad}_{1,i}$ (resp. $\mathsf{bad}_{2,i}$) denotes the event that bad_2 (resp. bad_2) occurs for the entry in L that is inserted in the ith query to O^*. As Q is polynomial, it is sufficient to bound $\mathsf{bad}_{1,i} \vee \mathsf{bad}_{2,i}$ for all i. To this end, we sketch a reduction from the EUF-CMA security of SIG. The reduction gets as input a public key pk and it gets access to a signing oracle Sig. It will not make use of Sig. The reduction simulates $\mathbf{G}_1$ as it is, except for the ith call to oracle O^*, and the random oracle simulation of H:

- In the ith call to oracle O^*, the reduction sets $\mathsf{pk}_s := \mathsf{pk}$, instead of sampling the pair $(\mathsf{pk}_s, \mathsf{sk}_s)$ on its own. Also, it does not compute σ_s as in the game. Instead, if for one of the previous random oracle queries $\mathsf{H}(x)$ it holds that x is a valid signature for tx with respect to pk_s, it outputs (tx, x) to the EUF-CMA game and stops (cf. $\mathsf{bad}_{1,i}$). Otherwise, it samples $\mathsf{ct} \leftarrow_\$ \{0,1\}^{\ell_1}$ at random.
- To simulate random oracle queries $\mathsf{H}(x)$ after the ith call to oracle O^*, the reduction checks if $\mathsf{BS.Ver}(\mathsf{pk}, \mathsf{tx}, x) = 1$. If this holds, it returns (tx, x) to its game and stops (cf. $\mathsf{bad}_{2,i}$).

To argue that the reduction perfectly simulates game $\mathbf{G}_1$ until it stops, it is sufficient to consider the distribution of ct. First, if event $\mathsf{bad}_{1,i}$ occurs, the simulation is clearly perfect until the reduction terminates. Also, if event $\mathsf{bad}_{1,i}$ does not occur, in $\mathbf{G}_1$, the value ct is distributed uniformly. Note that due to uniqueness of signatures, the reduction can efficiently check if $\mathsf{bad}_{1,i}$ occurs. Finally, we see that if event $\mathsf{bad}_{1,i} \vee \mathsf{bad}_{2,i}$ occurs, then the reduction outputs a valid forgery (tx, x). As the reduction never used its signing oracle, we obtain that the probability of $\mathsf{bad}_{1,i} \vee \mathsf{bad}_{2,i}$ is upper bounded by the advantage of the reduction against the EUF-CMA security of SIG, which is negligible by assumption.
Game $\mathbf{G}_3$: This game is as game $\mathbf{G}_2$, but we change how ciphertexts ct are simulated in executions of oracle O^*. Namely, we sample $\mathsf{ct} \leftarrow_\$ \{0,1\}^{\ell_1}$. Later, before the oracle returns signatures σ_b, σ_s, it defines $\mathsf{H}(\sigma_s) := \mathsf{ct} \oplus \mathsf{bsm}_2$, where bsm_2 is computed using algorithm $\mathsf{BS.U}_2$ as in algorithm Setup. Due to the bad events and aborts that we introduced in previous games, we see that this change does not change the view of the adversary. Finally, note that the security game with $b = 1$, using algorithms $\mathsf{Sim}_1, \mathsf{Sim}_{RO}$, $\mathsf{Sim}_2, \mathsf{Sim}_3$, is exactly the same as $\mathbf{G}_3$, finishing the proof. □

B.3 Proofs for the BLS Cut-and-Choose Construction

Proof of Lemma 5.9 (Mal. Seller - BLS). Consider an adversary $\mathcal{A}$ against the security of $\mathsf{EXC}^{\mathsf{cc}}_{\mathsf{BLS}}[\mathsf{SIG}, \mathsf{BS}]$ against malicious sellers. We define three events in the security game, following the three possible ways $\mathcal{A}$ can win.

- win_1: This occurs if the security game outputs 1 and $\mathsf{tx} \neq \mathsf{tx}'$.
- win_2: This occurs if the security game outputs 1, $\mathsf{tx} = \mathsf{tx}'$ and $\mathsf{xm}_2 = \perp$.
- win_3: This occurs if the security game outputs 1, $\mathsf{tx} = \mathsf{tx}', \mathsf{xm}_2 \neq \perp$, and $\mathsf{BS.Ver}(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sn}, \sigma_{\mathsf{BS}}) = 0$.

First, we bound the probability of $\mathsf{win}_1 \vee \mathsf{win}_2$. Intuitively, this follows from EUF-CMA security of SIG, because if one of the events occurs, the adversary came up with a valid signature σ_b for a message tx', for which the game did not compute a signature before. Formally, we give a reduction that runs in the EUF-CMA security game. The reduction gets as input a public key pk, and it gets access to a signing oracle Sig. Then, the reduction runs $\mathcal{A}$ as in the security game for $\mathsf{EXC}^{\mathsf{cc}}_{\mathsf{BLS}}[\mathsf{SIG}, \mathsf{BS}]$ against malicious sellers. Precisely, it runs $\mathcal{A}$, obtains a public key $\mathsf{pk}_{\mathsf{BS}}$ and a nonce sn. Then, it runs $(\mathsf{bsm}_1, St) \leftarrow \mathsf{U}_1(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sn})$. It sets $\mathsf{pk}_b := \mathsf{pk}$, and passes $\mathsf{bsm}_1, \mathsf{pk}_b$ to $\mathcal{A}$. The adversary outputs $\mathsf{pk}_s, \mathsf{tx}$, and a message xm_1. If $\mathsf{xm}_1 = \perp$ the reduction sets $\mathsf{xm}_2 := \perp$. Otherwise, if $\mathsf{xm}_1 = (\mathsf{xm}_{1,1}, \mathsf{xm}_{1,2})$, the reduction starts running algorithm $\mathsf{Buy}(\mathsf{xpar}, \mathsf{sk}_b, \mathsf{xm}_1)$. Concretely, if this algorithm would return $\mathsf{xm}_2 \neq \perp$, it uses its signing oracle Sig on input tx to compute xm_2. Otherwise, it continues with $\mathsf{xm}_2 = \perp$. The reduction passes xm_2 to $\mathcal{A}$ and obtains $\mathsf{tx}', \sigma_b, \sigma_s$ in return. If $\mathsf{win}_1 \vee \mathsf{win}_2$ occurs, it returns (tx', σ_b) to its game. Otherwise, it aborts. It is clear that the reduction perfectly simulates the game for $\mathcal{A}$. Also, note that the pair (tx', σ_b) that the reduction outputs in the end is valid, i.e. $\mathsf{SIG.Ver}(\mathsf{pk}, \mathsf{tx}', \sigma_b) = 1$, by definition of $\mathsf{win}_1 \vee \mathsf{win}_2$. Further, note that if win_1 occurs, the reduction did only query oracle Sig on input $\mathsf{tx} \neq \mathsf{tx}'$, and not on input tx'. Similarly, if win_2 occurs, the reduction did not query Sig at all. Therefore, the probability of $\mathsf{win}_1 \vee \mathsf{win}_2$ can be upper bounded by the probability that the reduction wins the EUF-CMA game. This is negligible by assumption.

It remains to bound the probability of event win_3. Intuitively, this follows via a statistical argument based on the cut-and-choose technique. Recall that win_3 occurs, if $\mathsf{tx} = \mathsf{tx}', \mathsf{xm}_2 \neq \perp$, and $\mathsf{BS.Ver}(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sn}, \sigma_{\mathsf{BS}}) = 0$. We make the following observations.

1. If win_3 occurs, then algorithm Get must have output $\perp$. This is because due $\mathsf{xm}_2 \neq \perp$ we know that $e(\mathsf{bsm}_1, \mathsf{pk}_{\mathsf{BS},k_j}) = e(\mathsf{bsm}_{2,k_j}, g_2)$ for all $j \in [\lambda]$, for notation as in algorithm Buy. Also, assuming Get does not output $\perp$, we know that $e(\mathsf{bsm}_1, \mathsf{pk}_{\mathsf{BS},\bar{k}_j}) = e(\mathsf{bsm}_{2,\bar{k}_j}, g_2)$ for some $j \in [\lambda]$, with notation as in Get. Correctness of algorthm $\mathsf{reconst}_{g_1,0}$ now implies that bsm_2 as computed by Get is a valid second message for the first message bsm_1, which has to lead to a valid blind signature σ_{BS} via algorithm U_2.

2. If algorithm Get outputs $\perp$, then all $\mathsf{bsm}_{2,\bar{k}_j}$ for $j \in [\lambda]$ as computed in Get are invalid, i.e. $e(\mathsf{bsm}_1, \mathsf{pk}_{\mathsf{BS},\bar{k}_j}) \neq e(\mathsf{bsm}_{2,\bar{k}_j}, g_2)$. This is by definition of Get.

3. If win_3 occurs, then all $\sigma_{\bar{k}_j}$ for $j \in [\lambda]$ (as computed in Get) are valid, i.e. for all $j \in [\lambda]$, $\sigma_{\bar{k}_j}$ is the unique value satisfying $\mathsf{SIG.Ver}(\mathsf{pk}_{s,\bar{k}_j}, \mathsf{tx}, \sigma_{\bar{k}_j}) = 1$ for $\mathsf{pk}_{s,\bar{k}_j} := \mathsf{pk}_s \cdot \prod_{i=1}^{\lambda} (\mathsf{coeff}'_i)^{\bar{k}_j^i}$. This is because all σ_{k_j} are valid in the same sense (due to $\mathsf{xm}_2 \neq \perp$) and due to the correctness of algorithm $\mathsf{reconst}_{g_1,\bar{k}_j}$.

Using these three observations, we now finish the statistical argument. For that, consider the moment of the first query of the form $\mathsf{H}_c(\mathsf{xm}_{1,1})$. It is clear that $\mathsf{xm}_{1,1} = ((\mathsf{ct}_j)_{j\in[2\lambda]}, (\mathsf{coeff}_j, \mathsf{coeff}'_j)_{j\in[\lambda]})$ information theoretically determines the polynomials f, f' and therefore all σ_j and $\mathsf{pk}_{\mathsf{BS},j}$ for $j \in [2\lambda]$. Therefore, $\mathsf{xm}_{1,1}$ also determines the values $\mathsf{bsm}_{2,j} := \mathsf{ct}_j \oplus \mathsf{H}(\sigma_j)$ for all $j \in [2\lambda]$. Due to the third observation, these correspond to the values computed in Buy and Get. Due to the first and second observation, and the fact that Buy output $\mathsf{xm}_2 \neq \perp$ if win_3 occurs, we therefore have

$$e(\mathsf{bsm}_1, \mathsf{pk}_{\mathsf{BS},k_j}) = e(\mathsf{bsm}_{2,k_j}, g_2) \text{ for all } j \in [\lambda],$$
$$e(\mathsf{bsm}_1, \mathsf{pk}_{\mathsf{BS},\bar{k}_j}) \neq e(\mathsf{bsm}_{2,\bar{k}_j}, g_2) \text{ for all } j \in [\lambda].$$

Thus, conditioned on win_3, the value $\mathsf{xm}_{1,1}$ fully determines $b_0, \ldots, b_{\lambda-1}$. This means that win_3 can only occur if for some query of the form $\mathsf{H}_c(\mathsf{xm}_{1,1})$, the hash value coincides with the bits $b_0, \ldots, b_{\lambda-1}$ that are determined by $\mathsf{xm}_{1,1}$, which happens with probability $1/2^\lambda$. As there are at most polynomially many queries of this form, the probability of win_3 is negligible, which ends the proof. □

Proof of Lemma 5.10 (Mal. Buyer - BLS). Before we provide algorithms $\mathsf{Sim}_1, \mathsf{Sim}_{RO}, \mathsf{Sim}_2, \mathsf{Sim}_3$, we give a sequence of hybrid games, starting from the security game against malicious buyers with bit $b = 0$ (i.e. computing xm_1 and σ_b honestly via algorithms Setup and Sell). The final game will be equivalent to the security game against malicious buyers game with bit $b = 1$ for the simulators we define then.

Game $\mathbf{G}_0$: We start with game $\mathbf{G}_0$, which is the security game against malicious buyers with bit $b = 0$. To recall, in this game a key pair $(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sk}_{\mathsf{BS}})$ is sampled. Then, $\mathsf{pk}_{\mathsf{BS}}$ is given to the adversary. The adversary also gets access to a signer oracle O for BS simulating $\mathsf{BS.S}(\mathsf{sk}_{\mathsf{BS}}, \cdot)$, and an oracle O^* which is as follows. When called, it first samples a key pair $(\mathsf{pk}_s = g_2^{\mathsf{sk}_s}, \mathsf{sk}_s)$ and outputs pk_s. Then, it gets a key pk_b, a transaction tx, and a message $\mathsf{bsm}_1 \in \mathbb{G}_1$ from the adversary. It sets $\mathsf{xpar} := (\mathsf{pk}_{\mathsf{BS}}, \mathsf{bsm}_1, \mathsf{pk}_b, \mathsf{pk}_s, \mathsf{tx})$ and runs $(\mathsf{xm}_1, St) \leftarrow \mathsf{Setup}(\mathsf{xpar}, \mathsf{sk}_{\mathsf{BS}}, \mathsf{sk}_s)$. In this scheme, xm_1 has the form $\mathsf{xm}_1 = (\mathsf{xm}_{1,1}, \mathsf{xm}_{1,2})$ with $\mathsf{xm}_{1,1} = ((\mathsf{ct}_j)_{j\in[2\lambda]}, (\mathsf{coeff}_j, \mathsf{coeff}'_j)_{j\in[\lambda]})$ and $\mathsf{xm}_{1,2} = (\sigma_{k_j})_{j\in[\lambda]})$. Then, the oracle gives xm_1 to the adversary, obtains $\mathsf{xm}_2 = \sigma_b$, runs Sell (which does not do anything for this scheme), and aborts if σ_b is not valid, i.e. $\mathsf{SIG.Ver}(\mathsf{pk}_b, \mathsf{tx}, \sigma_b) = 0$. Otherwise, it returns σ_b, σ_s to the adversary, where $\sigma_s \leftarrow \mathsf{SIG.Sig}(\mathsf{sk}_s, \mathsf{tx})$. In the end, the game outputs whatever the adversary outputs.

Overall, our goal is to move towards an indistinguishable game, in which xm_1 can be provided without access to $\mathsf{sk}_{\mathsf{BS}}$, and σ_s can be provided only by knowing $\mathsf{bsm}_2 \leftarrow \mathsf{BS.S}(\mathsf{sk}_{\mathsf{BS}}, \mathsf{bsm}_1)$. We will only make changes to oracle O^* and the random oracles involved.

Game $\mathbf{G}_1$: In this game, we change the execution of algorithm Setup in oracle O^*. Namely, in the beginning of the algorithms execution, we now sample uniformly random bits $b_0, \ldots, b_{\lambda-1}$. Then, we compute $\mathsf{xm}_{1,1}$ as before, and abort if $\mathsf{H}_c(\mathsf{xm}_{1,1})$ is already defined. Otherwise, we program $\mathsf{H}_c(\mathsf{xm}_{1,1}) := b_0, \ldots, b_{\lambda-1}$, and continue as before. The probability of such an abort is negligible, due to the entropy of coeff'_1. Thus, $\mathbf{G}_0$ and $\mathbf{G}_1$ are indistinguishable. Observe the effect of this change: We can now define the values $k_j := 2j - b_{j-1}$ and $\bar{k}_j := \bar{k}_j := 2j - (1 - b_{j-1})$ before we compute $\mathsf{xm}_{1,1}$.

Game $\mathbf{G}_2$: In this game we introduce a bad event bad and let the game abort if it occurs. The event occurs if in some interaction between the adversary and oracle O^*, one of the following happens.

- bad_1: When the game computes the values $(\mathsf{ct}_j)_{j\in[2\lambda]}$ during the execution of Setup, the hash value $\mathsf{H}(\sigma_{\bar{k}_j})$ is already defined for some $j \in [\lambda]$.

- bad_2: After the game computes the values $(\mathsf{ct}_j)_{j\in[2\lambda]}$ during the execution of Setup, but before the game gives σ_s to the adversary in the same interaction, a query $\mathsf{H}(\sigma_{\bar{k}_j})$ is made for some $j \in [\lambda]$.

We have

$$\mathsf{bad} = \mathsf{bad}_1 \vee \mathsf{bad}_2 = \bigvee_{i\in[Q]} \mathsf{bad}_{1,i} \vee \mathsf{bad}_{2,i},$$

where Q is the number of queries to oracle O^*, and the event $\mathsf{bad}_{1,i}$ (resp. $\mathsf{bad}_{2,i}$) occurs if bad_1 (resp. bad_2) occurs in the ith interaction between the adversary and O^*. As Q is polynomial, it is sufficient to bound $\mathsf{bad}_{1,i} \vee \mathsf{bad}_{2,i}$ for all i. To this end, we sketch a reduction from the $\mathsf{EUF\text{-}CMA}$ security of SIG. The reduction gets as input a public key pk and it gets access to a signing oracle $\textsc{Sig}$. It will not make use of $\textsc{Sig}$. The reduction simulates $\mathbf{G}_1$ as it is, except for the ith call to oracle O^*, and the random oracle simulation of H:

- In the ith call to oracle O^*, the reduction sets $\mathsf{pk}_s := \mathsf{pk}$, instead of sampling the pair $(\mathsf{pk}_s, \mathsf{sk}_s)$ on its own.
- This means that it can not define the polynomial f' as in the game explicitly. Instead, the reduction runs $((\mathsf{sk}_{s,k_j})_{j\in[\lambda]}, (\mathsf{coeff}'_j)_{j\in[\lambda]}) \leftarrow \mathsf{polyGen}_{g_2,p}(\lambda, \mathsf{pk}_s, (k_j)_{j\in[\lambda]})$.
- The reduction checks checks if event $\mathsf{bad}_{1,i}$ occurs, by checking for each previous random oracle query $\mathsf{H}(x)$ if $\mathsf{SIG.Ver}(\mathsf{pk}_{s,\bar{k}_j}, \mathsf{tx}, x) = 1$ for some $j \in [\lambda]$, where $\mathsf{pk}_{s,\bar{k}_j} := \mathsf{pk}_s \cdot \prod_{i=1}^{\lambda} (\mathsf{coeff}'_i)^{\bar{k}_j^i}$. Note that this check is correct due to the uniqueness of SIG. If $\mathsf{bad}_{1,i}$ occurs, say for $j^* \in [\lambda]$, the reduction computes a signature σ for tx via $\sigma := \mathsf{reconst}_{g,0}((j^*, x), (k_i, \sigma_{s,k_i})_{i\in[\lambda]})$. Then, it outputs (tx, σ) as a forgery to the $\mathsf{EUF\text{-}CMA}$ game. If $\mathsf{bad}_{1,i}$ does not occur, it continues by sampling all $\mathsf{ct}_{\bar{k}_j}$ at random.
- The reduction can check if event $\mathsf{bad}_{2,i}$ occurs similar to event $\mathsf{bad}_{1,i}$ using algorithm $\mathsf{SIG.Ver}$ whenever the adversary queries H. If $\mathsf{bad}_{2,i}$ occurs, the reduction computes a signature σ in a similar way as above and outputs (tx, σ) as a forgery to the $\mathsf{EUF\text{-}CMA}$ game.
- If the reduction has to output σ_s to the adversary in the ith interaction, the reduction aborts.

It is easy to see that until the reduction aborts, it perfectly simulates $\mathbf{G}_1$ for the adversary. This is due to the correctness of algorithm $\mathsf{polyGen}_{g_2,p}$. Also, if $\mathsf{bad}_{1,i} \vee \mathsf{bad}_{2,i}$ occurs, the reduction does not abort and returns a valid forgery, following from the correctness of algorithm $\mathsf{reconst}_{g,0}$. Also, the reduction never uses its signing oracle. This implies that the probability of $\mathsf{bad}_{1,i} \vee \mathsf{bad}_{2,i}$ is negligible, by the $\mathsf{EUF\text{-}CMA}$ security of SIG.

Game $\mathbf{G}_3$: In game $\mathbf{G}_3$, we change how the values $\mathsf{ct}_{\bar{k}_j}$ for $j \in [\lambda]$ are computed in executions of algorithm Setup in oracle O^*. Concretely, while they were computed as $\mathsf{ct}_{\bar{k}_j} = \mathsf{H}(\sigma_{\bar{k}_j}) \oplus \mathsf{bsm}_{2,\bar{k}_j}$ before, we now sample them at random as $\mathsf{ct}_{\bar{k}_j} \leftarrow\!\!\$\ \{0,1\}^{\ell}$. Later, before giving σ_s to the adversary in the same interaction, we let the game program $\mathsf{H}(\sigma_{\bar{k}_j}) := \mathsf{ct}_{\bar{k}_j} \oplus \mathsf{bsm}_{2,\bar{k}_j}$. Clearly, this does not change the view of the adversary due to the bad event and abort that we introduced in the previous game.

Game $\mathbf{G}_4$: In game $\mathbf{G}_4$, we change the oracle O^* again. Namely, note that due to the previous change, we do not need the values $\mathsf{bsm}_{2,\bar{k}_j}$ to compute xm_1, but only once we output σ_s. This will allow us to compute xm_1 without access to $\mathsf{sk}_{\mathsf{BS}}$. Namely, we will now compute the values coeff_j used during the computation of xm_1 as

$$((\mathsf{sk}_{\mathsf{BS},k_j})_{j\in[\lambda]}, (\mathsf{coeff}_j)_{j\in[\lambda]}) \leftarrow \mathsf{polyGen}_{g_2,p}(\lambda, \mathsf{pk}_{\mathsf{BS}}, (k_j)_{j\in[\lambda]}).$$

Later, before outputting σ_s to the adversary, we compute the values $\mathsf{bsm}_{2,\bar{k}_j}$ via by first computing $\mathsf{bsm}_2 \leftarrow \mathsf{BS.S}(\mathsf{sk}_{\mathsf{BS}}, \mathsf{bsm}_1)$, and then computing

$$\mathsf{bsm}_{2,\bar{k}_j} := \mathsf{reconst}_{g_1,\bar{k}_j}((0, \mathsf{bsm}_2), (k_i, \mathsf{bsm}_{k_i})_{i\in[\lambda]}) \text{ for all } j \in [\lambda].$$

Then, we continue as in $\mathbf{G}_3$.

Summarizing the implications of these changes, we now compute the messages xm_1 without access to $\mathsf{sk}_{\mathsf{BS}}$. Further, after we obtain $\mathsf{xm}_2 = \sigma_b$ and before we output σ_s, we do not need direct access to $\mathsf{sk}_{\mathsf{BS}}$, but only to $\mathsf{bsm}_2 \leftarrow \mathsf{BS.S}(\mathsf{sk}_{\mathsf{BS}}, \mathsf{bsm}_1)$. This can easily be captured by algorithms $\mathsf{Sim}_1, \mathsf{Sim}_{RO}$, $\mathsf{Sim}_2, \mathsf{Sim}_3$ as desired. Then, $\mathbf{G}_3$ is identical to the security game against malicious buyers with bit $b = 1$, showing the claim. □

B.4 Proofs for the Adaptor Cut-and-Choose Construction

Proof of Lemma 5.11 (Mal. Seller - Adaptor CC). We consider an adversary $\mathcal{A}$ against the security of $\mathsf{EXC}_{\mathsf{a}}^{\mathsf{cc}}[\mathsf{SIG}, \mathsf{aSIG}, \mathsf{BS}]$ against malicious sellers. We define three events in the security game, following the three possible ways $\mathcal{A}$ can win.

- win_1: This occurs if the security game outputs 1 and $\mathsf{tx} \neq \mathsf{tx}'$.
- win_2: This occurs if the security game outputs 1, $\mathsf{tx} = \mathsf{tx}'$ and $\mathsf{xm}_2 = \perp$.
- win_3: This occurs if the security game outputs 1, $\mathsf{tx} = \mathsf{tx}', \mathsf{xm}_2 \neq \perp$, and $\mathsf{BS.Ver}(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sn}, \sigma_{\mathsf{BS}}) = 0$.

First, we bound the probability of $\mathsf{win}_1 \vee \mathsf{win}_2$. Intuitively if one of the events occurs, the adversary came up with a valid signature σ_b for a message tx', for which the game did not compute a signature or pre-signature before. Formally, we give a reduction that runs in the $\mathsf{aEUF\text{-}CMA}$ security game of aSIG. The reduction gets pk as input and access to oracles $\textsc{Sig}$ and $\textsc{PreSig}$. It runs $\mathcal{A}$ and obtains a public key $\mathsf{pk}_{\mathsf{BS}}$ and a message sn from $\mathcal{A}$. Then, it runs $(\mathsf{bsm}_1, St) \leftarrow \mathsf{U}_1(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sn})$. It sets $\mathsf{pk}_b := \mathsf{pk}$. Next, it gives pk_b and bsm_1 to $\mathcal{A}$, which outputs a key pk_s, a transaction tx and a message xm_1. If $\mathsf{xm}_1 = \perp$ the reduction sets $\mathsf{xm}_2 := \perp$. Else if $\mathsf{xm}_1 = (\mathsf{xm}_{1,1}, \mathsf{xm}_{1,2})$, the reduction checks the validity of xm_1 similar to what is done in algorithm $\mathsf{Buy}(\mathsf{xpar}, \mathsf{sk}_b, \mathsf{xm}_1)$. Note that the unknown secret key sk_b is only used by Buy if it does not output $\perp$. In this case, the reduction uses oracle $\textsc{PreSig}$ via $\tilde{\sigma}_b \leftarrow \textsc{PreSig}(\mathsf{tx}, \mathsf{stmt}')$ and sets $\mathsf{xm}_2 := \tilde{\sigma}_b$. Otherwise, it sets $\mathsf{xm}_2 := \perp$. The reduction passes xm_2 to $\mathcal{A}$ and obtains $\mathsf{tx}', \sigma_b, \sigma_s$ in return. If $\mathsf{win}_1 \vee \mathsf{win}_2$ occurs, it returns (tx', σ_b) to its game. Otherwise, it aborts. It is clear that the reduction perfectly simulates the game for $\mathcal{A}$. Also, note that the pair (tx', σ_b) that the reduction outputs in the end is valid, i.e. $\mathsf{SIG.Ver}(\mathsf{pk}, \mathsf{tx}', \sigma_b) = 1$, by definition of $\mathsf{win}_1 \vee \mathsf{win}_2$. Further, note that if win_1 occurs, the reduction did only query oracle $\textsc{PreSig}$ on input $\mathsf{tx} \neq \mathsf{tx}'$, and not on input tx'. Similarly, if win_2 occurs, the reduction did not query $\textsc{PreSig}$ at all. In both cases, the reduction did never query oracle $\textsc{Sig}$. Therefore, the probability of $\mathsf{win}_1 \vee \mathsf{win}_2$ can be upper bounded by the probability that the reduction wins the $\mathsf{aEUF\text{-}CMA}$ game. This is negligible by assumption.

It remains to bound the probability of event win_3. Recall that win_3 occurs, if $\mathsf{tx} = \mathsf{tx}', \mathsf{xm}_2 \neq \perp$, and $\mathsf{BS.Ver}(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sn}, \sigma_{\mathsf{BS}}) = 0$. We bound the probability of event win_3 by partitioning it into two sub-events.

- $\mathsf{win}_{3,1}$: This event occurs, if win_3 occurs, and for $y := \mathsf{Ext}(\tilde{\sigma}_b, \sigma_b)$ computed in Get, we have $g^y \neq Y$.
- $\mathsf{win}_{3,2}$: This event occurs, if win_3 occurs, and for $y := \mathsf{Ext}(\tilde{\sigma}_b, \sigma_b)$ computed in Get, we have $g^y = Y$.

Clearly, it is sufficient to bound the probability of $\mathsf{win}_{3,1}$ and $\mathsf{win}_{3,2}$ separately. We start with event $\mathsf{win}_{3,1}$. Intuitively, in this case, the adversary managed to turn the pre-signature $\mathsf{xm}_2 = \tilde{\sigma}_b$ into a valid signature, but we can not extract a witness, contradicting the witness extractability of aSIG. Formally, we give a reduction against the witness extractability of aSIG. The reduction gets pk as input and access to oracles $\textsc{Sig}$ and $\textsc{PreSig}$. It runs $\mathcal{A}$ and obtains a public key $\mathsf{pk}_{\mathsf{BS}}$ and a message sn from $\mathcal{A}$. Next, it runs $(\mathsf{bsm}_1, St) \leftarrow \mathsf{U}_1(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sn})$, sets $\mathsf{pk}_b := \mathsf{pk}$, and gives pk_b and bsm_1 to $\mathcal{A}$, which outputs a key pk_s, a transaction tx and a message xm_1. If $\mathsf{xm}_1 = \perp$ or π does not verify, the reduction aborts. Otherwise, it parses $\mathsf{xm}_1 = (\mathsf{xm}_{1,1}, \mathsf{xm}_{1,2})$, and $\mathsf{xm}_{1,1} = (Y, (\mathsf{ct}_j)_{j \in [2\lambda]}, (\mathsf{coeff}_j, \mathsf{coeff}'_j)_{j \in [\lambda]})$ and outputs (tx, Y) to its game. It obtains a pre-signature $\tilde{\sigma}$ in return and sets $\mathsf{xm}_2 := \tilde{\sigma}_b := \tilde{\sigma}$. Then, the reduction passes xm_2 to $\mathcal{A}$ and obtains tx', σ_b, and σ_s in return. If the event $\mathsf{win}_{3,1}$ occurs, it outputs σ_b to its game. Note that the reduction did not use the oracles $\textsc{Sig}$ and $\textsc{PreSig}$ at all. This shows that the probability of $\mathsf{win}_{3,1}$ is negligible, assuming witness extractability of aSIG.

Finally, we bound the probability of $\mathsf{win}_{3,2}$ using a statistical argument. To this end, we make the following observations.

1. If $\mathsf{win}_{3,2}$ occurs, then algorithm Get must have output $\perp$. This is because due $\mathsf{xm}_2 \neq \perp$ we know that $e(\mathsf{bsm}_1, \mathsf{pk}_{\mathsf{BS},k_j}) = e(\mathsf{bsm}_{2,k_j}, g_2)$ for all $j \in [\lambda]$, for notation as in algorithm Buy. Also, assuming Get does not output $\perp$, we know that $e(\mathsf{bsm}_1, \mathsf{pk}_{\mathsf{BS},\bar{k}_j}) = e(\mathsf{bsm}_{2,\bar{k}_j}, g_2)$ for some $j \in [\lambda]$, with notation as in Get. Correctness of algorthm $\mathsf{reconst}_{g_1,0}$ now implies that bsm_2 as computed by Get is a valid second message for the first message bsm_1, which has to lead to a valid blind signature σ_{BS} via algorithm U_2.
2. If algorithm Get outputs $\perp$, then all $\mathsf{bsm}_{2,\bar{k}_j}$ for $j \in [\lambda]$ as computed in Get are invalid, i.e. $e(\mathsf{bsm}_1, \mathsf{pk}_{\mathsf{BS},\bar{k}_j}) \neq e(\mathsf{bsm}_{2,\bar{k}_j}, g_2)$. This is by definition of Get.

3. If $\mathsf{win}_{3,2}$ occurs, then the polynomial f' computed by Get is exactly the same polynomial as defined by the values coeff'_j. This is because in this event we assume $g^y = Y$, and as $\mathsf{xm}_2 \neq \perp$ we know that $g^{y_{k_j}} = Y_{k_j}$ for all $j \in [\lambda]$. Therefore, correctness of algorithm $\mathsf{reconst}_q$ shows the claim.

Using these three observations, we now finish the statistical argument. For that, consider the moment of the first query of the form $\mathsf{H}_c(\mathsf{xm}_{1,1})$. It is clear that $\mathsf{xm}_{1,1} = (Y, (\mathsf{ct}_j)_{j\in[2\lambda]}, (\mathsf{coeff}_j, \mathsf{coeff}'_j)_{j\in[\lambda]})$ information theoretically determines the polynomials f, f' and therefore all $y_j = f'(j)$ and $\mathsf{pk}_{\mathsf{BS},j}$ for $j \in [2\lambda]$. Therefore, $\mathsf{xm}_{1,1}$ also determines the values $\mathsf{bsm}_{2,j} := \mathsf{ct}_j \oplus \mathsf{H}(y_j)$ for all $j \in [2\lambda]$. By the third observation, we know that these correspond to the values computed in Buy and Get. Due to the first and second observation, and the fact that Buy output $\mathsf{xm}_2 \neq \perp$ if $\mathsf{win}_{3,2}$ occurs, we therefore have

$$e(\mathsf{bsm}_1, \mathsf{pk}_{\mathsf{BS},k_j}) = e(\mathsf{bsm}_{2,k_j}, g_2) \text{ for all } j \in [\lambda],$$
$$e(\mathsf{bsm}_1, \mathsf{pk}_{\mathsf{BS},\bar{k}_j}) \neq e(\mathsf{bsm}_{2,\bar{k}_j}, g_2) \text{ for all } j \in [\lambda].$$

Thus, conditioned on $\mathsf{win}_{3,2}$, the value $\mathsf{xm}_{1,1}$ fully determines $b_0, \ldots, b_{\lambda-1}$. This means that $\mathsf{win}_{3,2}$ can only occur if for some query of the form $\mathsf{H}_c(\mathsf{xm}_{1,1})$, the hash value coincides with the bits $b_0, \ldots, b_{\lambda-1}$ that are determined by $\mathsf{xm}_{1,1}$, which happens with probability $1/2^\lambda$. As there are at most polynomially many queries of this form, the probability of $\mathsf{win}_{3,2}$ is negligible, which ends the proof. □

Proof of Lemma 5.12 (Mal. Buyer - Adaptor CC). Before we provide algorithms $\mathsf{Sim}_1, \mathsf{Sim}_{RO}, \mathsf{Sim}_2, \mathsf{Sim}_3$, we give a sequence of hybrid games, starting from the security game against malicious buyers with bit $b = 0$ (i.e. computing xm_1 and σ_b honestly via algorithms Setup and Sell). The final game will be equivalent to the security game against malicious buyers game with bit $b = 1$ for the simulators we define then.

Game $\mathbf{G}_0$: We start with game $\mathbf{G}_0$, which is the security game against malicious buyers with bit $b = 0$. To recall, in this game a key pair $(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sk}_{\mathsf{BS}})$ is sampled. Then, $\mathsf{pk}_{\mathsf{BS}}$ is given to the adversary. The adversary also gets access to a signer oracle O for BS simulating $\mathsf{BS.S}(\mathsf{sk}_{\mathsf{BS}}, \cdot)$, and an oracle O^* which is as follows. When called, it first samples a key pair $(\mathsf{pk}_s, \mathsf{sk}_s) \leftarrow \mathsf{SIG.Gen}(1^\lambda)$ and outputs pk_s. Then, it gets a key pk_b, a transaction tx, and a message $\mathsf{bsm}_1 \in \mathbb{G}_1$ from the adversary. It sets $\mathsf{xpar} := (\mathsf{pk}_{\mathsf{BS}}, \mathsf{bsm}_1, \mathsf{pk}_b, \mathsf{pk}_s, \mathsf{tx})$ and runs $(\mathsf{xm}_1, St) \leftarrow \mathsf{Setup}(\mathsf{xpar}, \mathsf{sk}_{\mathsf{BS}}, \mathsf{sk}_s)$. In this scheme, xm_1 has the form $\mathsf{xm}_1 = (\mathsf{xm}_{1,1}, \mathsf{xm}_{1,2})$ with $\mathsf{xm}_{1,1} = (Y = g^y, (\mathsf{ct}_j)_{j\in[2\lambda]}, (\mathsf{coeff}_j, \mathsf{coeff}'_j)_{j\in[\lambda]})$ and $\mathsf{xm}_{1,2} = (y_{k_j})_{j\in[\lambda]})$. Then, the oracle gives xm_1 to the adversary, obtains $\mathsf{xm}_2 = \tilde{\sigma}_b$, and runs Sell, which aborts if $\mathsf{PreVer}(\mathsf{pk}_b, \mathsf{tx}, g^y, \tilde{\sigma}_b) = 0$ and computes $\sigma_b := \mathsf{Adapt}(\mathsf{pk}_b, \tilde{\sigma}_b, y)$. Further, the oracle aborts if σ_b is not valid, i.e. $\mathsf{SIG.Ver}(\mathsf{pk}_b, \mathsf{tx}, \sigma_b) = 0$. From now on, we omit this check, which is redundant due to adaptability of SIG. In case there is no abort, the oracle returns σ_b, σ_s to the adversary, where $\sigma_s \leftarrow \mathsf{SIG.Sig}(\mathsf{sk}_s, \mathsf{tx})$. In the end, the game outputs whatever the adversary outputs.

Overall, our goal is to move towards an indistinguishable game, in which xm_1 can be provided without access to $\mathsf{sk}_{\mathsf{BS}}$, and σ_s can be provided only by knowing $\mathsf{bsm}_2 \leftarrow \mathsf{BS.S}(\mathsf{sk}_{\mathsf{BS}}, \mathsf{bsm}_1)$. We will only make changes to oracle O^* and the random oracles involved.

Game $\mathbf{G}_1$: In this game, we change the execution of algorithm Setup in oracle O^*. Namely, in the beginning of the algorithms execution, we now sample uniformly random bits $b_0, \ldots, b_{\lambda-1}$. Then, we compute $\mathsf{xm}_{1,1}$ as before, and abort if $\mathsf{H}_c(\mathsf{xm}_{1,1})$ is already defined. Otherwise, we program $\mathsf{H}_c(\mathsf{xm}_{1,1}) := b_0, \ldots, b_{\lambda-1}$, and continue as before. The probability of such an abort is negligible, due to the entropy of Y. Thus, $\mathbf{G}_0$ and $\mathbf{G}_1$ are indistinguishable. Observe the effect of this change: We can now define the values $k_j := 2j - b_{j-1}$ and $\bar{k}_j := \bar{k}_j := 2j - (1 - b_{j-1})$ before we compute $\mathsf{xm}_{1,1}$.

Game $\mathbf{G}_2$: In this game we introduce a bad event bad and let the game abort if it occurs. The event occurs if in some interaction between the adversary and oracle O^*, one of the following happens.

- bad_1: When the game computes the values $(\mathsf{ct}_j)_{j\in[2\lambda]}$ during the execution of Setup, the hash value $\mathsf{H}(y_{\bar{k}_j})$ is already defined for some $j \in [\lambda]$.
- bad_2: After the game computes the values $(\mathsf{ct}_j)_{j\in[2\lambda]}$ during the execution of Setup, but before the game gives σ_s to the adversary in the same interaction, a query $\mathsf{H}(y_{\bar{k}_j})$ is made for some $j \in [\lambda]$.

We have

$$\mathsf{bad} = \mathsf{bad}_1 \vee \mathsf{bad}_2 = \bigvee_{i\in[Q]} \mathsf{bad}_{1,i} \vee \mathsf{bad}_{2,i},$$

where Q is the number of queries to oracle O^*, and the event $\mathsf{bad}_{1,i}$ (resp. $\mathsf{bad}_{2,i}$) occurs if bad_1 (resp. bad_2) occurs in the ith interaction between the adversary and O^*. As Q is polynomial, it is sufficient to bound $\mathsf{bad}_{1,i} \vee \mathsf{bad}_{2,i}$ for all i. To this end, we sketch a reduction from the DLOG assumption in $\mathbb{G}$. The reduction gets as input a group element Y^*. The reduction simulates $\mathbf{G}_1$ as it is, except for the ith call to oracle O^*, and the random oracle simulation of H:

- In the ith call to oracle O^*, the reduction sets $Y := Y^*$, instead of sampling $y \leftarrow\$ \mathbb{Z}_q$ and setting $Y := g^y$.
- This means that it can not define the polynomial f' as in the game explicitly. Instead, the reduction runs $((y_{k_j})_{j\in[\lambda]}, (\mathsf{coeff}'_j)_{j\in[\lambda]}) \leftarrow \mathsf{polyGen}_{g,q}(\lambda, Y, (k_j)_{j\in[\lambda]})$.
- The reduction checks checks if event $\mathsf{bad}_{1,i}$ occurs, by checking for each previous random oracle query $\mathsf{H}(x)$ if $g^x = Y_{\bar{k}_j}$ for some $j \in [\lambda]$, where $Y_{\bar{k}_j} := Y \cdot \prod_{i=1}^{\lambda} (\mathsf{coeff}'_i)^{\bar{k}_j^i}$. If $\mathsf{bad}_{1,i}$ occurs, say for $j^* \in [\lambda]$, the reduction computes the discrete logarithm y of Y via $f'(X) := \mathsf{reconst}_q((j^*, x), (k_i, y_{k_j})_{i\in[\lambda]})$and $y = f'(0)$. Then, it outputs y as a DLOG solution. If $\mathsf{bad}_{1,i}$ does not occur, it continues by sampling all $\mathsf{ct}_{\bar{k}_j}$ at random.
- The reduction can check if event $\mathsf{bad}_{2,i}$ occurs similar to event $\mathsf{bad}_{1,i}$ using the check $g^x = Y_{\bar{k}_j}$ for all $j \in [\lambda]$ whenever the adversary queries $\mathsf{H}(x)$. If $\mathsf{bad}_{2,i}$ occurs, the reduction computes y in a similar way as above and outputs y as a DLOG solution.
- If the reduction has to output σ_s to the adversary in the ith interaction, the reduction aborts.

It is easy to see that until the reduction aborts, it perfectly simulates $\mathbf{G}_1$ for the adversary. This is due to the correctness of algorithm $\mathsf{polyGen}_{g,q}$. Also, if $\mathsf{bad}_{1,i} \vee \mathsf{bad}_{2,i}$ occurs, the reduction does not abort and returns a valid forgery, following from the correctness of algorithm $\mathsf{reconst}_q$. This implies that the probability of $\mathsf{bad}_{1,i} \vee \mathsf{bad}_{2,i}$ is negligible, by the DLOG assumption in $\mathbb{G}$.

Game $\mathbf{G}_3$: In game $\mathbf{G}_3$, we change how the values $\mathsf{ct}_{\bar{k}_j}$ for $j \in [\lambda]$ are computed in executions of algorithm Setup in oracle O^*. Concretely, while they were computed as $\mathsf{ct}_{\bar{k}_j} = \mathsf{H}(y_{\bar{k}_j}) \oplus \mathsf{bsm}_{2,\bar{k}_j}$ before, we now sample them at random as $\mathsf{ct}_{\bar{k}_j} \leftarrow\$ \{0,1\}^{\ell}$. Later, before giving σ_s to the adversary in the same interaction, we let the game program $\mathsf{H}(y_{\bar{k}_j}) := \mathsf{ct}_{\bar{k}_j} \oplus \mathsf{bsm}_{2,\bar{k}_j}$. Clearly, this does not change the view of the adversary due to the bad event and abort that we introduced in the previous game.

Game $\mathbf{G}_4$: In game $\mathbf{G}_4$, we change the oracle O^* again. Namely, note that due to the previous change, we do not need the values $\mathsf{bsm}_{2,\bar{k}_j}$ to compute xm_1, but only once we output σ_s. This will allow us to compute xm_1 without access to $\mathsf{sk}_{\mathsf{BS}}$. Namely, we will now compute the values coeff_j used during the computation of xm_1 as

$$((\mathsf{sk}_{\mathsf{BS},k_j})_{j\in[\lambda]}, (\mathsf{coeff}_j)_{j\in[\lambda]}) \leftarrow \mathsf{polyGen}_{g_2,p}(\lambda, \mathsf{pk}_{\mathsf{BS}}, (k_j)_{j\in[\lambda]}).$$

Later, before outputting σ_s to the adversary, we compute the values $\mathsf{bsm}_{2,\bar{k}_j}$ via by first computing $\mathsf{bsm}_2 \leftarrow \mathsf{BS.S}(\mathsf{sk}_{\mathsf{BS}}, \mathsf{bsm}_1)$, and then computing

$$\mathsf{bsm}_{2,\bar{k}_j} := \mathsf{reconst}_{g_2,\bar{k}_j}((0, \mathsf{bsm}_2), (k_i, \mathsf{bsm}_{k_i})_{i\in[\lambda]}) \text{ for all } j \in [\lambda].$$

Then, we continue as in $\mathbf{G}_3$.

Summarizing the implications of these changes, we now compute the messages xm_1 without access to $\mathsf{sk}_{\mathsf{BS}}$. Further, after we obtain $\mathsf{xm}_2 = \sigma_b$ and before we output σ_s, we do not need direct access to $\mathsf{sk}_{\mathsf{BS}}$, but only to $\mathsf{bsm}_2 \leftarrow \mathsf{BS.S}(\mathsf{sk}_{\mathsf{BS}}, \mathsf{bsm}_1)$. This can easily be captured by algorithms $\mathsf{Sim}_1, \mathsf{Sim}_{RO}$, $\mathsf{Sim}_2, \mathsf{Sim}_3$ as desired. Then, $\mathbf{G}_3$ is identical to the security game against malicious buyers with bit $b = 1$, showing the claim. □

C Security Proofs of Redeem Protocols

Remark. The key ideas and many steps of our proofs for redeem protocols are very similar, which is why we reuse parts verbatim in different proofs. It is recommended to understand the proofs for the generic construction first, before reading the proofs for the cut-and-choose construction.

C.1 Proofs for the Generic Construction

Proof of Lemma 6.4 (Mal. Service - Generic). To prove the claim, we present an algorithm Ext that takes as input parameters rpar, a promise message $\mathsf{prom} = (\mathsf{ct}, \pi)$, and a list $\mathcal{Q}$ of random oracle queries and outputs a blind signature σ_{BS}. Algorithm $\mathsf{Ext}(\mathsf{rpar}, \mathsf{prom}, \mathcal{Q})$ is as follows:

1. Parse $\mathsf{rpar} = (\mathsf{pk}_{\mathsf{BS}}, \mathsf{pk}_s, \mathsf{tx}, \mathsf{sn})$.
2. Find an entry $((\mathsf{sn}, \sigma_{\mathsf{BS}}), \mathsf{H}(\mathsf{sn}, \sigma_{\mathsf{BS}}))$ in $\mathcal{Q}$, such that $\mathsf{BS.Ver}(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sn}, \sigma_{\mathsf{BS}}) = 1$.
3. If such an entry is found, return σ_{BS}. Otherwise, return $\bot$.

It remains to prove that for this algorithm Ext, the probability that the security game outputs 1 is negligible. In the security game, we define the event win_1 which occurs if $\mathsf{VerPromise}(\mathsf{rpar}, \mathsf{prom}) = 1$ and Ext outputs $\bot$. We also define the event win_2 which occurs if $\mathsf{VerPromise}(\mathsf{rpar}, \mathsf{prom}) = 1$, algorithm Ext outputs a valid blind signature σ_{BS}, but for $\sigma_s \leftarrow \mathsf{Redeem}(\mathsf{rpar}, \mathsf{prom}, \sigma_{\mathsf{BS}})$ we have $\mathsf{SIG.Ver}(\mathsf{pk}_s, \mathsf{tx}, \sigma_s) = 0$. Note that whenever algorithm Ext does not output $\bot$, it outputs a valid blind signature for sn. Therefore, the game outputs 1 if and only if win_1 or win_2 occurs.

First, we upper bound the probability of win_1. If win_1 occurs, we have $\mathsf{PVer}(\mathsf{stmt}, \pi) = 1$ for $\mathsf{stmt} = (\mathsf{pk}_{\mathsf{BS}}, \mathsf{pk}_s, \mathsf{tx}, \mathsf{sn}, \mathsf{ct})$. Further, if Ext outputs $\bot$, then $\mathsf{H}(\mathsf{sn}, \sigma_{\mathsf{BS}})$ is not yet defined, where σ_{BS} is the unique signature that satisfies $\mathsf{BS.Ver}(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sn}, \sigma_{\mathsf{BS}}) = 1$. Therefore, the value $\mathsf{H}(\mathsf{sn}, \sigma_{\mathsf{BS}}) \oplus \mathsf{ct}$ is uniformly random at this point. By smoothness of SIG, we therefore know that the probability that $\mathsf{SIG.Ver}(\mathsf{pk}_s, \mathsf{tx}, \mathsf{ct} \oplus \mathsf{H}(\mathsf{sn}, \sigma_{\mathsf{BS}})) = 1$ is negligible. Thus, assuming win_1 occurs, we have $\mathsf{stmt} \notin \mathcal{L}_\lambda$ with overwhelming probability, violating soundness of PS. Therefore, the probability of win_1 is negligible.

Next, we upper bound the probability of win_2. Note that by definition of algorithm Redeem, if win_2 occurs, we have that

$$\mathsf{SIG.Ver}(\mathsf{pk}_s, \mathsf{tx}, \mathsf{ct} \oplus \mathsf{H}(\mathsf{sn}, \sigma_{\mathsf{BS}})) = 0,$$

where σ_{BS} is output by Ext and satisfies $\mathsf{BS.Ver}(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sn}, \sigma_{\mathsf{BS}}) = 1$. Due to uniqueness of BS, this implies that $\mathsf{stmt} \notin \mathcal{L}_\lambda$, violating the soundness of PS. Therefore, the probability of win_2 is also negligible. □

Proof of Lemma 6.5 (Mal. User - Generic). In order to prove the statement, we provide algorithms $\mathsf{Sim}, \mathsf{Sim}_{RO}$ and Ext that share state.

Simulatability. Algorithms $\mathsf{Sim}, \mathsf{Sim}_{RO}$ simulate promise messages $\mathsf{prom} = (\mathsf{ct}, \pi)$ and the random oracle H. The algorithms share a list L, that stores tuples $(\mathsf{sn}, \mathsf{ct}, \sigma_s)$. The list is initially empty. Algorithm $\mathsf{Sim}(\mathsf{rpar}, \mathsf{sk}_s)$ is as follows:

1. Parse $\mathsf{rpar} = (\mathsf{pk}_{\mathsf{BS}}, \mathsf{pk}_s, \mathsf{tx}, \mathsf{sn})$.
2. If there is an x such that $\mathsf{H}(\mathsf{sn}, x)$ is already defined and $\mathsf{BS.Ver}(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sn}, x) = 1$, then run $\sigma_s \leftarrow \mathsf{SIG.Sig}(\mathsf{sk}_s, \mathsf{tx})$, and set $\mathsf{ct} := \mathsf{H}(\mathsf{sn}, x) \oplus \sigma_s$. Otherwise, sample $\mathsf{ct} \leftarrow\!\$\, \{0,1\}^\ell$.
3. Set $\mathsf{stmt} := (\mathsf{pk}_{\mathsf{BS}}, \mathsf{pk}_s, \mathsf{tx}, \mathsf{sn}, \mathsf{ct})$ and run $\pi \leftarrow \mathsf{PSim}(\mathsf{stmt})$.
4. Insert $(\mathsf{sn}, \mathsf{ct}, \sigma_s)$ into L.
5. Output (ct, π).

Note that algorithm Sim needs to simulate the proof π via zero-knowledge here, as it does not have the secret key $\mathsf{sk}_{\mathsf{BS}}$ and therefore it may not know the witness σ_{BS} to compute the proof honestly.

On a query (sn, x) for which $\mathsf{H}(\mathsf{sn}, x)$ is not yet defined, Sim_{RO} first checks if $\mathsf{BS.Ver}(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sn}, x) = 1$ and there is an entry of the form $(\mathsf{sn}, \mathsf{ct}, \sigma_s)$ in L. Note that there can be at most one such entry by the definition of the security game in which Sim and Sim_{RO} run. If these two conditions hold, it sets $\mathsf{H}(\mathsf{sn}, x) := \mathsf{ct} \oplus \sigma_s$. Otherwise, it samples $\mathsf{H}(\mathsf{sn}, x)$ at random.

It follows directly from the definition of zero-knowledge that $(\mathsf{Sim}, \mathsf{Sim}_{RO})$ is a simulator against malicious users for $\mathsf{RP}[\mathsf{SIG}, \mathsf{BS}, \mathsf{PS}]$, i.e. the security game with $b = 0$ is indistinguishable from the security game with $b = 1$.

Extractability. We provide algorithm Ext that shares state with algorithms Sim and Sim_{RO} as above, and extracts blind signatures σ_{BS} from signatures σ_s that are computed from a (simulated) promise message. Algorithm $\mathsf{Ext}(\mathsf{rpar}, \mathsf{sk}_s, \sigma_s)$ searches for a query $(\mathsf{sn}, \sigma_{\mathsf{BS}})$ for which $\mathsf{H}(\mathsf{sn}, \sigma_{\mathsf{BS}})$ is defined and it holds that $\mathsf{BS.Ver}(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sn}, \sigma_{\mathsf{BS}}) = 1$. If it finds such a query, it returns σ_{BS}. Otherwise, it returns $\perp$.

We have to show that the probability that the security game for extractability outputs 1 is negligible. Note that to do this, we only have to bound the probability of the bad event bad defined in the security game. Recall that this bad event occurs, if after getting message prom, the adversary $\mathcal{A}$ sends σ_s to oracle O such that $\mathsf{BS.Ver}(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sn}, \sigma_{\mathsf{BS}}) = 0$ and $\mathsf{SIG.Ver}(\mathsf{pk}_s, \mathsf{tx}, \sigma_s) = 1$, where $\sigma_{\mathsf{BS}} \leftarrow \mathsf{Ext}(\mathsf{rpar}, \mathsf{sk}_s, \sigma_s)$. Due to the definition of algorithm Ext this means that the hash value $\mathsf{H}(\mathsf{sn}, \sigma_{\mathsf{BS}})$ is not defined, where σ_{BS} is the unique signature satisfying $\mathsf{BS.Ver}(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sn}, \sigma_{\mathsf{BS}}) = 1$. The probability that this bad event occurs in the i-th interaction with oracle O can be bounded using a reduction from the $\mathsf{EUF\text{-}CMA}$ security of SIG.

We sketch the reduction. The reduction gets as input a public key pk_s^*. It simulates the security game honestly, except for the i-th interaction. In this interaction, it uses $\mathsf{pk}_s := \mathsf{pk}_s^*$ instead of sampling a fresh key pair $(\mathsf{pk}_s, \mathsf{sk}_s)$. Note that the corresponding secret key and a signature σ_s is never needed to compute prom or to answer random oracle queries, assuming that the bad event occurs. This is because sk_s is only used by algorithm Sim if $\mathsf{H}(\mathsf{sn}, \sigma_{\mathsf{BS}})$ is defined. Also, if the bad event occurs, the reduction can return (tx, σ_s), which is valid if the bad event occurs. Note that the reduction never uses its signing oracle. Therefore, the forgery (tx, σ_s) is fresh. □

C.2 Proofs for the Schnorr Cut-and-Choose Construction

Proof of Lemma 6.8 (Mal. Service - Schnorr). To prove the claim, we present an algorithm Ext. It takes as input parameters rpar, a promise message prom, and a list $\mathcal{Q}$ of random oracle queries and outputs a blind signature σ_{BS}. Algorithm $\mathsf{Ext}(\mathsf{rpar}, \mathsf{prom}, \mathcal{Q})$ is as follows:

1. Parse $\mathsf{rpar} = (\mathsf{pk}_{\mathsf{BS}}, \mathsf{pk}_s, \mathsf{tx}, \mathsf{sn})$ and $\mathsf{prom} = (\mathsf{prom}_1, \mathsf{prom}_2)$.
2. Parse $\mathsf{prom}_1 = ((\mathsf{ct}_j)_{j \in [2\lambda]}, (\mathsf{coeff}'_0, e), (\mathsf{coeff}_j, \mathsf{coeff}'_j)_{j \in [\lambda]})$.
3. Compute $b_0 \dots b_{\lambda-1} := \mathsf{H}_c(\mathsf{prom}_1)$ and for $j \in [\lambda]$ compute $\bar{k}_j := 2j - (1 - b_{j-1})$.
4. For each $j \in [\lambda]$ compute $\mathsf{pk}_{\mathsf{BS},\bar{k}_j} := \mathsf{pk}_{\mathsf{BS}} \cdot \prod_{i=1}^{\lambda} (\mathsf{coeff}_j)^{\bar{k}_j^i}$.
5. Find an index $j^* \in [\lambda]$ and an entry $((\mathsf{sn}, \sigma_{\bar{k}_{j^*}}), \hat{\mathsf{H}}_q(\mathsf{sn}, \sigma_{\bar{k}_{j^*}}))$ in $\mathcal{Q}$, such that $\mathsf{BS.Ver}(\mathsf{pk}_{\mathsf{BS},\bar{k}_{j^*}}, \mathsf{sn}, \sigma_{\bar{k}_{j^*}}) = 1$.
6. If such a $\sigma_{\bar{k}_{j^*}}$ is found for $j^* \in [\lambda]$, return
$$\mathsf{reconst}_{g_1,0}((\bar{k}_{j^*}, \sigma_{\bar{k}_{j^*}}), (k_j, \sigma_{k_j})_{j \in [\lambda]}).$$
Otherwise, return $\perp$.

It remains to prove that for this algorithm Ext, the probability that the security game outputs 1 is negligible. In the security game, we define the event win_1 which occurs if $\mathsf{VerPromise}(\mathsf{rpar}, \mathsf{prom}) = 1$ and Ext outputs $\perp$. We also define the event win_2 which occurs if $\mathsf{VerPromise}(\mathsf{rpar}, \mathsf{prom}) = 1$, algorithm Ext outputs a valid blind signature σ_{BS} for sn, but for $\sigma_s \leftarrow \mathsf{Redeem}(\mathsf{rpar}, \mathsf{prom}, \sigma_{\mathsf{BS}})$ we have $\mathsf{SIG.Ver}(\mathsf{pk}_s, \mathsf{tx}, \sigma_s) = 0$. Note that whenever algorithm Ext does not output $\perp$, it outputs a valid blind signature for sn. Therefore, the game outputs 1 if and only if win_1 or win_2 occurs.

First, we upper bound the probability of win_1. To this end, consider the following two events partitioning win_1:

- $\mathsf{win}_{1,1}$: win_1 occurs and there is some $\hat{j} \in [\lambda]$ such that the adversary never queried $\hat{\mathsf{H}}_q(\mathsf{sn}, \sigma_{k_{\hat{j}}})$ before querying $\mathsf{H}_c(\mathsf{prom}_1)$.
- $\mathsf{win}_{1,2}$: win_1 occurs and $\mathsf{win}_{1,1}$ does not occurs, i.e. win_1 occurs, and for all $j \in [\lambda]$, the adversary queried $\hat{\mathsf{H}}_q(\mathsf{sn}, \sigma_{k_j})$ before querying $\mathsf{H}_c(\mathsf{prom}_1)$.

Clearly, we can bound the probability of win_1 by bounding the probability of $\mathsf{win}_{1,1}$ and $\mathsf{win}_{1,2}$ separately. We start with event $\mathsf{win}_{1,1}$. We can assume that $\mathsf{VerPromise}(\mathsf{rpar}, \mathsf{prom}) = 1$ and therefore $g^{s_{k_j}} = \prod_{i=0}^{\lambda}(\mathsf{coeff}'_j)^{k_j^i}$ for all $j \in [\lambda]$. Note that when the adversary queries $\mathsf{H}_c(\mathsf{prom}_1)$, the values $s_{k_{\hat{j}}}$ and $\mathsf{pk}_{\mathsf{BS},k_j}$ are information theoretically fixed by the values $\mathsf{coeff}'_0, (\mathsf{coeff}'_j)_j$ and $\mathsf{pk}_{\mathsf{BS}}, (\mathsf{coeff}_j)_j$, respectively. Therefore, the query $\mathsf{H}_c(\mathsf{prom}_1)$ also fixes the value of $\Delta := \mathsf{ct}_{k_{\hat{j}}} \oplus s_{k_{\hat{j}}}$. If $\mathsf{VerPromise}(\mathsf{rpar}, \mathsf{prom}) = 1$, this value must be equal to $\hat{\mathsf{H}}_q(\mathsf{sn}, \sigma_{k_{\hat{j}}})$. The probability that after Δ is fixed, any of the polynomial many queries to $\hat{\mathsf{H}}_q$ evaluates to Δ is negligible. Thus, the probability of $\mathsf{win}_{1,1}$ is negligible. Next, we bound the probability of event $\mathsf{win}_{1,2}$. If this event occurs, we know that at the moment where the adversary queries $\mathsf{H}_c(\mathsf{prom}_1)$, it holds that for all $j \in [\lambda]$, $\hat{\mathsf{H}}_q(\mathsf{sn}, \sigma_{k_j})$ has been queried, and $\hat{\mathsf{H}}_q(\mathsf{sn}, \sigma_{\bar{k}_{\hat{j}}})$ has not been queried (due to the definition of algorithm Ext and win_1). Thus, the bits $b_0, \ldots, b_{\lambda-1}$ are fixed before $\mathsf{H}_c(\mathsf{prom}_1)$ is queried, and $\mathsf{H}_c(\mathsf{prom}_1) = b_0, \ldots, b_{\lambda-1}$. This happens with negligible probability $1/2^{\lambda}$.

Next, we bound the probability of event win_2. By definition of algorithm $\mathsf{VerPromise}$ we know that $\mathsf{H}_q(\mathsf{coeff}'_0 \cdot (\mathsf{pk}_s)^e, \mathsf{tx}) = e$. Thus, if win_2 occurs, we know that Redeem did not return (s, e) such that $g^s = \mathsf{coeff}'_0$. This can only happen if for all $j \in [\lambda]$, we have $s_{\bar{k}_j} \neq f'(\bar{k}_j)$, where f' is the polynomial that is defined by the values $\mathsf{coeff}'_0, (\mathsf{coeff}'_j)_j$. As σ_{BS} is output by Ext and satisfies $\mathsf{BS.Ver}(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sn}, \sigma_{\mathsf{BS}}) = 1$, we know that the values $\sigma_{\bar{k}_j}$ computed in Redeem are the unique values satisfying $\mathsf{BS.Ver}(\mathsf{pk}_{\mathsf{BS},\bar{k}_j}, \mathsf{sn}, \sigma_{\bar{k}_j}) = 1$. This means that both the values $s_{k_j} = \mathsf{ct}_{k_j} \oplus \hat{\mathsf{H}}_q(\mathsf{sn}, \sigma_{k_j})$ and $s_{\bar{k}_j} = \mathsf{ct}_{\bar{k}_j} \oplus \hat{\mathsf{H}}_q(\mathsf{sn}, \sigma_{\bar{k}_j})$ are information theoretically fixed at the first time $\mathsf{H}_c(\mathsf{prom}_1)$ is queried. At the same time, we have $s_{\bar{k}_j} \neq f'(\bar{k}_j)$ and $s_{k_j} = f'(k_j)$ for all $j \in [\lambda]$, uniquely defining the bits $b_0, \ldots b_{\lambda-1}$. Thus, the probability that $\mathsf{win}_{2,1}$ occurs is at most the probability that $\mathsf{H}_c(\mathsf{prom}_1) = b_0, \ldots b_{\lambda-1}$, which is negligible. □

Proof of Lemma 6.9 (Mal. User - Schnorr). To prove the claim, we need provide algorithms $\mathsf{Sim}, \mathsf{Sim}_{RO}$ and Ext that share state.

Simulatability. Before we provide algorithms $\mathsf{Sim}, \mathsf{Sim}_{RO}$, we give a sequence of hybrid games, starting from the simulatability game with bit $b = 0$ (i.e. computing prom via algorithm $\mathsf{Promise}$). The final game will be equivalent to the simulatability game with bit $b = 1$ for the simulators we define then.

Game $\mathbf{G}_0$: We start with game $\mathbf{G}_0$, which is the simulatability game with $b = 0$. To recall, in this game, a pair of blind signature keys $(\mathsf{pk}_{\mathsf{BS}} = g_2^{\mathsf{sk}_{\mathsf{BS}}}, \mathsf{sk}_{\mathsf{BS}})$ is sampled and given to the adversary. Then, the adversary gets access to an oracle O that on input sn aborts if sn has already been submitted. Otherwise, it samples Schnorr signing keys $(\mathsf{pk}_s = g^{\mathsf{sk}_s}, \mathsf{sk}_s)$ and gives pk_s to the adversary, receiving tx in return. It then defines $\mathsf{rpar} := (\mathsf{pk}_{\mathsf{BS}}, \mathsf{pk}_s, \mathsf{tx}, \mathsf{sn})$ and outputs $\mathsf{prom} \leftarrow \mathsf{Promise}(\mathsf{rpar}, \mathsf{sk}_{\mathsf{BS}}, \mathsf{sk}_s)$. For this scheme, prom has the form $\mathsf{prom} := (\mathsf{prom}_1, \mathsf{prom}_2 := (\sigma_{k_j}, s_{k_j})_{j\in[\lambda]})$ with $\mathsf{prom}_1 := ((\mathsf{ct}_j)_{j\in[2\lambda]}, (\mathsf{coeff}'_0, e), (\mathsf{coeff}_j, \mathsf{coeff}'_j)_{j\in[\lambda]})$. Additionally, the adversary gets access to random oracles $\hat{\mathsf{H}}_q, \mathsf{H}, \mathsf{H}_c, \mathsf{H}_q$ provided in the standard lazy manner.

Game $\mathbf{G}_1$: We add a change to the computation of prom. Namely, in the beginning of algorithm $\mathsf{Promise}$, the game samples random bits $b_0, \ldots, b_{\lambda_1} \leftarrow\!\!\$\, \{0, 1\}$. Then, it computes prom_1 as before. If $\mathsf{H}_c(\mathsf{prom}_1)$ is already defined, the game aborts. Otherwise, it sets $\mathsf{H}_c(\mathsf{prom}_1) := b_0, \ldots, b_{\lambda_1}$ and continues the computation of prom as before. Note that the probability of such an abort is negligible, due to the entropy of $\mathsf{coeff}'_0 = g^k \cdot \mathsf{pk}_s^{-e}$. Thus, $\mathbf{G}_0$ and $\mathbf{G}_1$ are indistinguishable. Observe the effect of this change: We can now define the values $k_j := 2j - b_{j-1}$ and $\bar{k}_j := \bar{k}_j := 2j - (1 - b_{j-1})$ before we compute prom_1.

Game $\mathbf{G}_2$: We change how the values $\mathsf{ct}_{\bar{k}_j}$ for $j \in [\lambda]$ are computed. Namely, note that they were defined as $\mathsf{ct}_{\bar{k}_j} := \hat{\mathsf{H}}_q(\mathsf{sn}, \sigma_{\bar{k}_j}) \oplus s_{\bar{k}_j}$ before, where $s_{\bar{k}_j} := f'(\bar{k}_j)$, and $\sigma_{\bar{k}_j}$ is the unique value satisfying $\mathsf{BS.Ver}(\mathsf{pk}_{\mathsf{BS},\bar{k}_j}, \mathsf{sn}, \sigma_{\bar{k}_j}) = 1$. From now on, the game first checks if $\hat{\mathsf{H}}_q(\mathsf{sn}, \sigma_{\bar{k}_j})$ is already defined. Note that the game can do that without knowing $\mathsf{sk}_{\mathsf{BS}}$ or $\sigma_{\bar{k}_j}$, just by iterating over all queries and running $\mathsf{BS.Ver}$. If it is already defined, the game sets $\mathsf{ct}_{\bar{k}_j} := \hat{\mathsf{H}}_q(\mathsf{sn}, \sigma_{\bar{k}_j}) \oplus s_{\bar{k}_j}$. Otherwise, it samples a random $\mathsf{ct}_{\bar{k}_j} \leftarrow\!\!\$\, \mathbb{Z}_p$, and for any subsequent random oracle query $\hat{\mathsf{H}}_q(\mathsf{sn}, \sigma_{\bar{k}_j})$ with $\mathsf{BS.Ver}(\mathsf{pk}_{\mathsf{BS},\bar{k}_j}, \mathsf{sn}, \sigma_{\bar{k}_j}) = 1$, it sets $\hat{\mathsf{H}}_q(\mathsf{sn}, \sigma_{\bar{k}_j}) := \mathsf{ct}_{\bar{k}_j} \oplus s_{\bar{k}_j}$. It is easy to see that this does not change the view of the adversary. Note that from now on, the values $\mathsf{sk}_{\mathsf{BS}}, (\mathsf{sk}_{\bar{k}_j})_j$ are no longer needed, except for the computation of coeff_j.

Game $\mathbf{G}_3$: We change the computation of prom again. The effect of this change will be that the key $\mathsf{sk}_{\mathsf{BS}}$ is no longer needed. Namely, we change how the values coeff_j are computed. They are now computed as

$$((\mathsf{sk}_{k_j}, \mathsf{coeff}_j)_{j\in[\lambda]}) \leftarrow \mathsf{polyGen}_{g_2,p}(\lambda, \mathsf{pk}_{\mathsf{BS}}, (k_j)_{j\in[\lambda]})$$

It is clear that game $\mathbf{G}_2$ and $\mathbf{G}_3$ are indistinguishable.

It is easy to see that in $\mathbf{G}_3$, the oracle O can be run without using $\mathsf{sk_{BS}}$. In other words, there are simulators $\mathsf{Sim}, \mathsf{Sim}_{RO}$ that share state, such that Sim_{RO} controls the random oracles as in $\mathbf{G}_3$, and $\mathsf{Sim}(\mathsf{rpar}, \mathsf{sk}_s)$ computes the values prom in oracle O as in $\mathbf{G}_3$. This shows simulatability.

Extractability. Next, we show extractability. To this end, we provide algorithm Ext that shares state with algorithms Sim and Sim_{RO} as above, and extracts blind signatures σ_{BS} from signatures σ_s that are computed from a (simulated) promise message. Algorithm $\mathsf{Ext}(\mathsf{rpar}, \mathsf{sk}_s, \sigma_s)$ for $\mathsf{rpar} = (\mathsf{pk_{BS}}, \mathsf{pk}_s, \mathsf{tx}, \mathsf{sn})$ works as follows:

1. Let $\mathsf{sn}, \mathsf{prom}_1, \mathsf{prom}_2, b_0 \dots b_{\lambda-1}$ be as in the execution of Sim that took place in the same oracle call.
2. For $j \in [\lambda]$ compute $\bar{k}_j := 2j - (1 - b_{j-1})$.
3. For each $j \in [\lambda]$ compute $\mathsf{pk}_{\mathsf{BS},\bar{k}_j} := \mathsf{pk_{BS}} \cdot \prod_{i=1}^{\lambda} (\mathsf{coeff}_j)^{\bar{k}_j^i}$
4. Find an index $j^* \in [\lambda]$ and an entry $(\mathsf{sn}, \sigma_{\bar{k}_{j^*}})$ in the list of queries to random oracle $\hat{\mathsf{H}}_q$ such that $\mathsf{BS.Ver}(\mathsf{pk}_{\mathsf{BS},\bar{k}_{j^*}}, \mathsf{sn}, \sigma_{\mathsf{BS},\bar{k}_{j^*}}) = 1$.
5. If such a $\sigma_{\mathsf{BS},\bar{k}_{j^*}}$ is found for some $j^* \in [\lambda]$, return

$$\mathsf{reconst}_{g_1,0}((\bar{k}_{j^*}, \sigma_{\bar{k}_{j^*}}), (k_j, \sigma_{k_j})_{j \in [\lambda]}).$$

   Otherwise, return $\perp$.

We have to show that the probability that the security game for extractability outputs 1 is negligible. Note that this game is as $\mathbf{G}_3$, but now after outputting prom, oracle O gets σ_s in return. The game outputs 1 if in any of these interactions, the event bad occurs, i.e. it holds that $\mathsf{BS.Ver}(\mathsf{pk_{BS}}, \mathsf{sn}, \sigma_{\mathsf{BS}}) = 0$ and $\mathsf{SIG.Ver}(\mathsf{pk}_s, \mathsf{tx}, \sigma_s) = 1$, where $\sigma_{\mathsf{BS}} \leftarrow \mathsf{Ext}(\mathsf{rpar}, \mathsf{sk}_s, \sigma_s)$. We distinguish two cases. In the first case, the adversary reuses the exact signature (s, e) that the game computes during the generation of prom. In this case, the adversary implicitly breaks the DLOG assumption by extracting s from $\mathsf{coeff}'_0 = g^s$. In the second case, the adversary comes up with a different signature (s, e), thereby breaking strong unforgeability of Schnorr signatures.

More precisely, we partition the bad event bad into the following two sub-events:

- bad_1: bad occurs and σ_s is sent to O by $\mathcal{A}$, initiated with sn and there exists an entry such that $\sigma_s = (s, e)$.
- bad_2: bad occurs and the returned signature σ_s is fresh, i.e. $\sigma_s \neq (s, e)$.

We first bound the the probability that event bad_2 occurs in the ith interaction with oracle O. This is done using a reduction from the $\mathsf{sEUF\text{-}CMA}$ security of SIG. We sketch the reduction. The reduction gets as input a public key pk_s^* and access to a signing oracle. It simulates the security game honestly, except for the ith interaction. In this interaction, it uses $\mathsf{pk}_s := \mathsf{pk}_s^*$ instead of sampling a fresh key pair $(\mathsf{pk}_s, \mathsf{sk}_s)$. It also gets the Schnorr signature (s, e) using the signing oracle. Finally, if event bad_2 occurs, the reduction can return (tx, σ_s), which is a valid forgery. Note that in such a case we have $\sigma_s \neq (s, e)$. Therefore, the reduction breaks $\mathsf{sEUF\text{-}CMA}$ security of SIG.

Next, we want to bound the probability of event bad_1. To do that, we first need to eliminate the dependency on s. This is done using two more hybrids.

Game $\mathbf{G}_4$: This is as the extractability game, but assuming the are at most q_{O} queries to the oracle O, the game picks an index $i \leftarrow_\$ [q_{\mathrm{O}}]$ and aborts in case the event bad_1 does not occur in the ith query to O. As q_{O} is polynomial and the view of the adversary is independent of i, it is sufficient to bound the probability of bad_1 in game $\mathbf{G}_4$.

Game $\mathbf{G}_5$: This is as $\mathbf{G}_4$, but we change how prom is computed in the ith query to O. Namely, the game first samples $\mathsf{coeff}'_0 \leftarrow_\$ \mathbb{G}$, then samples $e \leftarrow_\$ \mathbb{Z}_q^*$, and aborts if $\mathsf{H}_q(\mathsf{coeff}'_0 \cdot (\mathsf{pk}_s)^e, \mathsf{tx})$ is already defined. Otherwise, it programs $\mathsf{H}_q(\mathsf{coeff}'_0 \cdot (\mathsf{pk}_s)^e, \mathsf{tx}) := e$ and continues the computation of prom as before. If the game ever has to access $s_{\bar{k}_j}$ for some $j \in [\lambda]$ (recall that this happens if $\hat{\mathsf{H}}_q(\mathsf{sn}, \sigma_{\bar{k}_j})$ with $\mathsf{BS.Ver}(\mathsf{pk}_{\mathsf{BS},\bar{k}_j}, \mathsf{sn}, \sigma_{\bar{k}_j}) = 1$ is ever queried), then it aborts. Observe that the probability of the first abort is negligible due to the entropy of coeff'_0, and the second abort only occurs if bad does not occur in the ith interaction.

We show that the probability of event bad_1 occurring in game $\mathbf{G}_5$ is negligible, using a reduction to the DLOG assumption. We sketch the reduction. It gets as input the instance $Y = g^\alpha$. It simulates game $\mathbf{G}_5$ honestly, except for the ith interaction of $\mathcal{A}$ with the oracle O. In this interaction, it sets $\mathsf{coeff}'_0 := Y$ and continues the simulation as in game $\mathbf{G}_5$. Note that the polynomial f' and the discrete logarithm of coeff'_0 is never needed for that, due to the previous change. In the end, the adversary returns a signature σ_s for which we know that $\mathsf{SIG.Ver}(\mathsf{pk}_s, \mathsf{tx}, \sigma_s) = 1$ and because of event bad_1 we know that $\sigma_s = (\alpha, e)$. The reduction can return α as the solution. □

C.3 Proofs for the BLS Cut-and-Choose Construction

Lemma C.1 *Let $\mathbb{G}_1, \mathbb{G}_2$ be cyclic groups of prime order $p > 2^\lambda$ with respective generators $g_1 \in \mathbb{G}_1, g_2 \in \mathbb{G}_2$. For any two elements $h, \bar{h} \in \mathbb{G}_1$ consider the function*

$$F_{h,\bar{h}} : \mathbb{Z}_p^2 \to \mathbb{G}_1^2 \times \mathbb{G}_2, \quad (s_0, \mathsf{sk}_s) \mapsto (h^{s_0} \cdot \bar{h}^{\mathsf{sk}_s}, g_1^{s_0}, g_2^{\mathsf{sk}_s}).$$

For any algorithm $\mathcal{A}$ consider the following game:

1. *Sample $h, \bar{h} \leftarrow_{\$} \mathbb{G}_1$ and run $\mathcal{A}$ on input $h, \bar{h}$.*
2. *Obtain $(\mathsf{ct}_0, \mathsf{coeff}'_0, \mathsf{pk}_s) \in \mathbb{G}_1^2 \times \mathbb{G}_2$ and $(T_1, T_2, T_3) \in \mathbb{G}_1^2 \times \mathbb{G}_2$ from $\mathcal{A}$.*
3. *If $(\mathsf{ct}_0, \mathsf{coeff}'_0, \mathsf{pk}_s) \in F_{h,\bar{h}}(\mathbb{Z}_p^2)$, return 0.*
4. *Sample $e \leftarrow_{\$} \mathbb{Z}_p$ and give e to $\mathcal{A}$.*
5. *Obtain $(\pi_0, \pi_1) \in \mathbb{Z}_p^2$ from $\mathcal{A}$.*
6. *Return 1 if $T_0 = h^{\pi_0} \cdot \bar{h}^{\pi_1} \cdot \mathsf{ct}_0^{-e}$, $T_1 = g_1^{\pi_0} \cdot (\mathsf{coeff}'_0)^{-e}$, and $T_2 = g_2^{\pi_1} \cdot (\mathsf{pk}_s)^{-e}$. Otherwise, return 0.*

Then, for any algorithm $\mathcal{A}$, the probability that the above game outputs 1 is negligible.

Proof. Note that if the game outputs 1, we know that $\mathcal{A}$ returned a tuple $(\mathsf{ct}_0, \mathsf{coeff}'_0, \mathsf{pk}_s)$ which is not in the image of $F_{h,\bar{h}}$. We consider two cases. In the first case, assume that for each tuple $(T_1, T_2, T_3) \in\in \mathbb{G}_1^2 \times \mathbb{G}_2$, there is at most one $e \in \mathbb{Z}_p$ such that there exists a response $(\pi_0, \pi_1) \in \mathbb{Z}_p^2$ that lets the game output 1. In this case, it is clear that the probability of $\mathcal{A}$ is at most $1/|\mathbb{Z}_p|$, which is negligible.

In the second case, assume that there is a tuple $(T_1, T_2, T_3) \in\in \mathbb{G}_1^2 \times \mathbb{G}_2$, such that there are at least two distinct $e \neq e'$ in $\mathbb{Z}_p$, such that there exist responses $(\pi_0, \pi_1), (\pi'_0, \pi'_1) \in \mathbb{Z}_p^2$ that let the game output 1. We show that this case can not occur by deriving that in this case, $(\mathsf{ct}_0, \mathsf{coeff}'_0, \mathsf{pk}_s)$ is in the image of $F_{h,\bar{h}}$. Namely, from the existence of such responses for the same tuple (T_1, T_2, T_3), we obtain

$$\begin{aligned} h^{\pi_0} \cdot \bar{h}^{\pi_1} \cdot \mathsf{ct}_0^{-e} = \quad & T_0 & = h^{\pi'_0} \cdot \bar{h}^{\pi'_1} \cdot \mathsf{ct}_0^{-e'} \\ g_1^{\pi_0} \cdot (\mathsf{coeff}'_0)^{-e} = \quad & T_1 & = g_1^{\pi'_0} \cdot (\mathsf{coeff}'_0)^{-e'} \\ g_2^{\pi_1} \cdot (\mathsf{pk}_s)^{-e} = \quad & T_2 & = g_2^{\pi'_1} \cdot (\mathsf{pk}_s)^{-e'}. \end{aligned}$$

Rearranging terms, we get that

$$\left(\frac{\pi_0 - \pi'_0}{e - e'}, \frac{\pi_1 - \pi'_1}{e - e'}\right)$$

is a pre-image of $(\mathsf{ct}_0, \mathsf{coeff}'_0, \mathsf{pk}_s)$ under $F_{h,\bar{h}}$. □

Proof of Lemma 6.6 (Mal. Service - BLS). The proof is almost identical to the proof of Lemma 6.8, and we take it partially verbatim. To prove the claim, we present an algorithm Ext that takes as input parameters rpar, a promise message prom, and a list $\mathcal{Q}$ of random oracle queries and outputs a blind signature σ_{BS}. The algorithm is as follows:

1. Parse $\mathsf{rpar} = (\mathsf{pk}_{\mathsf{BS}}, \mathsf{pk}_s, \mathsf{tx}, \mathsf{sn})$ and $\mathsf{prom} = (\mathsf{prom}_1, \mathsf{prom}_2)$.
2. Let$\mathsf{prom}_1 := (\mathsf{ct}_0, (\mathsf{ct}_j)_{j\in[2\lambda]}, (\pi_0, \pi_1, e), \mathsf{coeff}'_0, (\mathsf{coeff}_j, \mathsf{coeff}'_j)_{j\in[\lambda]})$.

3. Compute $b_0 \dots b_{\lambda-1} := \mathsf{H}_c(\mathsf{prom}_1)$ and for all $j \in [\lambda]$ compute $\bar{k}_j := 2j - (1 - b_{j-1})$.

4. For each $j \in [\lambda]$ compute $\mathsf{pk}_{\mathsf{BS},\bar{k}_j} := \mathsf{pk}_{\mathsf{BS}} \cdot \prod_{i=1}^{\lambda} (\mathsf{coeff}_j)^{\bar{k}_j^i}$.

5. Find an index $j^* \in [\lambda]$ and an entry $((\mathsf{sn}, \sigma_{\bar{k}_{j^*}}), \hat{\mathsf{H}}(\mathsf{sn}, \sigma_{\bar{k}_{j^*}}))$ in $\mathcal{Q}$, such that $\mathsf{BS.Ver}(\mathsf{pk}_{\mathsf{BS},\bar{k}_{j^*}}, \mathsf{sn}, \sigma_{\bar{k}_{j^*}}) = 1$.

6. If such a $\sigma_{\bar{k}_{j^*}}$ is found for some $j^* \in [\lambda]$, return

$$\mathsf{reconst}_{g_1,0}((\bar{k}_{j^*}, \sigma_{\bar{k}_{j^*}}), (k_j, \sigma_{k_j})_{j\in[\lambda]}).$$

Otherwise, return $\perp$.

It remains to prove that for this algorithm Ext, the probability that the security game outputs 1 is negligible. In the security game, we define the event win_1 which occurs if $\mathsf{VerPromise}(\mathsf{rpar}, \mathsf{prom}) = 1$ and Ext outputs $\perp$. We also define the event win_2 which occurs if $\mathsf{VerPromise}(\mathsf{rpar}, \mathsf{prom}) = 1$, algorithm Ext outputs a valid blind signature σ_{BS} for sn, but for $\sigma_s \leftarrow \mathsf{Redeem}(\mathsf{rpar}, \mathsf{prom}, \sigma_{\mathsf{BS}})$ we have $\mathsf{SIG.Ver}(\mathsf{pk}_s, \mathsf{tx}, \sigma_s) = 0$. Note that whenever algorithm Ext does not output $\perp$, it outputs a valid blind signature for sn. Therefore, the game outputs 1 if and only if win_1 or win_2 occurs.

First, we upper bound the probability of win_1. To this end, consider the following two events partitioning win_1:

- $\mathsf{win}_{1,1}$: win_1 occurs and there is some $\hat{j} \in [\lambda]$ such that the adversary never queried $\hat{\mathsf{H}}(\mathsf{sn}, \sigma_{k_{\hat{j}}})$ before querying $\mathsf{H}_c(\mathsf{prom}_1)$.
- $\mathsf{win}_{1,2}$: win_1 occurs and $\mathsf{win}_{1,1}$ does not occurs, i.e. win_1 occurs, and for all $j \in [\lambda]$, the adversary queried $\hat{\mathsf{H}}(\mathsf{sn}, \sigma_{k_j})$ before querying $\mathsf{H}_c(\mathsf{prom}_1)$.

Clearly, we can bound the probability of win_1 by bounding the probability of $\mathsf{win}_{1,1}$ and $\mathsf{win}_{1,2}$ separately. We start with event $\mathsf{win}_{1,1}$. We can assume that $\mathsf{VerPromise}(\mathsf{rpar}, \mathsf{prom}) = 1$ and therefore $g_1^{s_{k_j}} = \prod_{i=0}^{\lambda} (\mathsf{coeff}'_j)^{k_j^i}$ for all $j \in [\lambda]$. Note that when the adversary queries $\mathsf{H}_c(\mathsf{prom}_1)$, the values $s_{k_{\hat{j}}}$ and $\mathsf{pk}_{\mathsf{BS},k_j}$ are information theoretically fixed by the values $\mathsf{coeff}'_0, (\mathsf{coeff}'_j)_j$ and $\mathsf{pk}_{\mathsf{BS}}, (\mathsf{coeff}_j)_j$, respectively. Therefore, the query $\mathsf{H}_c(\mathsf{prom}_1)$ also fixes the value of $\Delta := \mathsf{ct}_{k_{\hat{j}}} \cdot h^{-s_{k_{\hat{j}}}}$. If $\mathsf{VerPromise}(\mathsf{rpar}, \mathsf{prom}) = 1$, this value must be equal to $\hat{\mathsf{H}}(\mathsf{sn}, \sigma_{k_{\hat{j}}})$. The probability that after Δ is fixed, any of the polynomial many queries to $\hat{\mathsf{H}}$ evaluates to Δ is negligible. Thus, the probability of $\mathsf{win}_{1,1}$ is negligible. Next, we bound the probability of event $\mathsf{win}_{1,2}$. If this event occurs, we know that at the moment where the adversary queries $\mathsf{H}_c(\mathsf{prom}_1)$, it holds that for all $j \in [\lambda]$, $\hat{\mathsf{H}}(\mathsf{sn}, \sigma_{k_j})$ has been queried, and $\hat{\mathsf{H}}(\mathsf{sn}, \sigma_{\bar{k}_{\hat{j}}})$ has not been queried (due to the definition of algorithm Ext and win_1). Thus, the bits $b_0, \dots, b_{\lambda-1}$ are fixed before $\mathsf{H}_c(\mathsf{prom}_1)$ is queried, and $\mathsf{H}_c(\mathsf{prom}_1) = b_0, \dots, b_{\lambda-1}$. This happens with negligible probability $1/2^{\lambda}$.

Next, we bound the probability of event win_2. Consider the values $h_{k_j}, h_{\bar{k}_j}$ for $j \in [\lambda]$ as in the definition of algorithm Redeem. We partition win_2 into the following sub-events:

- $\mathsf{win}_{2,1}$: win_2 occurs and $\mathsf{ct}_0 = h^{f'(0)} \cdot \mathsf{H}(\mathsf{tx})^{\mathsf{sk}_s}$.
- $\mathsf{win}_{2,2}$: win_2 occurs and $\mathsf{ct}_0 \neq h^{f'(0)} \cdot \mathsf{H}(\mathsf{tx})^{\mathsf{sk}_s}$.

First, assume that $\mathsf{win}_{2,1}$ occurs. In this case, we know that $h_{\bar{k}_j} \neq h^{f'(\bar{k}_j)}$ for all $j \in [\lambda]$, where f' is the polynomial that is defined by the values coeff'_j that are contained in prom. We know that σ_{BS} is a valid blind signature for sn, and therefore the values $\sigma_{\bar{k}_j}$ computed in Redeem are the unique valid blind signatures for sn with respect to $\mathsf{pk}_{\mathsf{BS},k_j}$. Note that this means that both the values $h_{k_j} = \mathsf{ct}_{k_j}/\hat{\mathsf{H}}(\mathsf{sn}, \sigma_{k_j})$ and $h_{\bar{k}_j} = \mathsf{ct}_{\bar{k}_j}/\hat{\mathsf{H}}(\mathsf{sn}, \sigma_{\bar{k}_j})$ are information theoretically fixed at the first time $\mathsf{H}_c(\mathsf{prom}_1)$ is queried. At the same time, we have $h_{k_j} = h^{f'(k_j)}$ and $h_{\bar{k}_j} \neq h^{f'(\bar{k}_j)}$ for all $j \in [\lambda]$, uniquely defining the bits $b_0, \dots b_{\lambda-1}$. Thus, the probability that $\mathsf{win}_{2,1}$ occurs is at most the probability that $\mathsf{H}_c(\mathsf{prom}_1) = b_0, \dots b_{\lambda-1}$, which is negligible. Finally, we can bound the probability of $\mathsf{win}_{2,2}$ by Lemma C.1. □

Proof of Lemma 6.7 (Mal. User - BLS). To prove the claim, we need provide algorithms $\mathsf{Sim}, \mathsf{Sim}_{RO}$ and Ext that share state.

Simulatability. Before we provide algorithms $\mathsf{Sim}, \mathsf{Sim}_{RO}$, we give a sequence of hybrid games, starting from the simulatability game with bit $b = 0$ (i.e. computing prom via algorithm $\mathsf{Promise}$). The final game will be equivalent to the simulatability game with bit $b = 1$ for the simulators we define then.

Game $\mathbf{G}_0$: We start with game $\mathbf{G}_0$, which is the simulatability game with $b = 0$. To recall, in this game, a pair of blind signature keys $(\mathsf{pk}_{\mathsf{BS}} = g_2^{\mathsf{sk}_{\mathsf{BS}}}, \mathsf{sk}_{\mathsf{BS}})$ is sampled and given to the adversary. Then, the adversary gets access to an oracle O that on input sn aborts if sn has already been submitted. Otherwise, it samples signing keys $(\mathsf{pk}_s = g_2^{\mathsf{sk}_s}, \mathsf{sk}_s)$ and gives pk_s to the adversary, receiving tx in return. It then defines $\mathsf{rpar} := (\mathsf{pk}_{\mathsf{BS}}, \mathsf{pk}_s, \mathsf{tx}, \mathsf{sn})$ and outputs $\mathsf{prom} \leftarrow \mathsf{Promise}(\mathsf{rpar}, \mathsf{sk}_{\mathsf{BS}}, \mathsf{sk}_s)$. For this scheme, prom has the form $\mathsf{prom} := (\mathsf{prom}_1, \mathsf{prom}_2 := (\sigma_{k_j}, s_{k_j})_{j\in[\lambda]})$ with $\mathsf{prom}_1 := (\mathsf{ct}_0, (\mathsf{ct}_j)_{j\in[2\lambda]}, (\pi_0, \pi_1, e), \mathsf{coeff}'_0,$ $(\mathsf{coeff}_j, \mathsf{coeff}'_j)_{j\in[\lambda]})$. Additionally, the adversary gets access to random oracles $\hat{\mathsf{H}}, \mathsf{H}, \mathsf{H}_c, \mathsf{H}_p$ provided in the standard lazy manner.

Game $\mathbf{G}_1$: In this game, we change how the proofs π_0, π_1, e are computed. Namely, they are from now on simulated by sampling $\pi_0, \pi_1, e \leftarrow\!\!\$\, \mathbb{Z}_p^*$, setting $T_0 := h^{\pi_0} \cdot \mathsf{H}(\mathsf{tx})^{\pi_1} \cdot \mathsf{ct}_0^{-e}$, $T_1 := g_1^{\pi_0} \cdot (\mathsf{coeff}'_0)^{-e}$, and $T_2 := g_2^{\pi_1} \cdot (\mathsf{pk}_s)^{-e}$, and then aborting if $\mathsf{H}_p(T_0, T_1, T_2, h, \mathsf{H}(\mathsf{tx}), \mathsf{ct}_0, \mathsf{coeff}'_0, \mathsf{pk}_s)$ is already defined, and setting $\mathsf{H}_p(T_0, T_1, T_2, h, \mathsf{H}(\mathsf{tx}), \mathsf{ct}_0, \mathsf{coeff}'_0, \mathsf{pk}_s) := e$ otherwise. Due to the entropy of T_1, the probability of a potential abort is negligible. This implies that $\mathbf{G}_0$ and $\mathbf{G}_1$ are indistinguishable.

Game $\mathbf{G}_2$: We change how queries of the form $\hat{\mathsf{H}}(\mathsf{sn})$ are answered. Namely, from now on, whenever the hash value is not yet defined, the game first samples a random $h_{\mathsf{sn}} \leftarrow\!\!\$\, \mathbb{Z}_p$, and then sets $\hat{\mathsf{H}}(\mathsf{sn}) := g_1^{h_{\mathsf{sn}}}$. Clearly, this does not change the view of the adversary.

Game $\mathbf{G}_3$: We change how the component ct_0 of prom is computed. Namely, note that ct_0 has been computed via

$$\mathsf{ct}_0 = h^{s_0} \cdot \sigma_s = \hat{\mathsf{H}}(\mathsf{sn})^{s_0} \cdot \sigma_s = g^{h_{\mathsf{sn}} s_0} \cdot \sigma_s = \mathsf{coeff}_0'^{h_{\mathsf{sn}}} \cdot \sigma_s.$$

before. From now on, we compute ct_0 directly as $\mathsf{ct}_0 := \mathsf{coeff}_0'^{h_{\mathsf{sn}}} \cdot \sigma_s$. Clearly, this is only a conceptual change.

Game $\mathbf{G}_4$: We add another change to the computation of prom. Namely, we now sample bits $b_0, \ldots, b_{\lambda-1}$ uniformly from $\{0, 1\}$ in the beginning of algorithm $\mathsf{Promise}$. Then, we compute prom_1 as before and abort if $\mathsf{H}_c(\mathsf{prom}_1)$ is already defined. Otherwise, we set $\mathsf{H}_c(\mathsf{prom}_1) := b_0, \ldots, b_{\lambda-1}$ and continue. Note that the probability of such an abort is negligible, due to the entropy of π_0. Thus, $\mathbf{G}_3$ and $\mathbf{G}_4$ are indistinguishable. Observe the effect of this change: We can now define the values $k_j := 2j - b_{j-1}$ and $\bar{k}_j := \bar{k}_j := 2j - (1 - b_{j-1})$ before we compute prom_1.

Game $\mathbf{G}_5$: We change how the values $\mathsf{ct}_{\bar{k}_j}$ for $j \in [\lambda]$ are computed. Namely, note that they were defined as $\mathsf{ct}_{\bar{k}_j} := \hat{\mathsf{H}}(\mathsf{sn}, \sigma_{\bar{k}_j}) \cdot h^{s_{\bar{k}_j}}$ before, where $s_{\bar{k}_j} := f'(\bar{k}_j)$, and $\sigma_{\bar{k}_j}$ is the unique value satisfying $\mathsf{BS.Ver}(\mathsf{pk}_{\mathsf{BS},\bar{k}_j}, \mathsf{sn}, \sigma_{\bar{k}_j}) = 1$. From now on, the game first checks if $\hat{\mathsf{H}}(\mathsf{sn}, \sigma_{\bar{k}_j})$ is already defined. Note that the game can do that without knowing $\mathsf{sk}_{\mathsf{BS}}$ or $\sigma_{\bar{k}_j}$, just by iterating over all queries and running $\mathsf{BS.Ver}$. If it is already defined, the game sets $\mathsf{ct}_{\bar{k}_j} := \hat{\mathsf{H}}(\mathsf{sn}, \sigma_{\bar{k}_j}) \cdot \mathsf{coeff}'^{h_{\mathsf{sn}}}_{\bar{k}_j}$. Otherwise, it samples a random $\mathsf{ct}_{\bar{k}_j} \leftarrow\!\!\$\, \mathbb{G}_1$, and for any subsequent random oracle query $\hat{\mathsf{H}}(\mathsf{sn}, \sigma_{\bar{k}_j})$ with $\mathsf{BS.Ver}(\mathsf{pk}_{\mathsf{BS},\bar{k}_j}, \mathsf{sn}, \sigma_{\bar{k}_j}) = 1$, it sets $\hat{\mathsf{H}}(\mathsf{sn}, \sigma_{\bar{k}_j}) := \mathsf{coeff}'^{h_{\mathsf{sn}}}_{\bar{k}_j} / \mathsf{ct}_{\bar{k}_j}$. It is easy to see that this does not change the view of the adversary. Note that from now on, the values $\mathsf{sk}_{\mathsf{BS}}, (\mathsf{sk}_{\bar{k}_j}, s_{\bar{k}_j})_j$ are no longer needed, except for the computation of $\mathsf{coeff}_j, \mathsf{coeff}'_j$.

Game $\mathbf{G}_6$: In this game, we eliminate the last dependency on value $\mathsf{sk}_{\mathsf{BS}}$, by computing the values $\mathsf{coeff}_j, \mathsf{coeff}'_j$ via

$$((\mathsf{sk}_{k_j}, \mathsf{coeff}_j)_{j\in[\lambda]}) \leftarrow \mathsf{polyGen}_{g_2,p}(\lambda, \mathsf{pk}_{\mathsf{BS}}, (k_j)_{j\in[\lambda]}),$$
$$((s_{k_j}, \mathsf{coeff}'_j)_{j\in[\lambda]}) \leftarrow \mathsf{polyGen}_{g_1,p}(\lambda, \mathsf{coeff}'_0, (k_j)_{j\in[\lambda]}).$$

Clearly, this does not change the view of the adversary.

It is easy to see that in $\mathbf{G}_6$, the oracle O can be run without using $\mathsf{sk}_{\mathsf{BS}}$. In other words, there are simulators $\mathsf{Sim}, \mathsf{Sim}_{RO}$ that share state, such that Sim_{RO} controls the random oracles as in $\mathbf{G}_6$, and $\mathsf{Sim}(\mathsf{rpar}, \mathsf{sk}_s)$ computes the values prom in oracle O as in $\mathbf{G}_6$. This shows simulatability.

Extractability. For extractability, consider the following algorithm Ext that shares state with algorithms Sim and Sim_{RO} as above, and extracts blind signatures σ_{BS} from signatures σ_s that are computed from a (simulated) promise message prom. Algorithm $\mathsf{Ext}(\mathsf{rpar}, \mathsf{sk}_s, \sigma_s)$ for $\mathsf{rpar} = (\mathsf{pk}_{\mathsf{BS}}, \mathsf{pk}_s, \mathsf{tx}, \mathsf{sn})$ is defined as follows:

1. Let $\mathsf{sn}, \mathsf{prom}_1, \mathsf{prom}_2, b_0 \dots b_{\lambda-1}$ be as in the execution of Sim that took place in the same oracle call.
2. For $j \in [\lambda]$ compute $\bar{k}_j := 2j - (1 - b_{j-1})$.
3. For each $j \in [\lambda]$ compute $\mathsf{pk}_{\mathsf{BS},\bar{k}_j} := \mathsf{pk}_{\mathsf{BS}} \cdot \prod_{i=1}^{\lambda} (\mathsf{coeff}_j)^{\bar{k}_j^i}$
4. Find an index $j^* \in [\lambda]$ and an entry $(\mathsf{sn}, \sigma_{\bar{k}_{j^*}})$ in the list of queries to random oracle $\hat{\mathsf{H}}$ such that $\mathsf{BS.Ver}(\mathsf{pk}_{\mathsf{BS},\bar{k}_{j^*}}, \mathsf{sn}, \sigma_{\mathsf{BS},\bar{k}_{j^*}}) = 1$.
5. If such a $\sigma_{\mathsf{BS},\bar{k}_{j^*}}$ is found for some $j^* \in [\lambda]$, return

$$\mathsf{reconst}_{g_1,0}((\bar{k}_{j^*}, \sigma_{\bar{k}_{j^*}}), (k_j, \sigma_{k_j})_{j\in[\lambda]}).$$

   Otherwise, return $\perp$.

We have to show that the probability that the security game for extractability outputs 1 is negligible. To show this, we continue our sequence of hybrids. The overall idea is to reduce to $\mathsf{EUF\text{-}CMA}$ security of SIG. To this end, our sequence of hybrids eliminates the dependency on sk_s.

Game $\mathbf{G}_7$: Game $\mathbf{G}_7$ is the extractability security game with simulators Sim and Sim_{RO} and algorithm Ext. Note that this means that $\mathbf{G}_7$ is as $\mathbf{G}_6$, but now after outputting prom, oracle O gets σ_s in return. The game outputs 1 if in any of these interactions, the event bad occurs, i.e. it holds that $\mathsf{BS.Ver}(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sn}, \sigma_{\mathsf{BS}}) = 0$ and $\mathsf{SIG.Ver}(\mathsf{pk}_s, \mathsf{tx}, \sigma_s) = 1$, where $\sigma_{\mathsf{BS}} \leftarrow \mathsf{Ext}(\mathsf{rpar}, \mathsf{sk}_s, \sigma_s)$.

Game $\mathbf{G}_8$: Assuming the are at most q_{O} queries to the oracle O, the game picks an index $i \leftarrow\!\!\$\, [q_{\mathrm{O}}]$ and aborts in case the event bad does not occur in the ith query to O. As q_{O} is polynomial and the view of the adversary is independent of i, it is sufficient to bound the probability of bad in game $\mathbf{G}_8$.

Game $\mathbf{G}_9$: Assuming the are at most $q_{\hat{\mathsf{H}}}$ queries to the oracle $\hat{\mathsf{H}}$, the game picks an index $i_h \leftarrow\!\!\$\, [q_{\hat{\mathsf{H}}}]$ and aborts in case the i_hth query is for a sn' such that the ith query to O used a different $\mathsf{sn} \neq \mathsf{sn}'$. As $q_{\hat{\mathsf{H}}}$ is polynomial and the view of the adversary is independent of i_h, it is sufficient to bound the probability of bad in game $\mathbf{G}_9$.

Game $\mathbf{G}_{10}$: In this game, we change how the promise message prom for the ith query with sn of the adversary to O. Precisely, we change the way we compute ciphertext ct_0 to $\mathsf{ct}_0 := K$, for a random $K \leftarrow\!\!\$\, \mathbb{G}_1$. This change is indistinguishable under the DDH assumption in $\mathbb{G}_1$. For that we sketch a reduction. Let $(g_1^\alpha, g_1^\beta, g_1^\gamma)$ be an instance of the DDH assumption. The reduction computes prom honestly as defined in Game $\mathbf{G}_9$, but for the ith interaction it sets $K := g_1^\gamma \cdot \sigma_s$ and $\mathsf{coeff}'_0 := g_1^\beta$. Moreover, the reduction changes the way oracle $\hat{\mathsf{H}}$ is simulated in the i_hth query. Namely, for this query, it sets $h := g_1^\alpha$. Note that the only place where value $h_{\mathsf{sn}} = \alpha$ is used is in if the adversary makes query $\hat{\mathsf{H}}(\mathsf{sn}, \sigma_{\bar{k}_j})$ with $\mathsf{BS.Ver}(\mathsf{pk}_{\mathsf{BS},\bar{k}_j}, \mathsf{sn}, \sigma_{\bar{k}_j}) = 1$ for some $j \in [\lambda]$. However, if bad occurs, this will never happen. If bad occurs, the reduction outputs 1, and 0 otherwise. It follows that if $(g_1^\alpha, g_1^\beta, g_1^\gamma)$ is a DDH tuple then conditioned on event bad the reduction simulates $\mathbf{G}_9$ and $\mathbf{G}_{10}$ otherwise.

Finally, it remains to bound the probability of event bad in game $\mathbf{G}_{10}$. The intuition is now that the computation of prom in the ith query to oracle O in $\mathbf{G}_{10}$ does not knowledge of a valid signature σ_s and we can bound the probability of event bad using a reduction from the $\mathsf{EUF\text{-}CMA}$ security of SIG.

We sketch the reduction. The reduction gets as input a public key pk_s^*. It simulates the security game honestly as in $\mathbf{G}_{10}$. In the ith interaction, it uses $\mathsf{pk}_s := \mathsf{pk}_s^*$ instead of sampling a fresh key pair $(\mathsf{pk}_s, \mathsf{sk}_s)$. The corresponding secret key and a signature σ_s is never needed as already mentioned. Now in case event bad occurs, the reduction can return (tx, σ_s). Note that the reduction never used its signing oracle. Therefore, the forgery (tx, σ_s) is fresh. □

D Security Proof of Sweep-UC

Definition D.1 Let EXC be an exchange for SIG and BS as in Definition 5.1. We say that EXC is a secure exchange for SIG and BS if it is secure against malicious buyers and it is secure against malicious sellers.

Definition D.2 Let RP be an redeem protocol for SIG and BS as in Definition 6.1. We say that RP is a secure redeem protocol for SIG and BS if it is secure against malicious services and it is secure against malicious users.

Theorem D.3 *Let* SIG *be a signature scheme with public key entropy* $\omega(\log(\lambda))$*. Let* BS *be a two-move blind signature scheme with unique signatures. Let* EXC *be a secure exchange for* SIG *and* BS *with well distributed signatures. Let* RP *be a secure redeem protocol for* SIG *and* BS.

Then, the protocol Sweep-UC *realizes the functionality* $\mathcal{F}_{\mathsf{ux}}$ *in the synchronous* $(\mathcal{L}^{\mathsf{SIG}}, \mathcal{F}_s)$*-hybrid model with static corruptions.*

Proof. To prove the statement, for any adversary $\mathcal{A}$, we have to present a simulator $\mathcal{S}$, such that for any environment $\mathcal{Z}$ the real world execution and the ideal world simulation is indistinguishable. We will consider two cases separately. In the first case, the sweeper $\mathcal{W}$ is not corrupted, i.e. it is honest. In the second one, it is corrupted. Also, we follow the standard methodology of assuming that $\mathcal{A}$ is the dummy adversary, and thus we omit $\mathcal{A}$ from our description and talk about corrupted parties instead.

Case 1: Honest Sweeper. Consider the case of an honest party $\mathcal{W}$. We will first describe the setting for which we have to give a simulator. Then, we present the overall idea and detailed description of the simulator. Finally, we show indistinguishability from the real world execution.

Setting. The environment can call interfaces `Register`, `AddPayment`, `GetPayment` for honest parties. Precisely, it calls dummy parties which forward these calls to the ideal functionality $\mathcal{F}_{\mathsf{ux}}$. Especially, a dummy party corresponding to the sweeper $\mathcal{W}$ forwards messages that are exchanged between $\mathcal{F}_{\mathsf{ux}}$ and $\mathcal{W}$ to the environment. When honest parties communicate, they do that using the secure channel by definition of the protocol. Therefore, we can assume that the messages sent between honest parties do not have to be simulated. Corrupted parties $\mathcal{P}$ are controlled by the environment. When a corrupted party wants to interact with the sweeper $\mathcal{W}$, the simulator $\mathcal{S}$ takes the role of $\mathcal{W}$ in this interaction, i.e. it simulates the behavior of $\mathcal{W}$ to the corrupted party. To make these interactions consistent with the information that the environment obtains via the dummy parties, the simulator can access the interface of such corrupted parties $\mathcal{P}$ at the ideal functionality $\mathcal{F}_{\mathsf{ux}}$. Additionally, the ideal functionality $\mathcal{F}_{\mathsf{ux}}$ communicates with the global ledger functionality $\mathcal{L}^{\mathsf{SIG}}$. Also, corrupted parties may call this functionality $\mathcal{L}^{\mathsf{SIG}}$. Finally, corrupted parties communicate with the functionality $\mathcal{F}_s$, which is provided by the simulator $\mathcal{S}$. Thus, calls to $\mathcal{F}_s$ are answered by $\mathcal{S}$, and $\mathcal{S}$ has to send the messages that corrupted parties expect on behalf of $\mathcal{F}_s$.

Idea. We present an intuitive overview of our simulator. Note that at a high level, what we want to show is that malicious users can not steal coins from the honest sweeper. In other words, it should not happen that more shared addresses are closed in sub-protocol `GetPayment` than in sub-protocol `AddPayment`. This is also the main bad event that we have to rule out in our simulation. Intuitively, this should follow from the one-more unforgeability of the blind signature scheme BS. To capture this intuition formally, we need to give a reduction to one-more unforgeability. This reduction should satisfy two properties: First, it should query its signing oracle if and only if a shared address is closed in sub-protocol `AddPayment`, i.e. if the sweeper gets coins from a party. Second, whenever a shared address is closed in `GetPayment`, it should obtain a valid blind signature. Then, if the above bad event occurs, the reduction can output a one-more forgery.

To ensure the first property, we have to avoid using the secret key $\mathsf{sk}_{\mathsf{BS}}$ to compute the promise message prom in the sub-protocol `Register`. This can be established using the simulatability of the redeem protocol. Then, we also have to avoid using the secret key $\mathsf{sk}_{\mathsf{BS}}$ in the exchange protocol before the sweeper obtains a valid signature to close the shared address. This is possible using the security of the redeem protocol.

For the second property, we use the extraction that is guaranteed by the security of the redeem protocol. This allows us to extract a blind signature whenever a malicious user closes a shared address to get coins from the sweeper.

A second obstacle that we have to face is induced by the use of an anonymous channel and the blindness of BS. Namely, when a corrupted party interacts with the sweeper in `AddPayment`, the simulator should call the corresponding interface at the ideal functionality. However, at this point we do not know which party actually interacts and which key pk_b it pays to. The solution is to just call the interface on random values, and later change this payment using the interface `ChangePayment`.

Beyond that, there are also some straight-forward things that the simulator has to take care of. For example, when an honest party registers, in the real world the functionality $\mathcal{F}_s$ would send a message about the opening of a shared address to all parties. Therefore, in the ideal world simulation, the simulator has to provide a similar message to the adversary.

Simulator Description. The simulator makes use of simulators and extractors $\mathsf{RP.Sim}, \mathsf{RP.Sim}_{RO}, \mathsf{RP.Ext}$ for the redeem protocol RP and simulators $\mathsf{EXC.Sim}_1, \mathsf{EXC.Sim}_{RO}, \mathsf{EXC.Sim}_2, \mathsf{EXC.Sim}_3$ for the exchange protocol EXC. To give a more formal description of the simulator $\mathcal{S}$, we first describe the data structures that it holds. All of these are initially empty.

- List DSpend: This list contains nonces sn that parties $\mathcal{P}$ submit in `Register`, similar to the list with the same name in the actual protocol. Therefore, these nonces can either come from corrupted $\mathcal{P}$, or be sampled by $\mathcal{S}$ itself, to simulate the behavior of an honest $\mathcal{P}$.
- Map Shared: This maps tuples $(\mathcal{P}, \mathsf{pk}_b)$ to tuples $(\bar{\mathsf{pk}}_{r,\mathcal{W}}, \bar{\mathsf{pk}}_{r,\mathcal{P}}, \bar{\mathsf{sk}}_{r,\mathcal{W}}, \bar{\mathsf{sk}}_{r,\mathcal{P}}, \mathsf{sn})$. It is used by $\mathcal{S}$ to store information about the `Register`(pk_b) sub-protocol.
- List Open: This list contains tuples $(\mathsf{pk}_a, \mathsf{pk}_c)$. Whenever a corrupted party $\mathcal{P}$ completes the `AddPayment` sub-protocol with $\mathcal{S}$ (in the role of $\mathcal{W}$) for public key pk_a, the simulator samples a random key pk_c and inserts such an entry into the list. Entries are removed from the list whenever a corrupted $\mathcal{P}$ successfully closes a shared address in the `GetPayment` sub-protocol.

Next, we give an overview of the bad events, for which $\mathcal{S}$ will abort the entire execution if they occur.

- bad_1: This event occurs if a random nonce is used twice, i.e. an honest party $\mathcal{P}$ (simulated by $\mathcal{S}$) samples a nonce sn in sub-protocol `Register` that is already in DSpend.
- bad_2: Informally, this event occurs if the corrupted parties break security of the redeem protocol RP. More precisely, it occurs if algorithm RP.Ext can not extract a valid blind signature σ_{BS} on message sn for public key $\mathsf{pk}_{\mathsf{BS}}$ from the signature $\sigma_{r,\mathcal{W}}$. Here, $\sigma_{r,\mathcal{W}}$ is the signature that the adversary uses to close a shared address in `GetPayment`, and sn is the nonce sent by the adversary in the corresponding execution of sub-protocol `Register`.
- bad_3: This event occurs if the simulator samples a key pk_c randomly when a corrupted party interacts in `AddPayment` with $\mathcal{W}$, and after that the environment calls `GetPayment`(pk_c).
- bad_4: Informally, this event occurs if the adversary breaks security of the exchange protocol EXC. More precisely, when a corrupted party successfully closes a shared address in `GetPayment` and the list Open is empty, we say that event bad_4 occurs.

Let us now describe the detailed behavior of $\mathcal{S}$ using these data structures and bad events. We will adhere to the following convention: Whenever $\mathcal{S}$ answers calls to $\mathcal{F}_s$ that are not related to protocol interactions, it answers them honestly, including calls to $\mathcal{L}^{\mathsf{SIG}}$. If on the other hand, these calls are related to protocol interactions, the calls to $\mathcal{L}^{\mathsf{SIG}}$ are omitted. Here, calls are related to protocol interactions if they are with respect to shared addresses that are used in interactions.

<u>`Register`, Honest Party $\mathcal{P}$:</u>

1. When $\mathcal{Z}$ calls $\mathcal{F}_{\mathsf{ux}}$ on interface `Register` via a dummy party, $\mathcal{S}$ receives a notification message ("register", $\mathcal{P}, \mathsf{pk}_b$) from $\mathcal{F}_{\mathsf{ux}}$. Then, it samples a random nonce $\mathsf{sn} \leftarrow_{\$} \{0,1\}^{\lambda}$. If sn is already in list DSpend, it sets $\mathsf{bad}_1 := 1$ and aborts the execution. Otherwise, it adds sn to list DSpend.
2. Then, $\mathcal{S}$ generates a shared address as follows: It generates keys by running $(\bar{\mathsf{pk}}_{r,\mathcal{W}}, \bar{\mathsf{sk}}_{r,\mathcal{W}}) \leftarrow \mathsf{SIG.Gen}(1^{\lambda})$ and $(\bar{\mathsf{pk}}_{r,\mathcal{P}}, \bar{\mathsf{sk}}_{r,\mathcal{P}}) \leftarrow \mathsf{SIG.Gen}(1^{\lambda})$ on behalf of functionality $\mathcal{F}_s$. Once $\mathcal{S}$ receives ("registered", $\mathcal{P}, \mathsf{pk}_b$) from $\mathcal{F}_{\mathsf{ux}}$, it sends the message ("openedSharedAddress", $\bar{\mathsf{pk}}_{r,\mathcal{W}}, \bar{\mathsf{pk}}_{r,\mathcal{P}}, \mathsf{pk}_{\mathcal{W}}, \mathsf{amt}$) on behalf of $\mathcal{F}_s$ to all parties.

3. Finally, it sets $\mathsf{Shared}[\mathcal{P}, \mathsf{pk}_b] := (\bar{\mathsf{pk}}_{r,\mathcal{W}}, \bar{\mathsf{pk}}_{r,\mathcal{P}}, \bar{\mathsf{sk}}_{r,\mathcal{W}}, \bar{\mathsf{sk}}_{r,\mathcal{P}}, \mathsf{sn})$.

<u>`Register`, Corrupted Party $\mathcal{P}$:</u>

1. Assume a corrupted $\mathcal{P}$ with $\mathcal{S}$, which plays the role of $\mathcal{W}$, and sends $\mathsf{sn}, \mathsf{pk}_b$ to $\mathcal{S}$. Then, $\mathcal{S}$ first checks if sn is already in list DSpend. If it is, it aborts this interaction as the honest sweeper would do. Otherwise, it adds sn to DSpend, and calls the ideal functionality $\mathcal{F}_{\mathsf{ux}}$ on interface $\mathtt{Register}(\mathsf{pk}_b)$. The functionality $\mathcal{F}_{\mathsf{ux}}$ sends ("register", $\mathcal{P}, \mathsf{pk}_b$) to $\mathcal{S}$, which responds with "noabort". Then, if $\mathcal{F}_{\mathsf{ux}}$ responds with "failDoubleRegister" or "failNoFunds", the simulator aborts the interaction.

2. Otherwise, it simulates opening a shared address for $\mathcal{P}$. Concretely, it generates $(\bar{\mathsf{pk}}_{r,\mathcal{W}}, \bar{\mathsf{sk}}_{r,\mathcal{W}}) \leftarrow \mathsf{SIG.Gen}(1^\lambda)$ and $(\bar{\mathsf{pk}}_{r,\mathcal{P}}, \bar{\mathsf{sk}}_{r,\mathcal{P}}) \leftarrow \mathsf{SIG.Gen}(1^\lambda)$ on behalf of functionality $\mathcal{F}_s$. Then, it sends $(\bar{\mathsf{pk}}_{r,\mathcal{W}}, \bar{\mathsf{pk}}_{r,\mathcal{P}}, \bar{\mathsf{sk}}_{r,\mathcal{P}})$ to $\mathcal{P}$ and the message ("openedSharedAddress", $\bar{\mathsf{pk}}_{r,\mathcal{W}}, \bar{\mathsf{pk}}_{r,\mathcal{P}}, \mathsf{pk}_{\mathcal{W}}, \mathsf{amt}$) on behalf of $\mathcal{F}_s$ to all parties.

3. Next, it simulates the promise message prom for $\mathcal{P}$. To do so, it sets a transaction $\mathsf{tx}_r := (\bar{\mathsf{pk}}_{r,\mathcal{W}}, \bar{\mathsf{pk}}_{r,\mathcal{P}}, \mathsf{pk}_b, \mathsf{amt})$ and redeem parameters $\mathsf{rpar} := (\mathsf{pk}_{\mathsf{BS}}, \bar{\mathsf{pk}}_{r,\mathcal{W}}, \mathsf{tx}_r, \mathsf{sn})$ as in the protocol. Then, it computes a promise prom via $\mathsf{prom} \leftarrow \mathsf{RP.Sim}(\mathsf{rpar}, \bar{\mathsf{sk}}_{r,\mathcal{W}})$. From now on, it uses algorithm $\mathsf{RP.Sim}_{RO}$ to simulate the random oracle related to RP.

4. Finally, it sets $\mathsf{Shared}[\mathcal{P}, \mathsf{pk}_b] := (\bar{\mathsf{pk}}_{r,\mathcal{W}}, \bar{\mathsf{pk}}_{r,\mathcal{P}}, \bar{\mathsf{sk}}_{r,\mathcal{W}}, \bar{\mathsf{sk}}_{r,\mathcal{P}}, \mathsf{sn})$.

<u>`AddPayment`, Honest Party $\mathcal{P}$:</u>

1. When the environment calls $\mathcal{F}_{\mathsf{ux}}$ on interface `AddPayment` via a dummy party, $\mathcal{S}$ receives a message ("addPayment", pk_a) from $\mathcal{F}_{\mathsf{ux}}$. When $\mathcal{S}$ receives ("addPaymentFreeze", pk_a) from $\mathcal{F}_{\mathsf{ux}}$, it responds with "noabort".

2. Then, it generates a shared address as follows: It generates key $(\bar{\mathsf{pk}}_{l,\mathcal{P}}, \bar{\mathsf{sk}}_{l,\mathcal{P}}) \leftarrow \mathsf{SIG.Gen}(1^\lambda)$ and $(\bar{\mathsf{pk}}_{l,\mathcal{W}}, \bar{\mathsf{sk}}_{l,\mathcal{W}}) \leftarrow \mathsf{SIG.Gen}(1^\lambda)$. It sends message ("openedSharedAddress", $\bar{\mathsf{pk}}_{l,\mathcal{P}}, \bar{\mathsf{pk}}_{l,\mathcal{W}}, \mathsf{pk}_a, \mathsf{amt}$) on behalf of the functionality $\mathcal{F}_s$ to all parties.

3. Next, $\mathcal{S}$ simulates the closing of the shared address as follows. It sets $\mathsf{tx}_l := (\bar{\mathsf{pk}}_{l,\mathcal{P}}, \bar{\mathsf{pk}}_{l,\mathcal{W}}, \mathsf{pk}_{\mathcal{W}}, \mathsf{amt})$. Then, it executes $\sigma_{l,\mathcal{P}} \leftarrow \mathsf{SIG.Sig}(\bar{\mathsf{sk}}_{l,\mathcal{P}}, \mathsf{tx}_l)$ and $\sigma_{l,\mathcal{W}} \leftarrow \mathsf{SIG.Sig}(\bar{\mathsf{sk}}_{l,\mathcal{W}}, \mathsf{tx}_l)$. Finally, it sends a message ("closedSharedAddress", $\bar{\mathsf{pk}}_{l,\mathcal{P}}, \bar{\mathsf{pk}}_{l,\mathcal{W}}, \mathsf{pk}_{\mathcal{W}}, \mathsf{amt}, \sigma_{l,\mathcal{P}}, \sigma_{l,\mathcal{W}}$) on behalf of $\mathcal{F}_s$ to all parties.

<u>`AddPayment`, Corrupted Party $\mathcal{P}$:</u>

1. Assume a corrupted party sends a message bsm_1 via an anonymous channel to $\mathcal{S}$ (which plays the role of $\mathcal{W}$) and opens a shared address using a call $\mathcal{F}_s.\mathtt{OpenSh}(T, \mathsf{pk}_a, \mathcal{W}, \mathsf{amt}, \mathsf{sk}_a)$. Then, $\mathcal{S}$ calls the ideal functionality $\mathcal{F}_{\mathsf{ux}}$ via interface $\mathtt{AddPayment}(\mathsf{pk}_a, \mathsf{sk}_a, \mathsf{pk}_c)$ for an arbitrary corrupted party, for some fresh key $(\mathsf{pk}_c, \mathsf{sk}_c) \leftarrow \mathsf{SIG.Gen}(1^\lambda)$. If the environment ever queries $\mathtt{GetPayment}(\mathsf{pk}_c)$ via a dummy party afterwards, the simulator sets $\mathsf{bad}_3 := 1$ and aborts the entire execution.

2. If $\mathcal{F}_{\mathsf{ux}}$ sends "failInvalidKey", $\mathcal{S}$ sends "failInvalidKey" on behalf of $\mathcal{F}_s$. Similarly, if $\mathcal{F}_{\mathsf{ux}}$ aborts with "failNoFunds", $\mathcal{S}$ sends message "failNoFunds" on behalf of $\mathcal{F}_s$.

3. If $\mathcal{F}_{\mathsf{ux}}$ sends ("addPaymentFreeze", pk_a) to $\mathcal{S}$, then $\mathcal{S}$ computes message xm_1 using the simulator $\mathsf{EXC.Sim}_1$, i.e. it runs $\mathsf{xm}_1 \leftarrow \mathsf{EXC.Sim}_1(\mathsf{xpar}, \bar{\mathsf{sk}}_{l,\mathcal{W}})$ for $\mathsf{tx}_l := (\bar{\mathsf{pk}}_{l,\mathcal{P}}, \bar{\mathsf{pk}}_{l,\mathcal{W}}, \mathsf{pk}_{\mathcal{W}}, \mathsf{amt})$ and exchange parameters $\mathsf{xpar} := (\mathsf{pk}_{\mathsf{BS}}, \mathsf{bsm}_1, \bar{\mathsf{pk}}_{l,\mathcal{P}}, \bar{\mathsf{pk}}_{l,\mathcal{W}}, \mathsf{tx}_l)$. It sends xm_1 to the corrupted party.

4. When the corrupted party responds with xm_2, the simulator $\mathcal{S}$ runs $\sigma_{l,\mathcal{W}} \leftarrow \mathsf{SIG.Sig}(\bar{\mathsf{sk}}_{l,\mathcal{W}}, \mathsf{tx}_l)$ as in the protocol. If $\mathsf{EXC.Sim}_2(\mathsf{xm}_2) = 0$, it sends "abort" to $\mathcal{F}_{\mathsf{ux}}$. Otherwise, it runs $\mathsf{bsm}_2 \leftarrow \mathsf{BS.S}(\mathsf{sk}_{\mathsf{BS}}, \mathsf{bsm}_1)$ and $\sigma_{l,\mathcal{P}} \leftarrow \mathsf{EXC.Sim}_3(\mathsf{xm}_2, \mathsf{bsm}_2)$, and sends "noabort" to $\mathcal{F}_{\mathsf{ux}}$. It inserts $(\mathsf{pk}_a, \mathsf{pk}_c)$ into list Open and sends ("closedSharedAddress", $\bar{\mathsf{pk}}_{l,\mathcal{P}}, \bar{\mathsf{pk}}_{l,\mathcal{W}}, \mathsf{pk}_{\mathcal{W}}, \mathsf{amt}, \sigma_{l,\mathcal{P}}, \sigma_{l,\mathcal{W}}$) on behalf of $\mathcal{F}_s$ to all parties.

<u>`GetPayment`, Honest Party $\mathcal{P}$:</u>

1. When $\mathcal{Z}$ calls $\mathcal{F}_{\mathsf{ux}}$ on interface `Register` via a dummy party, $\mathcal{S}$ receives a notification message ("getPayment", $\mathcal{P}, \mathsf{pk}_b$) from $\mathcal{F}_{\mathsf{ux}}$.

2. Once $\mathcal{S}$ receives ("gotPayment", $\mathcal{P}, \mathsf{pk}_b$) from $\mathcal{F}_{\mathsf{ux}}$, it computes the closing signature $(\sigma_{r,\mathcal{W}}, \sigma_{r,\mathcal{P}})$ as follows: It first restores details from the corresponding registration call, i.e. it sets $(\bar{\mathsf{pk}}_{r,\mathcal{W}}, \bar{\mathsf{pk}}_{r,\mathcal{P}}, \bar{\mathsf{sk}}_{r,\mathcal{W}}, \bar{\mathsf{sk}}_{r,\mathcal{P}}, \mathsf{sn}) := \mathsf{Shared}[\mathcal{P}, \mathsf{pk}_b]$. Then, it computes a blind signature $\sigma_{\mathsf{BS}} \leftarrow \mathsf{BS.Sig}(\mathsf{sk}_{\mathsf{BS}}, \mathsf{sn})$. Next, it runs $\sigma_{r,\mathcal{W}} \leftarrow \mathsf{Redeem}(\mathsf{rpar}, \mathsf{prom}, \sigma_{\mathsf{BS}})$ and $\sigma_{r,\mathcal{P}} \leftarrow \mathsf{SIG.Sig}(\bar{\mathsf{sk}}_{r,\mathcal{P}}, \mathsf{tx}_r)$. Finally, it sends ("closedSharedAddress", $\bar{\mathsf{pk}}_{r,\mathcal{W},\mathcal{P}}, \mathsf{pk}_b, \mathsf{amt}, \sigma_{r,\mathcal{W}}, \sigma_{r,\mathcal{P}}$) on behalf of $\mathcal{F}_s$ to all parties.

<u>`GetPayment`, Corrupted Party $\mathcal{P}$:</u>

1. Suppose a corrupted $\mathcal{P}$ calls interface $\mathcal{F}_s$.`CloseSh`$(\bar{\mathsf{pk}}_{r,\mathcal{W}}, \bar{\mathsf{pk}}_{r,\mathcal{P}}, \mathsf{pk}_b, \mathsf{amt}, \sigma_{r,\mathcal{W}}, \sigma_{r,\mathcal{P}})$. If the first two components of $\mathsf{Shared}[\mathcal{P}, \mathsf{pk}_b]$ is not equal to $\bar{\mathsf{pk}}_{r,\mathcal{W}}, \bar{\mathsf{pk}}_{r,\mathcal{P}}$, then $\mathcal{S}$ processes this call as $\mathcal{F}_s$ would do, including the calls to $\mathcal{L}^{\mathsf{SIG}}$.

2. Otherwise, it restores entry $(\bar{\mathsf{pk}}_{r,\mathcal{W}}, \bar{\mathsf{pk}}_{r,\mathcal{P}}, \bar{\mathsf{sk}}_{r,\mathcal{W}}, \bar{\mathsf{sk}}_{r,\mathcal{P}}, \mathsf{sn}) := \mathsf{Shared}[\mathcal{P}, \mathsf{pk}_b]$. Then, $\mathcal{S}$ sets $\mathsf{tx}_r := (\bar{\mathsf{pk}}_{r,\mathcal{W}}, \bar{\mathsf{pk}}_{r,\mathcal{P}}, \mathsf{pk}_b, \mathsf{amt})$ and $\mathsf{rpar} := (\mathsf{pk}_{\mathsf{BS}}, \bar{\mathsf{pk}}_{r,\mathcal{W}}, \mathsf{tx}_r, \mathsf{sn})$. It extracts a blind signature via $\sigma_{\mathsf{BS}} \leftarrow \mathsf{RP.Ext}(\mathsf{rpar}, \bar{\mathsf{sk}}_{r,\mathcal{W}}, \sigma_{r,\mathcal{W}})$ from $\sigma_{r,\mathcal{W}}$. If $\mathsf{BS.Ver}(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sn}, \sigma_{\mathsf{BS}}) = 0$, the simulator $\mathcal{S}$ sets $\mathsf{bad}_2 := 1$ and aborts the entire execution.

3. Otherwise, if the list Open is empty, it sets $\mathsf{bad}_3 := 1$ and aborts the entire execution. Otherwise, let $(\mathsf{pk}_a, \mathsf{pk}_c)$ be an arbitrary entry in Open (e.g. the first). Then, $\mathcal{S}$ removes the entry $(\mathsf{pk}_a, \mathsf{pk}_c)$ from Open and calls the interface `ChangePayment`$(\mathsf{pk}_a, \mathsf{pk}_c, \mathsf{pk}_b)$ of ideal functionality $\mathcal{F}_{\mathsf{ux}}$. Note that this interface will not abort, as the party for which the simulator called `AddPayment`$(\mathsf{pk}_a, \cdot, \mathsf{pk}_c)$ must be corrupted.

4. Finally, it calls `GetPayment`(pk_b). When it receives ("gotPayment", $\mathcal{P}, \mathsf{pk}_b$) from $\mathcal{F}_{\mathsf{ux}}$, it sends the message ("closedSharedAddress", $\bar{\mathsf{pk}}_{r,\mathcal{W}}, \bar{\mathsf{pk}}_{r,\mathcal{P}}, \mathsf{pk}_b, \mathsf{amt}, \sigma_{r,\mathcal{W}}, \sigma_{r,\mathcal{P}}$) to every party.

Analysis. To show that the ideal world simulation using $\mathcal{S}$ is indistinguishable from the real world execution, we present a sequence of hybrid executions. Then, we show that two subsequent hybrid executions are indistinguishable.

- $\mathcal{H}_0$: This hybrid is the real world execution with environment $\mathcal{Z}$. It keeps the same data structures as the simulator $\mathcal{S}$, but does not use them yet.

- $\mathcal{H}_1$: In this hybrid, we rule out bad event bad_1. More precisely, the execution aborts if an honest party $\mathcal{P}$ samples a nonce sn in sub-protocol `Register`, which is already in list DSpend.

- $\mathcal{H}_2$: In this hybrid, we change how the honest sweeper $\mathcal{W}$ interacts with corrupted parties $\mathcal{P}$ in sub-protocol `Register`. Precisely, when corrupted $\mathcal{P}$ sends $\mathsf{sn}, \mathsf{pk}_b$, instead of computing and sending the promise message prom as in the protocol, the message prom is now computed as follows: A transaction $\mathsf{tx}_r := (\bar{\mathsf{pk}}_{r,\mathcal{W}}, \bar{\mathsf{pk}}_{r,\mathcal{P}}, \mathsf{pk}_b, \mathsf{amt})$ and redeem parameters $\mathsf{rpar} := (\mathsf{pk}_{\mathsf{BS}}, \bar{\mathsf{pk}}_{r,\mathcal{W}}, \mathsf{tx}_r, \mathsf{sn})$ are set as in the protocol. Then, prom is computed as $\mathsf{prom} \leftarrow \mathsf{RP.Sim}(\mathsf{rpar}, \bar{\mathsf{sk}}_{r,\mathcal{W}})$, and to answer random oracle queries for the redeem protocol, algorithm $\mathsf{RP.Sim}_{RO}$ is used. Also, we make the change that details about the `Register` protocol are now stored in the map Shared, as in the desciption of $\mathcal{S}$.

- $\mathcal{H}_3$: In this hybrid, we change how sub-protocol `GetPayment` is executed for a corrupted party $\mathcal{P}$. More precisely, consider the case where a corrupted party closes a shared address $(\bar{\mathsf{pk}}_{r,\mathcal{W}}, \bar{\mathsf{pk}}_{r,\mathcal{P}})$ that has been opened in an interaction of the sub-protocol `Register` using signatures $\sigma_{r,\mathcal{W}}, \sigma_{r,\mathcal{P}}$. Note that we can identify this case as in the description of the simulator $\mathcal{S}$ using the map Shared. In this case, the execution runs $\sigma_{\mathsf{BS}} \leftarrow \mathsf{RP.Ext}(\mathsf{rpar}, \bar{\mathsf{sk}}_{r,\mathcal{W}}, \sigma_{r,\mathcal{W}})$, where rpar and $\bar{\mathsf{sk}}_{r,\mathcal{W}}$ are restored using Shared. Then, it runs $b := \mathsf{BS.Ver}(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sn}, \sigma_{\mathsf{BS}})$. If $b = 0$, we say that event bad_2 occurs and the execution aborts.

- $\mathcal{H}_4$: We change how sub-protocol `GetPayment` is run between honest party $\mathcal{P}$ and honest sweeper $\mathcal{W}$. Recall that in this sub-protocol, the blind signature σ_{BS} is used to derive the signature $\sigma_{r,\mathcal{W}}$ using algorithm Redeem from the promise message prom. Here, prom has been sent from $\mathcal{W}$ to $\mathcal{P}$ in sub-protocol `Register` and σ_{BS} is generated during the sub-protocol `AddPayment`. We make the following change. In this hybrid, we now no longer use σ_{BS} that was generated in `AddPayment`, but instead generate σ_{BS} directly via $\sigma_{\mathsf{BS}} \leftarrow \mathsf{BS.Sig}(\mathsf{sk}_{\mathsf{BS}}, \mathsf{sn})$, where sn is the message sent by $\mathcal{P}$ to $\mathcal{W}$ in `Register`.

- $\mathcal{H}_5$: We change how honest parties $\mathcal{P}$ and $\mathcal{W}$ execute the `AddPayment` sub-protocol. Namely, while the signature $\sigma_{l,\mathcal{P}}$ was derived using algorithm Sell as a result of the exchange protocol, this signature is now computed directly using secret key $\bar{\mathsf{sk}}_{l,\mathcal{P}}$. More precisely, the execution first generates the keys $(\bar{\mathsf{pk}}_{l,\mathcal{P}}, \bar{\mathsf{pk}}_{l,\mathcal{W}}, \bar{\mathsf{sk}}_{l,\mathcal{P}}, \bar{\mathsf{sk}}_{l,\mathcal{W}})$ as before. Then, it computes $\sigma_{l,\mathcal{P}}$ via $\sigma_{l,\mathcal{P}} \leftarrow \mathsf{SIG.Sig}(\bar{\mathsf{sk}}_{l,\mathcal{P}}, \mathsf{tx}_l)$, where tx_l is as in the protocol. In particular, the parties do not run the exchange protocol anymore (Note that signatures $\sigma_{l,\mathcal{P}}, \sigma_{l,\mathcal{W}}$ and the blind signature σ_{BS} is computed directly now).
- $\mathcal{H}_6$: We change the execution for the case where a corrupted party interacts with $\mathcal{W}$ in `AddPayment`. Namely, consider the case where a corrupted party sends a message bsm_1 via an anonymous channel to $\mathcal{W}$, and opens a shared address using a call $\mathcal{F}_s.\mathtt{OpenSh}(T, \mathsf{pk}_a, \mathcal{W}, \mathsf{amt}, \mathsf{sk}_a)$. Then, the sweeper $\mathcal{W}$ does not compute xm_1 using algorithm $\mathsf{EXC.Setup}$ anymore, but instead it uses the algorithms $\mathsf{EXC.Sim}_1, \mathsf{EXC.Sim}_{RO}, \mathsf{EXC.Sim}_2, \mathsf{EXC.Sim}_3$. Concretely, it runs $\mathsf{xm}_1 \leftarrow \mathsf{EXC.Sim}_1(\mathsf{xpar}, \bar{\mathsf{sk}}_{l,\mathcal{W}})$ for xpar as before. Then, it sends xm_1 to the corrupted party. When it receives xm_2 in return, it runs $\sigma_{l,\mathcal{W}} \leftarrow \mathsf{SIG.Sig}(\bar{\mathsf{sk}}_{l,\mathcal{W}}, \mathsf{tx}_l)$ as in the protocol. If $\mathsf{EXC.Sim}_2(\mathsf{xm}_2) = 0$, it aborts. Otherwise, it runs $\mathsf{bsm}_2 \leftarrow \mathsf{BS.S}(\mathsf{sk}_{\mathsf{BS}}, \mathsf{bsm}_1)$ and $\sigma_{l,\mathcal{P}} \leftarrow \mathsf{EXC.Sim}_3(\mathsf{xm}_2, \mathsf{bsm}_2)$. Then, it continues as before.
- $\mathcal{H}_7$: We change the execution for the case where a corrupted party interacts in `AddPayment` again. When the corrupted party sends a message bsm_1 via an anonymous channel to $\mathcal{W}$ and opens a shared address using a call $\mathcal{F}_s.\mathtt{OpenSh}(T, \mathsf{pk}_a, \mathcal{W}, \mathsf{amt}, \mathsf{sk}_a)$, the execution generates $(\mathsf{pk}_c, \mathsf{sk}_c) \leftarrow \mathsf{SIG.Gen}(1^\lambda)$. When the interaction between $\mathcal{W}$ and the corrupted party is completed (i.e. the party sent the message xm_2 of protocol `AddPayment` that allowed $\mathcal{W}$ to derive a signature $\sigma_{l,\mathcal{P}}$), an entry $(\mathsf{pk}_a, \mathsf{pk}_c)$ is inserted into list Open. Then, if the environment ever calls `GetPayment`(pk_c) afterwards, we say that event bad_3 occurs and the execution aborts.
- $\mathcal{H}_8$: We add another bad event to the execution. Consider the case where a corrupted party calls the functionality $\mathcal{F}_s$ via $\mathcal{F}_s.\mathtt{CloseSh}(\bar{\mathsf{pk}}_{r,\mathcal{W}}, \bar{\mathsf{pk}}_{r,\mathcal{P}}, \mathsf{pk}_b, \mathsf{amt}, \sigma_{r,\mathcal{W}}, \sigma_{r,\mathcal{P}})$. If this call closes a shared address that was opened in an interaction of a corrupted party with $\mathcal{W}$ in the `Register` sub-protocol, then the execution tries to remove an arbitrary entry $(\mathsf{pk}_a, \mathsf{pk}_c)$ from list Open. If this fails because the list is empty, we say that bad_4 occurs and the execution aborts.
- $\mathcal{H}_9$: This is the ideal world simulation using simulator $\mathcal{S}$ as described above.

Claim D.4 $\mathcal{H}_0$ and $\mathcal{H}_1$ are indistinguishable.

Proof. Note that the distinguishing probability of these hybrids can be bounded by the probability of event bad_1. As nonces sn sampled by honest parties have λ bits of entropy, event bad_1 can only occur with negligible probability. □

Claim D.5 $\mathcal{H}_1$ and $\mathcal{H}_2$ are indistinguishable, if $(\mathsf{RP.Sim}, \mathsf{RP.Sim}_{RO})$ is a simulator against malicious users for RP.

Proof. The statement can be proven using a reduction from the simulatability game of RP. Precisely, the reduction gets $\mathsf{pk}_{\mathsf{BS}}, \mathsf{sk}_{\mathsf{BS}}$ as input and access to an oracle O. It uses $\mathsf{sk}_{\mathsf{BS}}$ to simulate interactions with honest users in `Register` and interactions with arbitrary users in `AddPayment`, according to hybrid $\mathcal{H}_1$. When a corrupted party $\mathcal{P}$ interacts with $\mathcal{W}$ (provided by the reduction) in `Register`, the reduction uses oracle O to simulate message prom. Concretely, assume that sn is not yet in DSpend. Then, to compute message prom, the reduction sends sn to O and gets a key $\bar{\mathsf{pk}}_{r,\mathcal{W}}$ in return. It generates $\bar{\mathsf{pk}}_{r,\mathcal{P}}$ and sets tx_r as in the protocol. Then, it sends tx_r to O and obtains prom from O. It continues the execution as in $\mathcal{H}_1$. Finally, it outputs whatever $\mathcal{Z}$ outputs.

It is easy to see that the reduction perfectly simulates $\mathcal{H}_1$, if the internal bit b of the simulation game of RP is $b = 0$, and $\mathcal{H}_2$ otherwise.

Finally, note that introducing the map Shared is only a conceptual change that is not visible for $\mathcal{Z}$. □

Claim D.6 $\mathcal{H}_2$ and $\mathcal{H}_3$ are indistinguishable, if $\mathsf{RP.Ext}$ is a an extractor against malicious users for RP and $(\mathsf{RP.Sim}, \mathsf{RP.Sim}_{RO})$.

Proof. To show the claim, we sketch a reduction from the extractablility game of RP. The reduction gets $\mathsf{pk}_{\mathsf{BS}}, \mathsf{sk}_{\mathsf{BS}}$ as input and access to an oracle O. It simulates the execution as in $\mathcal{H}_2$. However, when a corrupted party $\mathcal{P}$ interacts with $\mathcal{W}$ in the `Register` sub-protocol, it does not simulate the execution as in $\mathcal{H}_2$. Instead, it uses oracle O as follows. When $\mathcal{P}$ sends a nonce sn and a public key pk_b, the reduction passes sn to O. It obtains a key $\bar{\mathsf{pk}}_{r,\mathcal{W}}$ in return, and generates $\bar{\mathsf{pk}}_{r,\mathcal{P}}$ and sets tx_r as in the protocol. It sends tx_r to O, and obtains message prom in return. The reduction sends prom to $\mathcal{P}$, as in the protocol. Later, when a party closes the shared address $(\bar{\mathsf{pk}}_{r,\mathcal{W}}, \bar{\mathsf{pk}}_{r,\mathcal{P}})$ using signatures $\sigma_{r,\mathcal{W}}, \sigma_{r,\mathcal{P}}$, the reduction passes $\sigma_{r,\mathcal{W}}$ to oracle O. The rest is simulated as in $\mathcal{H}_2$.

It is easy to see that the reduction perfectly simulates $\mathcal{H}_2$. Furthermore, note that the variable bad defined in the extractability game of RP is set to 1 if and only if event bad_2 occurs. Thus, we can bound the probability of event bad_2 by the advantage of the above reduction. Clearly, the distinguishing advantage is upper bounded by the probability of bad_2. □

Claim D.7 $\mathcal{H}_3$ and $\mathcal{H}_4$ are indistinguishable, if BS has unique signatures.

Proof. As BS has unique signatures, the distribution of σ_{BS} computed directly (as in $\mathcal{H}_4$) is the same as the distribution of σ_{BS} computed using the exchange (as in $\mathcal{H}_3$). Therefore, the view of corrupted parties and the environment $\mathcal{Z}$ in both hybrids is the same. □

Claim D.8 $\mathcal{H}_4$ and $\mathcal{H}_5$ are indistinguishable, if EXC has well distributed signatures.

Proof. This follows directly from the definition of well distributed signatures. □

Claim D.9 $\mathcal{H}_5$ and $\mathcal{H}_6$ are indistinguishable, if EXC is secure against malicious buyers.

Proof. Note that due to the previous changes, the secret key $\mathsf{sk}_{\mathsf{BS}}$ is only needed in interactions of the sub-protocol `AddPayment`. Furthermore, in interactions between honest parties it is only needed to compute a blind signature directly, and not using the exchange protocol.

Thus, we can give a reduction against the security of EXC that interpolates between $\mathcal{H}_5$ and $\mathcal{H}_6$. The reduction gets $\mathsf{pk}_{\mathsf{BS}}$ as input and access to an oracle O^* and a signing oracle O. It simulates $\mathcal{H}_5$, except for the following changes. First, when an honest party $\mathcal{P}$ interacts with $\mathcal{W}$ in `AddPayment`, the final blind signature σ_{BS} is computed using the signing oracle O. Second, when a corrupted party interacts with $\mathcal{W}$ in `AddPayment`, the oracle O^* is used to simulate the exchange. Concretely, when the corrupted party sends bsm_1 to $\mathcal{W}$ and opens a shared address, the reduction calls oracle O^* and obtains a key $\bar{\mathsf{pk}}_{l,\mathcal{W}}$. This key is then used as part of the shared address $(\bar{\mathsf{pk}}_{l,\mathcal{P}}, \bar{\mathsf{pk}}_{l,\mathcal{W}})$. Then, the reduction defines a transaction tx_l as in the protocol and sends $\bar{\mathsf{pk}}_{l,\mathcal{P}}, \mathsf{tx}_l$ and bsm_1 to oracle O^*. The oracle returns xm_1, and the reduction sends xm_1 to the corrupted party, obtaining xm_2 in return. The reduction passes xm_2 to O^* and obtains signatures $\sigma_{l,\mathcal{P}}, \sigma_{l,\mathcal{W}}$ in return. The rest of the simulation is as before, using these signatures. Finally, the reduction forwards whatever the environment outputs. □

Claim D.10 $\mathcal{H}_6$ and $\mathcal{H}_7$ are indistinguishable, if SIG has public key entropy $\omega(\log(\lambda))$.

Proof. Clearly, the distinguishing advantage between the two hybrids can be bounded by the probability of event bad_3. Note that the environment obtains no information about the key pk_c. Therefore, the probability that the environment queries `GetPayment` for that key is negligible, by the assumption about entropy of public keys. □

Claim D.11 $\mathcal{H}_7$ and $\mathcal{H}_8$ are indistinguishable, if BS is one-more unforgeable.

Proof. Clearly, the distinguishing advantage between $\mathcal{H}_7$ and $\mathcal{H}_8$ can be upper bounded by the probability of event bad_4. We bound the probability of bad_4 using a reduction against the one-more unforgeability of BS. The reduction gets $\mathsf{pk}_{\mathsf{BS}}$ as input and access to a signer oracle O. It simulates $\mathcal{H}_7$, with the following modifications: First, to compute the blind signature σ_{BS} in interactions between honest parties, the reduction uses signer oracle O. We call these queries *queries of the first kind.* Second, when a corrupted party interacts with $\mathcal{W}$ in `AddPayment`, the reduction simulates everything as in $\mathcal{H}_7$, except for the computation of signature $\sigma_{l,\mathcal{P}}$. To compute $\sigma_{l,\mathcal{P}}$, it first queries the signer oracle O on input bsm_1, obtaining bsm_2 in return. We call these queries *queries of the second kind.* Then, it runs $\sigma_{l,\mathcal{P}} \leftarrow \mathsf{EXC.Sim}_3(\mathsf{xm}_2, \mathsf{bsm}_2)$ as in $\mathcal{H}_7$. When event bad_4 occurs, let Σ_{hon} denote the list of pairs $(\mathsf{sn}, \sigma_{\mathsf{BS}})$ that are computed by honest parties. Let Σ_{corr} denote the list of pairs $(\mathsf{sn}, \sigma_{\mathsf{BS}})$, for which the execution extracted the blind signature σ_{BS} for sn when a corrupted party closed a shared address that has been opened in `Register`. The reduction outputs $\Sigma_{hon} \cup \Sigma_{corr}$.

First, it is clear that the reduction perfectly simulates execution $\mathcal{H}_7$. Next, we want to argue that the reduction outputs a valid one-more forgery if event bad_4 occurs. To see that, note that due to the usage of list DSpend and the event bad_1, we know that all sn in the reductions final output are distinct. Further, all σ_{BS} are valid. This is because σ_{BS} in Σ_{hon} are computed honestly, and σ_{BS} in Σ_{corr} are valid by the definition of bad_2. It remains to argue that the reduction returned more pairs than the number of queries to the signer oracle O.

Let k_{add} denote the number of entries that are added to list Open, and k_{rem} the number of times the reduction tried to remove an entry from list Open. If bad_4 occurs, we have

$$k_{add} < k_{rem}.$$

Further, note that queries of the second kind occur if and only if an entry is added to list Open. Also, the number of queries of the first kind is exactly $|\Sigma_{hon}|$. Therefore, the number of queries that the reduction made is

$$k_{add} + |\Sigma_{hon}|.$$

Next, observe that whenever the reduction tries to remove an entry from list Open, if extracted a blind signature σ_{BS} before, leading to one entry in Σ_{corr}. Therefore, we have $|\Sigma_{corr}| = k_{rem}$. We conclude with

$$k_{add} + |\Sigma_{hon}| < k_{rem} + |\Sigma_{hon}| = |\Sigma_{corr}| + |\Sigma_{hon}|.$$

□

Claim D.12 $\mathcal{H}_8$ and $\mathcal{H}_9$ are indistinguishable.

Proof. We note that the execution in $\mathcal{H}_8$, including the simulation of functionality $\mathcal{F}_s$ is exactly as in the ideal world simulation with simulator $\mathcal{S}$. Note that whenever $\mathcal{S}$ uses $\mathcal{F}_{\mathsf{ux}}$ to simulate $\mathcal{F}_s$, this will lead to exactly the same calls to $\mathcal{L}$. □

Case 2: Corrupted Sweeper. Now, consider the case of a corrupted party $\mathcal{W}$. Again, we will first describe the overall setting and the idea of the proof. Then, we give a description of our simulator and show indistinguishability from the real world execution.

Setting. The setting is very similar to the setting for the case of an honest $\mathcal{W}$. The only difference is that the party $\mathcal{W}$ is corrupted now. Thus, the simulator $\mathcal{S}$ can access the interfaces corresponding to $\mathcal{W}$ of the ideal functionality $\mathcal{F}_{\mathsf{ux}}$. In general, when the environment calls one of the interfaces `Register`, `AddPayment`, `GetPayment` for an honest $\mathcal{P}_i$ via a dummy party, the simulator gets notified by $\mathcal{F}_{\mathsf{ux}}$ and has to simulate the interaction of the corresponding sub-protocol to the corrupted parties. As $\mathcal{W}$ is part of every sub-protocol, $\mathcal{S}$ has to provide the appropriate messages to $\mathcal{W}$.

Idea. We describe the main challenges that we encounter and how we solve them. On an intuitive level, we want to show two security claims. First, the malicious sweeper should not be able to link `Register`, `GetPayment` interactions to `AddPayment` interactions. Second, the malicious sweeper should not be able to steal coins. This means that whenever a promise message prom sent by the sweeper in `Register` gets verified, it should also lead to a valid signature once the blind signature is input into Redeem. Furthermore, we have to make sure that whenever the sweeper learns a signature to close the shared address in `AddPayment`, the honest user should learn a blind signature.

Let us now see how these two parts come up on a technical level during the simulation. The first part comes up when the environment calls `AddPayment` via a dummy party. Note that in this case, the simulator only gets notified that some public key pk_a pays, but it does not see which dummy party has been called and which public key pk_b receives the payment. Therefore, we have to simulate the `AddPayment` interaction to the corrupted $\mathcal{W}$, without knowing the actual nonce sn that would be signed in the real world execution. To do this, we make use of the anonymous channel and the blindness of BS, and let $\mathcal{W}$ blindly sign a random nonce sn' instead.

For the second part, we know that when honest parties register and add a payment in the ideal world simulation, the resulting call to `GetPayment` will lead to coins being transfered to pk_b. Thus, we also have to make sure that this is consistent with the interaction between the simulator and corrupted $\mathcal{W}$. To do this, we use the security of the redeem protocol and the exchange protocol.

In combination, these two parts lead to another obstacle. As we have pointed out, we obtain blind signatures on random nonces in the simulation of `AddPayment`. Then, when we get notified by $\mathcal{F}_{\mathsf{ux}}$ that an honest party got a payment, we have to simulate the signature that closes the shared address. This signature has to be distributed exactly as it would be in the real world, which is why we can not just compute it from scratch. Instead, we should use the blind signature on sn to derive the transaction signature, where sn is the nonce used in the corresponding simulation of `Register`. Due to the way we simulate `AddPayment`, we do not have a blind signature on sn. To solve this, we make use of the strong security notion for the redeem protocol that allows us to extract this blind signature from the promise message prom sent by $\mathcal{W}$ in `GetPayment`. Our assumption that blind signatures are unique implies that the resulting transaction signature is exactly distributed as it would be in the real world, where an honest user derives it using the blind signature that it learned in `AddPayment`.

Simulator Description. We first describe the data structures that the simulator $\mathcal{S}$ holds. All of these are initially empty.

- List DSpend: This list contains nonces sn that honest parties $\mathcal{P}$ submit in `Register`. We emphasize that compared to the actual protocol, this list only contains the nonces of honest parties.
- Map Shared: This maps tuples $(\mathcal{P}, \mathsf{pk}_b)$ to tuples $(\bar{\mathsf{pk}}_{r,\mathcal{W}}, \bar{\mathsf{pk}}_{r,\mathcal{P}}, \bar{\mathsf{sk}}_{r,\mathcal{W}}, \bar{\mathsf{sk}}_{r,\mathcal{P}}, \mathsf{sn}, \sigma_{r,\mathcal{W}})$. It is used by $\mathcal{S}$ to store information about the `Register(`pk_b`)` sub-protocol. Note that compared to the case of an honest sweeper, we additionally store signatures $\sigma_{r,\mathcal{W}}$ of transactions in this list.

Next, we give an overview of the bad events, for which $\mathcal{S}$ will abort the entire execution if they occur.

- bad_1: This event occurs if a random nonce is used twice by honest parties. More precisely, it occurs if an honest party $\mathcal{P}$ (simulated by $\mathcal{S}$) samples a nonce sn in sub-protocol `Register` that is already in DSpend.
- bad_2: This event occurs if the algorithm $\mathsf{RP.Ext}$ can not extract a valid blind signature σ_{BS} from the promise message prom or it does not lead to a valid transaction signature $\sigma_{r,\mathcal{W}}$. Concretely, when an honest party interacts with $\mathcal{W}$ in sub-protocol `Register` by sending $\mathsf{sn}, \mathsf{pk}_b$, and $\mathcal{W}$ sends prom, let $\sigma_{\mathsf{BS}} \leftarrow \mathsf{RP.Ext}(\mathsf{rpar}, \mathsf{prom}, \mathcal{Q})$ and $\sigma_{r,\mathcal{W}} \leftarrow \mathsf{Redeem}(\mathsf{rpar}, \mathsf{prom}, \sigma_{\mathsf{BS}})$, where $\mathcal{Q}$ is the list of random oracle queries that corrupted parties made. Then, the bad event occurs, if we have $\mathsf{BS.Ver}(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sn}, \sigma_{\mathsf{BS}}) = 0$ or $\mathsf{SIG.Ver}(\bar{\mathsf{pk}}_{r,\mathcal{W}}, \mathsf{tx}_r, \sigma_{r,\mathcal{W}}) = 0$. Here, $\bar{\mathsf{pk}}_{r,\mathcal{W}}, \mathsf{tx}_r$, and rpar are as in the protocol.
- $\mathsf{bad}_{3,1}$: This event occurs when an honest user can not derive a valid blind signature when $\mathcal{W}$ closes the shared address in sub-protocol `AddPayment`. More formally, consider the case where an honest user $\mathcal{P}$ runs the sub-protocol `AddPayment` with $\mathcal{W}$. Then, $\mathcal{P}$ first inputs sn into $\mathsf{BS.U}_1$ and sends the resulting message bsm_1 to $\mathcal{W}$. Next, it opens a shared address $(\bar{\mathsf{pk}}_{l,\mathcal{P}}, \bar{\mathsf{pk}}_{l,\mathcal{W}})$ using the functionality $\mathcal{F}_s$. Assume that $\mathcal{W}$ sent message xm_1 and received xm_2 from $\mathcal{P}$ in return. Further, assume that $\mathcal{W}$ closes the shared address $(\bar{\mathsf{pk}}_{l,\mathcal{P}}, \bar{\mathsf{pk}}_{l,\mathcal{W}})$ using signatures $(\sigma_{l,\mathcal{P}}, \sigma_{l,\mathcal{W}})$. Honest party $\mathcal{P}$ runs $\mathsf{bsm}_2 := \mathsf{Get}(\mathsf{xpar}, \mathsf{xm}_1, \mathsf{xm}_2, \sigma_{l,\mathcal{P}}, \sigma_{l,\mathcal{W}})$ and computes σ_{BS} from bsm_2 using algorithm $\mathsf{BS.U}_2$. Then, the bad event occurs if $\mathsf{BS.Ver}(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sn}, \sigma_{\mathsf{BS}}) = 0$.
- $\mathsf{bad}_{3,2}$: This event occurs if in the same situation as for $\mathsf{bad}_{3,1}$, $\mathcal{W}$ closes the shared address $(\bar{\mathsf{pk}}_{l,\mathcal{P}}, \bar{\mathsf{pk}}_{l,\mathcal{W}})$ before seeing message xm_2. This includes the case where $\mathcal{W}$ did not send xm_1, but closes the shared address.

Let us now describe the detailed behavior of $\mathcal{S}$. As for the case of an honest sweeper, we will adhere to the following convention: Whenever $\mathcal{S}$ answers calls to $\mathcal{F}_s$ that are not related to protocol interactions that include honest parties, it answers them honestly, including calls to $\mathcal{L}^{\mathsf{SIG}}$. For instance, these calls may occur when corrupted $\mathcal{W}$ and a corrupted $\mathcal{P}$ run the protocol. If on the other hand, these calls are related to protocol interactions with honest parties, the calls to $\mathcal{L}^{\mathsf{SIG}}$ are omitted (this is because in such a case these calls are issued by functionality $\mathcal{F}_{\mathsf{ux}}$). Calls are related to protocol interactions if they are with respect to shared addresses that are used in interactions. For the following description, note that the interaction between corrupted $\mathcal{P}$ and corrupted $\mathcal{W}$ does not have to be simulated for our protocol.

`Register`, Honest Party $\mathcal{P}$:

1. When $\mathcal{Z}$ calls $\mathcal{F}_{\mathsf{ux}}$ on interface `Register` via a dummy party, $\mathcal{S}$ receives a notification message ("register", $\mathcal{P}, \mathsf{pk}_b$) from $\mathcal{F}_{\mathsf{ux}}$. Then, it samples a random nonce $\mathsf{sn} \leftarrow\!\!\$\ \{0,1\}^{\lambda}$. If sn is already in list DSpend, it sets $\mathsf{bad}_1 := 1$ and aborts the execution. Otherwise, it adds sn to list DSpend and sends $\mathsf{sn}, \mathsf{pk}_b$ to the corrupted $\mathcal{W}$.

2. When $\mathcal{W}$ calls $\mathcal{F}_s.\mathtt{OpenSh}(T, \mathsf{pk}_{\mathcal{W}}, \mathcal{P}, \mathsf{amt}, \mathsf{sk}_{\mathcal{W}})$, the simulator $\mathcal{S}$ simulates the interface `OpenSh`, except for the calls to $\mathcal{L}^{\mathsf{SIG}}$. During this simulation, it generates $(\bar{\mathsf{pk}}_{r,\mathcal{W}}, \bar{\mathsf{sk}}_{r,\mathcal{W}}) \leftarrow \mathsf{SIG.Gen}(1^{\lambda})$ and $(\bar{\mathsf{pk}}_{r,\mathcal{P}}, \bar{\mathsf{sk}}_{r,\mathcal{P}}) \leftarrow \mathsf{SIG.Gen}(1^{\lambda})$ on behalf of $\mathcal{F}_s$. Once $\mathcal{S}$ receives ("registered", $\mathcal{P}, \mathsf{pk}_b$) from $\mathcal{F}_{\mathsf{ux}}$, it sends ("openedSharedAddress", $\bar{\mathsf{pk}}_{r,\mathcal{W}}, \bar{\mathsf{pk}}_{r,\mathcal{P}}, \mathsf{pk}_{\mathcal{W}}, \mathsf{amt}$) on behalf of $\mathcal{F}_s$ to all parties.

3. The simulator $\mathcal{S}$ sets $\mathsf{tx}_r := (\bar{\mathsf{pk}}_{r,\mathcal{W}}, \bar{\mathsf{pk}}_{r,\mathcal{P}}, \mathsf{pk}_b, \mathsf{amt})$ and $\mathsf{rpar} := (\mathsf{pk}_{\mathsf{BS}}, \bar{\mathsf{pk}}_{r,\mathcal{W}}, \mathsf{tx}_r, \mathsf{sn})$ as an honest party would do in the protocol. Then, when $\mathcal{W}$ sends the promise message prom, the simulator $\mathcal{S}$ checks if $\mathsf{VerPromise}(\mathsf{rpar}, \mathsf{prom}) = 1$. If this does not hold, it sends "abort" to $\mathcal{F}_{\mathsf{ux}}$.

4. Otherwise, $\mathcal{S}$ runs $\sigma_{\mathsf{BS}} \leftarrow \mathsf{RP.Ext}(\mathsf{rpar}, \mathsf{prom}, \mathcal{Q})$ and $\sigma_{r,\mathcal{W}} \leftarrow \mathsf{Redeem}(\mathsf{rpar}, \mathsf{prom}, \sigma_{\mathsf{BS}})$, where $\mathcal{Q}$ is the list of random oracle queries that corrupted parties made so far. Then, if $\mathsf{BS.Ver}(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sn}, \sigma_{\mathsf{BS}}) = 0$ or $\mathsf{SIG.Ver}(\bar{\mathsf{pk}}_{r,\mathcal{W}}, \mathsf{tx}_r, \sigma_{r,\mathcal{W}}) = 0$, the simulator sets $\mathsf{bad}_2 := 1$ and aborts the execution.

5. The simulator $\mathcal{S}$ sets $\mathsf{Shared}[\mathcal{P}, \mathsf{pk}_b] := (\bar{\mathsf{pk}}_{r,\mathcal{W}}, \bar{\mathsf{pk}}_{r,\mathcal{P}}, \bar{\mathsf{sk}}_{r,\mathcal{W}}, \bar{\mathsf{sk}}_{r,\mathcal{P}}, \mathsf{sn}, \sigma_{r,\mathcal{W}})$.

`AddPayment`, Honest Party $\mathcal{P}$:

1. When the environment calls $\mathcal{F}_{\mathsf{ux}}$ on interface `AddPayment` via a dummy party, $\mathcal{S}$ receives a message ("addPayment", pk_a) from $\mathcal{F}_{\mathsf{ux}}$.

2. The simulator $\mathcal{S}$ samples $\mathsf{sn}' \leftarrow\!\!\$\ \{0,1\}^{\lambda}$, runs $(\mathsf{bsm}_1, St) \leftarrow \mathsf{BS.U}_1(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sn}')$ and sends bsm_1 to $\mathcal{W}$ via the anonymous channel.

3. When $\mathcal{S}$ receives ("addPaymentFreeze", pk_a) from $\mathcal{F}_{\mathsf{ux}}$, it simulates the opening of a shared address as follows: It generates keys $(\bar{\mathsf{pk}}_{l,\mathcal{P}}, \bar{\mathsf{sk}}_{l,\mathcal{P}}) \leftarrow \mathsf{SIG.Gen}(1^{\lambda})$ and $(\bar{\mathsf{pk}}_{l,\mathcal{W}}, \bar{\mathsf{sk}}_{l,\mathcal{W}}) \leftarrow \mathsf{SIG.Gen}(1^{\lambda})$. It sends ("openedSharedAddress", $\bar{\mathsf{pk}}_{l,\mathcal{P}}, \bar{\mathsf{pk}}_{l,\mathcal{W}}, \mathsf{pk}_a, \mathsf{amt}$) on behalf of the functionality $\mathcal{F}_s$ to all parties.

4. If this shared address $(\bar{\mathsf{pk}}_{l,\mathcal{P}}, \bar{\mathsf{pk}}_{l,\mathcal{W}})$ is closed by a corrupted party before the message xm_2 (see below) is sent, $\mathcal{S}$ sets $\mathsf{bad}_{3,2} := 1$ and aborts the entire execution. If $\mathcal{W}$ does not send xm_1, then $\mathcal{S}$ sends "abort" to $\mathcal{F}_{\mathsf{ux}}$.

5. The simulator $\mathcal{S}$ sets $\mathsf{tx}_l := (\bar{\mathsf{pk}}_{l,\mathcal{P}}, \bar{\mathsf{pk}}_{l,\mathcal{W}}, \mathsf{pk}_{\mathcal{W}}, \mathsf{amt})$ and $\mathsf{xpar} := (\mathsf{pk}_{\mathsf{BS}}, \mathsf{bsm}_1, \bar{\mathsf{pk}}_{l,\mathcal{P}}, \bar{\mathsf{pk}}_{l,\mathcal{W}}, \mathsf{tx}_l)$ as in the protocol. When $\mathcal{W}$ sends xm_1, the simulator runs $\mathsf{xm}_2 \leftarrow \mathsf{Buy}(\mathsf{xpar}, \bar{\mathsf{sk}}_{l,\mathcal{P}}, \mathsf{xm}_1)$ and sends xm_2 to $\mathcal{W}$.

6. When $\mathcal{W}$ closes the shared address $(\bar{\mathsf{pk}}_{l,\mathcal{P}}, \bar{\mathsf{pk}}_{l,\mathcal{W}})$ via $\mathcal{F}_s.\mathtt{CloseSh}(\bar{\mathsf{pk}}_{l,\mathcal{P}}, \bar{\mathsf{pk}}_{l,\mathcal{W}}, \mathsf{pk}_{\mathcal{W}}, \mathsf{amt}, \sigma_{l,\mathcal{P}}, \sigma_{l,\mathcal{W}})$, $\mathcal{S}$ simulates `CloseSh` except for calls to $\mathcal{L}^{\mathsf{SIG}}$, and sends "noabort" to $\mathcal{F}_{\mathsf{ux}}$. During that, it also sends ("closedSharedAddress", $\bar{\mathsf{pk}}_{l,\mathcal{P}}, \bar{\mathsf{pk}}_{l,\mathcal{W}}, \mathsf{pk}_{\mathcal{W}}, \mathsf{amt}, \sigma_{l,\mathcal{P}}, \sigma_{l,\mathcal{W}}$) on behalf of $\mathcal{F}_s$ to all parties. Then, it runs $\mathsf{bsm}_2 := \mathsf{Get}(\mathsf{xpar}, \mathsf{xm}_1, \mathsf{xm}_2, \sigma_{l,\mathcal{P}}, \sigma_{l,\mathcal{W}})$ and $\sigma_{\mathsf{BS}} \leftarrow \mathsf{BS.U}_2(St, \mathsf{bsm}_2)$. It sets $\mathsf{bad}_{3,1} := 1$ and aborts the entire execution if $\mathsf{BS.Ver}(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sn}, \sigma_{\mathsf{BS}}) = 0$.

`GetPayment`, Honest Party $\mathcal{P}$:

1. When $\mathcal{Z}$ calls $\mathcal{F}_{\mathsf{ux}}$ on interface `GetPayment` via a dummy party, $\mathcal{S}$ receives a notification message ("getPayment", $\mathcal{P}_i, \mathsf{pk}_b$) from $\mathcal{F}_{\mathsf{ux}}$.

2. Once $\mathcal{S}$ receives ("gotPayment", $\mathcal{P}, \mathsf{pk}_b$) from $\mathcal{F}_{\mathsf{ux}}$, it sets $(\bar{\mathsf{pk}}_{r,\mathcal{W}}, \bar{\mathsf{pk}}_{r,\mathcal{P}}, \bar{\mathsf{sk}}_{r,\mathcal{W}}, \bar{\mathsf{sk}}_{r,\mathcal{P}}, \mathsf{sn}, \sigma_{r,\mathcal{W}}) := \mathsf{Shared}[\mathcal{P}, \mathsf{pk}_b]$. It computes $\sigma_{r,\mathcal{P}} \leftarrow \mathsf{SIG.Sig}(\bar{\mathsf{sk}}_{r,\mathcal{P}}, \mathsf{tx}_r)$.

3. Finally, it sends ("closedSharedAddress", $\bar{\mathsf{pk}}_{r,\mathcal{W},\mathcal{P}}, \mathsf{pk}_b, \mathsf{amt}, \sigma_{r,\mathcal{W}}, \sigma_{r,\mathcal{P}}$) on behalf of $\mathcal{F}_s$ to all parties.

Analysis. We show that the real world execution is indistinguishable from the ideal world simulation by giving a sequence of hybrid executions and showing that subsequent hybrid executions are indistinguishable.

- $\mathcal{H}_0$: This is the real world execution with environment $\mathcal{Z}$. It keeps the same data structures as the simulator $\mathcal{S}$. Let DSpend denote the list of nonces sn used by honest parties, as it is used by $\mathcal{S}$.

- $\mathcal{H}_1$: In this hybrid, the execution aborts whenever event bad_1 occurs. That is, if an honest party samples a nonce sn that is already in list DSpend.

- $\mathcal{H}_2$: In this hybrid, we change how `Register` is executed for honest parties $\mathcal{P}$. Namely, when $\mathcal{W}$ sends the promise prom, the execution runs $\sigma_{\mathsf{BS}} \leftarrow \mathsf{RP.Ext}(\mathsf{rpar}, \mathsf{prom}, \mathcal{Q})$ and $\sigma_{r,\mathcal{W}} \leftarrow \mathsf{Redeem}(\mathsf{rpar}, \mathsf{prom}, \sigma_{\mathsf{BS}})$, where $\mathcal{Q}$ is the list of random oracle queries that corrupted parties made so far. If $\mathsf{BS.Ver}(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sn}, \sigma_{\mathsf{BS}}) = 0$ or $\mathsf{SIG.Ver}(\bar{\mathsf{pk}}_{r,\mathcal{W}}, \mathsf{tx}_r, \sigma_{r,\mathcal{W}}) = 0$, we say that the event bad_2 occurs and the execution aborts. Otherwise, we now store the details of this sub-protocol in the map Shared as described for $\mathcal{S}$.

- $\mathcal{H}_3$: In this hybrid, we add additional bad events for which the execution aborts whenever they occur. Namely, the execution aborts if bad events $\mathsf{bad}_{3,1}$ or $\mathsf{bad}_{3,2}$ occur. Concretely, in an execution of the sub-protocol `AddPayment` for honest party $\mathcal{P}$, the event $\mathsf{bad}_{3,1}$ occurs if no valid blind signature σ_{BS} can be obtained from the signatures $(\sigma_{l,\mathcal{P}}, \sigma_{l,\mathcal{W}})$ using algorithms Get and $\mathsf{BS.U}_2$. The event $\mathsf{bad}_{3,2}$ occurs if a corrupted party closes the shared address $(\bar{\mathsf{pk}}_{l,\mathcal{P}}, \bar{\mathsf{pk}}_{l,\mathcal{W}})$ before the honest party $\mathcal{P}$ sends xm_2.

- $\mathcal{H}_4$: In this hybrid, we change how `GetPayment` is executed for honest parties $\mathcal{P}$. Recall that in previous hybrids, the party uses the blind signature derived in sub-protocol `AddPayment` and runs algorithm Redeem to obtain the signature that is used to close the shared address. Now, honest parties instead use the signature $\sigma_{r,\mathcal{W}}$ that is stored in Shared.

- $\mathcal{H}_5$: In this hybrid, we change which nonces sn are blindly signed in executions of `AddPayment` for honest parties $\mathcal{P}$. Recall that in previous hybrids, party $\mathcal{P}$ runs $(\mathsf{bsm}_1, St) \leftarrow \mathsf{BS.U}_1(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sn})$, sends bsm_1 to $\mathcal{W}$ and interacts in the exchange protocol with $\mathcal{W}$. Here, sn is is the random nonce sampled by $\mathcal{P}$ in the corresponding execution of `Register`. In this hybrid, $\mathcal{P}$ instead samples a random $\mathsf{sn}' \leftarrow_{\$} \{0,1\}^{\lambda}$ and computes $(\mathsf{bsm}_1, St) \leftarrow \mathsf{BS.U}_1(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sn}')$. Later, to check if event $\mathsf{bad}_{3,1}$ occurs, nonce sn' is also used instead of sn.

- $\mathcal{H}_6$: This is the ideal world simulation using simulator $\mathcal{S}$ as described above.

Claim D.13 $\mathcal{H}_0$ and $\mathcal{H}_1$ are indistinguishable.

Proof. The distinguishing advantage between $\mathcal{H}_0$ and $\mathcal{H}_1$ can be bound by the probability of bad_1. As nonces sn are sampled uniformly at random in $\{0,1\}^{\lambda}$, the probability of bad_1 is negligible. ☐

Claim D.14 $\mathcal{H}_1$ and $\mathcal{H}_2$ are indistinguishable, if RP is secure against malicious services.

Proof. We show the claim using intermediate hybrids $\mathcal{H}_{1,i}$ for $i \in \{0, \dots, Q\}$, where Q is the number of interactions between honest parties and $\mathcal{W}$ in sub-protocol `Register`. In hybrid $\mathcal{H}_{1,i}$, we apply the change described in $\mathcal{H}_2$ to the first i of these Q interactions. By definition we have that $\mathcal{H}_1 = \mathcal{H}_{1,0}$ and $\mathcal{H}_{1,Q} = \mathcal{H}_2$. Thus, it remains to show indistinguishability for $\mathcal{H}_{1,i-1}$ and $\mathcal{H}_{1,i}$ for $i \in [Q]$. Note that the distinguishing probability between $\mathcal{H}_{1,i-1}$ and $\mathcal{H}_{1,i}$ can be bounded by the probability that bad_2 occurs in the i-th interaction.

To bound this probability, we present a reduction against the security of RP against malicious services. The reduction simulates $\mathcal{H}_{1,i-1}$, except for the i-th interaction between honest parties and $\mathcal{W}$ in sub-protocol `Register`. This means that all except the i-th interaction are simulated honestly exactly as in $\mathcal{H}_{1,i-1}$. The i-th interaction is simulated as in $\mathcal{H}_{1,i-1}$, until it receives the promise message prom from $\mathcal{W}$. Then, it outputs $\bar{\mathsf{pk}}_{r,\mathcal{W}}, \mathsf{tx}_r, \mathsf{sn}, \mathsf{pk}_{\mathsf{BS}}$ and prom to its game.

It is clear that the reduction perfectly simulates $\mathcal{H}_{1,i-1}$. Also, the conditions defining bad_2 are exactly the winning conditions in the security game of RP. ☐

Claim D.15 $\mathcal{H}_2$ and $\mathcal{H}_3$ are indistinguishable, if EXC is secure against malicious sellers.

Proof. Again, we prove the claim using hybrids $\mathcal{H}_{2,i}$ for $i \in \{0, \dots, Q\}$, where Q is the number of interactions between honest parties and $\mathcal{W}$ in sub-protocol `AddPayment`. In hybrid $\mathcal{H}_{2,i}$, we apply the change described in $\mathcal{H}_3$ to the first i of these Q interactions. By definition we have that $\mathcal{H}_2 = \mathcal{H}_{2,0}$ and $\mathcal{H}_{2,Q} = \mathcal{H}_3$. It remains to bound the distinguishing advantage between $\mathcal{H}_{2,i-1}$ and $\mathcal{H}_{2,i}$ for $i \in [Q]$. This advantage is upper bounded by the probability that $\mathsf{bad}_{3,1}$ or $\mathsf{bad}_{3,2}$ occurs in the i-th of these interactions.

We bound this probability by giving a reduction against the security of EXC against malicious sellers. The reduction simulates $\mathcal{H}_{2,i-1}$, except for the i-th interaction between honest parties and $\mathcal{W}$ in sub-protocol `AddPayment`. This means that all except the i-th interaction are simulated honestly exactly as in $\mathcal{H}_{2,i-1}$. For the i-th interaction, the reduction first passes $\mathsf{pk}_{\mathsf{BS}}$ and sn to the security game. Then, it obtains a key $\bar{\mathsf{pk}}_{l,\mathcal{P}}$ and a message bsm_1 in return. It simulates the opening of a shared address $(\bar{\mathsf{pk}}_{l,\mathcal{P}}, \bar{\mathsf{pk}}_{l,\mathcal{W}})$, using the key that it got from the game. Then, it sends bsm_1 to $\mathcal{W}$ as in the protocol. If the reduction did not receive xm_1 from $\mathcal{W}$, it sets $\mathsf{xm}_1 := \perp$. This includes the case where a corrupted party already closed the shared address (cf. event $\mathsf{bad}_{3,2}$). Then, the reduction sends $\bar{\mathsf{pk}}_{l,\mathcal{W}}, \mathsf{tx}_l$, and xm_1 to the game, where tx_l is as in the protocol. It obtains xm_2 in return. If $\mathsf{xm}_2 \neq \perp$, it sends xm_2 to $\mathcal{W}$. Once a corrupted party (e.g. $\mathcal{W}$) closes the shared address $(\bar{\mathsf{pk}}_{l,\mathcal{P}}, \bar{\mathsf{pk}}_{l,\mathcal{W}})$ using signatures $(\sigma_{l,\mathcal{P}}, \sigma_{l,\mathcal{W}})$, the reduction returns tx_l and $\sigma_{l,\mathcal{P}}, \sigma_{l,\mathcal{W}}$ to the game.

Clearly, the reduction perfectly simulates execution $\mathcal{H}_{2,i-1}$. Also, by the definition of events $\mathsf{bad}_{3,1}$ and $\mathsf{bad}_{3,2}$, the security game of EXC outputs 1 if one of these events occurs in the i-th interaction. ☐

Claim D.16 $\mathcal{H}_3$ and $\mathcal{H}_4$ are indistinguishable, if BS has unique signatures.

Proof. Note that the difference between both hybrids is how the blind signature σ_{BS} that is input into algorithm Redeem is computed by honest parties. In both hybrids, σ_{BS} is a valid blind signature for nonce sn with respect to public key $\mathsf{pk}_{\mathsf{BS}}$. By the assumption that blind signatures are unique, these are therefore identical. Thus, the change is only conceptual, and the view of the corrupted parties does not change. ☐

Claim D.17 $\mathcal{H}_4$ and $\mathcal{H}_5$ are indistinguishable, if BS is weakly blind.

Proof. We show that the two hybrids are indistinguishable by presenting a sequence of hybrids $\mathcal{H}_{4,i}$ for $i \in \{0, \ldots, Q\}$, where Q denotes the number of interactions between honest parties $\mathcal{P}$ and the corrupted sweeper $\mathcal{W}$ in sub-protocol `AddPayment`. Concretely, hybrid $\mathcal{H}_{4,i}$ is as hybrid $\mathcal{H}_4$, but the change described in hybrid $\mathcal{H}_5$ is applied to the first i of such interactions.

To show that $\mathcal{H}_{4,i-1}$ and $\mathcal{H}_{4,i}$ are indistinguishable for all $i \in [Q]$, we give a reduction against the weak blindness of BS. Note that due to the previous change, we do not need the blind signature that is computed in `AddPayment` anymore. We only need to know if it is valid or not (cf. event $\mathsf{bad}_{3,1}$). The reduction simulates $\mathcal{H}_{4,i-1}$ as it is, except for the i-th interaction between honest parties and $\mathcal{W}$ in sub-protocol `AddPayment`. In this interaction, it samples $\mathsf{sn}' \leftarrow_\$ \{0,1\}^\lambda$ and outputs $\mathsf{pk}_{\mathsf{BS}}, \mathsf{m}_0 := \mathsf{sn}$ and $\mathsf{m}_1 := \mathsf{sn}'$ to its game. Here, sn denotes the nonce that is blindly signed in $\mathcal{H}_4$, which has been sent by the honest party to $\mathcal{W}$ in the corresponding interaction of `Register`. The game gives bsm_1 to the reduction. Then, the reduction continues the simulation of the `AddPayment` interaction as in $\mathcal{H}_4$, using this message bsm_1. When a corrupted party closes the shared address and event $\mathsf{bad}_{3,2}$ did not happen, the reduction extracts bsm_2 using algorithm Get. Then, the reduction outputs bsm_2 to its game, which returns a bit $v \in \{0,1\}$, indicating if a valid signature could be derived. If $v = 1$, the reduction sets $\mathsf{bad}_{3,1} := 1$ and aborts. Otherwise, it continues the execution. Finally, it outputs whatever the environment outputs.

It is easy to see that the reduction perfectly simulates hybrid $\mathcal{H}_{4,i-1}$ if it runs in the security game with $b = 0$, and it perfectly simulates hybrid $\mathcal{H}_{4,i}$ if it runs in the security game with $b = 1$. ☐

Claim D.18 $\mathcal{H}_5$ and $\mathcal{H}_6$ are indistinguishable.

Proof. Note that in the ideal world simulation, $\mathcal{S}$ simulates the execution in $\mathcal{H}_5$, except for the calls of $\mathcal{F}_s$ to $\mathcal{L}$. These calls are perfectly simulated by exactly the same calls that functionality $\mathcal{F}_{\mathsf{ux}}$ issues. Further, $\mathcal{S}$ does not know the party $\mathcal{P}$ that interacts with $\mathcal{W}$ in `AddPayment`. As the source of messages is the only dependency on $\mathcal{P}$ that remains in $\mathcal{H}_5$ (due to previous changes), the security of the anonymous channel implies indistinguishability. ☐

☐

E BLS Signatures and Blind Signatures

For completeness, we recall the BLS signature scheme [BLS01] and its blind version [Bol03]. We denote the signature scheme by $\mathsf{SIG} = (\mathsf{Gen}, \mathsf{SIG.Sig}, \mathsf{Ver})$ and the blind signature scheme by $\mathsf{BS} = (\mathsf{Gen}, \mathsf{BS.S}, \mathsf{BS.U}, \mathsf{Ver})$. Both schemes have the same key generation and verification algorithm and work over cyclic groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ of prime order p with generators $g_1 \in \mathbb{G}_1, g_2 \in \mathbb{G}_2$ and $g_T := e(g_1, g_2) \in \mathbb{G}_T$, where $e\colon \mathbb{G}_1 \times \mathbb{G}_2$ is a pairing. Also, they require a random oracle $\mathsf{H}\colon \{0,1\}^* \to \mathbb{G}_1$. Algorithm $\mathsf{Gen}(1^\lambda)$ first generates such parameters, then it samples a secret key $\mathsf{sk} \leftarrow\!\!\$\, \mathbb{Z}_p$, and defines the public key $\mathsf{pk} := g_2^{\mathsf{sk}}$. Then it returns $(\mathsf{pk}, \mathsf{sk})$. Signatures are computed via

$$\mathsf{SIG.Sig}(\mathsf{sk}, \mathsf{m}) = \mathsf{H}(\mathsf{m})^{\mathsf{sk}}.$$

Algorithm $\mathsf{Ver}(\mathsf{pk}, \mathsf{m}, \sigma)$ returns the evaluation of the verification equation

$$e(\sigma, g_2) = e(\mathsf{H}(\mathsf{m}), \mathsf{pk}).$$

To blindly sign messages, algorithm $\mathsf{BS.U}_1(\mathsf{pk}, \mathsf{m})$ samples a random $\alpha \leftarrow\!\!\$\, \mathbb{Z}_p^*$ and returns $St := \alpha$ and $\mathsf{bsm}_1 := \mathsf{H}(\mathsf{m})^\alpha$. Then, algorithm $\mathsf{BS.S}(\mathsf{sk}, \mathsf{bsm}_1)$ returns $\mathsf{bsm}_2 := \mathsf{bsm}_1^{\mathsf{sk}}$, and algorithm $\mathsf{BS.U}_2(St, \mathsf{bsm}_2)$ returns $\sigma := \mathsf{bsm}_2^{1/\alpha}$.

F Interpolation with Preprocessing

We sketch how to improve computation costs of interpolation in the exponent (i.e. algorithm $\mathsf{reconst}_{g,z}$), if multiple related instances have to be evaluated. First, we consider multiple evaluations of the same polynomial, then we look at multiple evaluations of the same position, but for different polynomials. For both scenarios, we manage to reduce the total cost for $O(\lambda)$ evaluations from $O(\lambda^3)$ operations to $O(\lambda^2)$ operations by using preprocessing.

Multiple Evaluations. Suppose we know all shares $(x_0, h_0), \dots, (x_\lambda, h_\lambda)$ and we have to evaluate the polynomial in the exponent at multiple positions. In other words, we have to evaluate the algorithm $\mathsf{reconst}_{g,z}((x_0, h_0), \dots, (x_\lambda, h_\lambda))$ for different z. In a preprocessing step independent of z we first compute a coefficient representation $a_{j,0}, \dots, a_{j,\lambda} \in \mathbb{Z}_p$ of the polynomials ℓ_j such that

$$\ell_j(X) = \sum_{i=0}^{\lambda} a_{j,i} X^i.$$

Then, for each $i \in \{0, \dots, \lambda\}$ we compute the group elements

$$C_i := \prod_{j=0}^{\lambda} h_j^{a_{j,i}}.$$

Now, once we know $z \in \mathbb{Z}_p$, we can obtain the result of $\mathsf{reconst}_{g,z}$ by

$$\prod_{i=0}^{\lambda} C_i^{z^i}.$$

Multiple Last Samples. Suppose we know λ shares, and we are allowed to do some preprocessing. This preprocessing is allowed to do $O(\lambda^2)$ operations. Then, once the $(\lambda + 1)$st share is known, it should be possible to compute the result of $\mathsf{reconst}_{g,z}$ using only $O(\lambda)$ operations.

For shares $(x_0, h_0), \dots, (x_{\lambda-1}, h_{\lambda-1})$, the preprocessing is as follows: For each $j \in \{0, \dots, \lambda - 1\}$, define the polynomial

$$\ell'_j(X) := \prod_{m \in \{0,\dots,\lambda-1\}, m \neq j} \frac{X - x_m}{x_j - x_m} \in \mathbb{Z}_p[X]$$

and compute the group element $Z_j := h_j^{\ell'_j(z)}$.

Then, assume that the last share is (x_λ, h_λ). The result can now be computed as

$$\left(\prod_{j=0}^{\lambda-1} Z_j^{\frac{z - x_\lambda}{x_j - x_\lambda}} \right) \cdot h_\lambda^{\ell_\lambda(z)},$$

where the polynomial ℓ_λ is defined as

$$\ell'_j(X) := \prod_{m \in \{0,\dots,\lambda\}, m \neq j} \frac{X - x_m}{x_\lambda - x_m} \in \mathbb{Z}_p[X].$$

G Script for Parameter Computation

Listing 1: Python Script to compute communication complexity of our protocol Sweep-UC for different instantiations. More explanation is given in Section 8.2.

```
#!/usr/bin/env python



################################################
#  Efficiency estimation script for Sweep-UC  #
################################################
from tabulate import tabulate


# parameter for cut and choose
ccpar = 128


#sizes for communication complexity
#for curve BLS12-381 and secp256k1 in bit
G1BLS = 48*8
G2BLS = 2*48*8
GTBLS = 12*48*8
ZpBLS = 256
GSchnorr = 33*8
ZpSchnorr = 32*8

#a 128 bit integer should suffice for the nonce
Sn = 128

##################buildingblocks################################


redeem_cc_bls = {
        "name":  "BLS",
        "sizepk":  G2BLS,
        "sizepromise":  G1BLS + 2*ccpar*G1BLS + 3*ZpBLS + G1BLS + ccpar*(G1BLS+G2BLS)
}

exchange_cc_bls = {
        "name":  "BLS",
        "sizexm1": 2*ccpar*G1BLS+2*ccpar*G2BLS+ccpar*G1BLS,
        "sizexm2": G1BLS,
}



redeem_cc_schnorr = {
        "name": "Schnorr",
        "sizepk": GSchnorr,
        "sizepromise":  2*ccpar*ZpSchnorr+GSchnorr+ZpSchnorr+ccpar*(G2BLS+GSchnorr)+ccpar*(G1BLS+ZpSchnorr)
}

exchange_cc_schnorr = {
        "name": "Schnorr",
        "sizexm1": GSchnorr+2*ccpar*G1BLS+ccpar*(G2BLS+GSchnorr)+ccpar*ZpSchnorr,
        "sizexm2": 2*ZpSchnorr,
}




#############################################protocol#################################


sweepucs = []

for redeem in [redeem_cc_bls,redeem_cc_schnorr]:
        for exchange in [exchange_cc_bls,exchange_cc_schnorr]:
                sweepucs.append({
                        "nameexchange": exchange["name"],
                        "nameredeem": redeem["name"],
                        "communicationleft": G1BLS+exchange["sizexm1"]+exchange["sizexm2"],
                        "communicationright": Sn+redeem["sizepk"]+redeem["sizepromise"],
                })




##############################################table#################################


data = [["Left", "Right", "Comm. Left", "Comm. Right", "Comm. Total"]]

for sweepuc in sweepucs:
        row = [sweepuc["nameexchange"],sweepuc["nameredeem"]]
        left = sweepuc["communicationleft"]
        right = sweepuc["communicationright"]
        total = left+right
        row.append('{:.2f}'.format(round(left/8000.0)))
        row.append('{:.2f}'.format(round(right/8000.0)))
        row.append('{:.2f}'.format(round(total/8000.0)))
        data.append(row)


print(tabulate(data,headers='firstrow',tablefmt='fancy_grid'))
print(tabulate(data,headers='firstrow'))
```

$\underline{\text{Buyer}(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sn}, \mathsf{pk}_b, \mathsf{pk}_s, \mathsf{sk}_b, \mathsf{tx})}$		$\underline{\text{Seller}(\mathsf{sk}_{\mathsf{BS}}, \mathsf{pk}_b, \mathsf{pk}_s, \mathsf{sk}_s, \mathsf{tx})}$
$(\mathsf{bsm}_1, St) \leftarrow \mathsf{BS.U}_1(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sn})$	$\xrightarrow{\mathsf{bsm}_1}$	
.... Exchange		....
$\mathsf{xpar} := (\mathsf{pk}_{\mathsf{BS}}, \mathsf{bsm}_1, \mathsf{pk}_b, \mathsf{pk}_s, \mathsf{tx})$		$\mathsf{xpar} := (\mathsf{pk}_{\mathsf{BS}}, \mathsf{bsm}_1, \mathsf{pk}_b, \mathsf{pk}_s, \mathsf{tx})$
	$\xleftarrow{\mathsf{xm}_1}$	$(\mathsf{xm}_1, St) \leftarrow \mathsf{Setup}(\mathsf{xpar}, \mathsf{sk}_{\mathsf{BS}}, \mathsf{sk}_s)$
$\mathsf{xm}_2 \leftarrow \mathsf{Buy}(\mathsf{xpar}, \mathsf{sk}_b, \mathsf{xm}_1)$	$\xrightarrow{\mathsf{xm}_2}$	$\sigma_b := \mathsf{Sell}(St, \mathsf{xm}_2)$
		$\sigma_s \leftarrow \mathsf{SIG.Sig}(\mathsf{sk}_s, \mathsf{tx})$
Learn (σ_b, σ_s)		**Publish** (σ_b, σ_s)
$\mathsf{bsm}_2 := \mathsf{Get}(\mathsf{xpar}, \mathsf{xm}_1, \mathsf{xm}_2, \sigma_b, \sigma_s)$		
.... End of Exchange		....
$\sigma_{\mathsf{BS}} \leftarrow \mathsf{BS.U}_2(St, \mathsf{bsm}_2)$		

Figure 9: Schematic Overview of an exchange protocol $\mathsf{EXC} = (\mathsf{Setup}, \mathsf{Buy}, \mathsf{Sell}, \mathsf{Get})$ for a signature scheme $\mathsf{SIG} = (\mathsf{SIG.Gen}, \mathsf{SIG.Sig}, \mathsf{SIG.Ver})$ and a blind signature scheme $\mathsf{BS} = (\mathsf{BS.Gen}, \mathsf{BS.S}, \mathsf{BS.U}, \mathsf{BS.Ver})$.

$\underline{\text{Service}(\mathsf{sk}_{\mathsf{BS}}, \mathsf{sk}_s, \mathsf{tx}, \mathsf{sn})}$		$\underline{\text{User}(\mathsf{pk}_{\mathsf{BS}}, \mathsf{pk}_s, \mathsf{tx}, \mathsf{sn})}$
$\mathsf{rpar} := (\mathsf{pk}_{\mathsf{BS}}, \mathsf{pk}_s, \mathsf{tx}, \mathsf{sn})$		$\mathsf{rpar} := (\mathsf{pk}_{\mathsf{BS}}, \mathsf{pk}_s, \mathsf{tx}, \mathsf{sn})$
$\mathsf{prom} \leftarrow \mathsf{Promise}(\mathsf{rpar}, \mathsf{sk}_{\mathsf{BS}}, \mathsf{sk}_s)$	$\xrightarrow{\mathsf{prom}}$	$b := \mathsf{VerPromise}(\mathsf{rpar}, \mathsf{prom})$
		if $b = 0$: **abort**
		Learn σ_{BS}
		$\sigma_s \leftarrow \mathsf{Redeem}(\mathsf{rpar}, \mathsf{prom}, \sigma_{\mathsf{BS}})$

Figure 10: Schematic overview of a redeem protocol $\mathsf{RP} = (\mathsf{Promise}, \mathsf{VerPromise}, \mathsf{Redeem})$ for a signature scheme $\mathsf{SIG} = (\mathsf{SIG.Gen}, \mathsf{SIG.Sig}, \mathsf{SIG.Ver})$ and a blind signature scheme $\mathsf{BS} = (\mathsf{BS.Gen}, \mathsf{BS.S}, \mathsf{BS.U}, \mathsf{BS.Ver})$.

$\underline{\mathcal{W}(\mathsf{sk}_{\mathsf{BS}}, \mathsf{sk}_{\mathcal{W}})}$		$\underline{\mathcal{P}_i(\mathsf{pk}_b, \mathsf{pk}_{\mathsf{BS}})}$
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 11: Overview of the sub-protocol `Register` of protocol Sweep-UC. The protocol is run between the sweeper $\mathcal{W}$ and a party $\mathcal{P}_i$.

$\underline{\mathcal{P}_i(\mathsf{sk}_a, \mathsf{pk}_{\mathsf{BS}})}$		$\underline{\mathcal{W}(\mathsf{sk}_{\mathsf{BS}}, \mathsf{pk}_{\mathsf{BS}})}$
$(\mathsf{bsm}_1, St) \leftarrow \mathsf{BS.U}_1(\mathsf{pk}_{\mathsf{BS}}, \mathsf{sn})$	$\xrightarrow{\mathsf{bsm}_1}$	
$\mathcal{F}_s.\mathtt{OpenSh}(T, \mathsf{pk}_a, \mathcal{W}, \mathsf{amt}, \mathsf{sk}_a)$		
Receive $(\bar{\mathsf{pk}}_{l,\mathcal{P}}, \bar{\mathsf{pk}}_{l,\mathcal{W}}, \bar{\mathsf{sk}}_{l,\mathcal{P}})$ **from** $\mathcal{F}_s$		**Receive** $(\bar{\mathsf{pk}}_{l,\mathcal{P}}, \bar{\mathsf{pk}}_{l,\mathcal{W}}, \bar{\mathsf{sk}}_{l,\mathcal{W}})$ **from** $\mathcal{F}_s$
$\mathsf{tx}_l := (\bar{\mathsf{pk}}_{l,\mathcal{P}}, \bar{\mathsf{pk}}_{l,\mathcal{W}}, \mathsf{pk}_{\mathcal{W}}, \mathsf{amt})$		$\mathsf{tx}_l := (\bar{\mathsf{pk}}_{l,\mathcal{P}}, \bar{\mathsf{pk}}_{l,\mathcal{W}}, \mathsf{pk}_{\mathcal{W}}, \mathsf{amt})$
$\mathsf{xpar} := (\mathsf{pk}_{\mathsf{BS}}, \mathsf{bsm}_1, \bar{\mathsf{pk}}_{l,\mathcal{P}}, \bar{\mathsf{pk}}_{l,\mathcal{W}}, \mathsf{tx}_l)$		$\mathsf{xpar} := (\mathsf{pk}_{\mathsf{BS}}, \mathsf{bsm}_1, \bar{\mathsf{pk}}_{l,\mathcal{P}}, \bar{\mathsf{pk}}_{l,\mathcal{W}}, \mathsf{tx}_l)$
	$\xleftarrow{\mathsf{xm}_1}$	$(\mathsf{xm}_1, St) \leftarrow \mathsf{Setup}(\mathsf{xpar}, \mathsf{sk}_{\mathsf{BS}}, \bar{\mathsf{sk}}_{l,\mathcal{W}})$
$\mathsf{xm}_2 \leftarrow \mathsf{Buy}(\mathsf{xpar}, \bar{\mathsf{sk}}_{l,\mathcal{P}}, \mathsf{xm}_1)$	$\xrightarrow{\mathsf{xm}_2}$	
		$\sigma_{l,\mathcal{P}} := \mathsf{Sell}(St, \mathsf{xm}_2)$
		$\sigma_{l,\mathcal{W}} \leftarrow \mathsf{SIG.Sig}(\bar{\mathsf{sk}}_{l,\mathcal{W}}, \mathsf{tx}_l)$
Receive $(\sigma_{l,\mathcal{P}}, \sigma_{l,\mathcal{W}})$ **from** $\mathcal{F}_s$		$\mathcal{F}_s.\mathtt{CloseSh}(\bar{\mathsf{pk}}_{l,\mathcal{P}}, \bar{\mathsf{pk}}_{l,\mathcal{W}}, \mathsf{pk}_{\mathcal{W}}, \mathsf{amt}, \sigma_{l,\mathcal{P}}, \sigma_{l,\mathcal{W}})$
$\mathsf{bsm}_2 := \mathsf{Get}(\mathsf{xpar}, \mathsf{xm}_1, \mathsf{xm}_2, \sigma_{l,\mathcal{P}}, \sigma_{l,\mathcal{W}})$		
$\sigma_{\mathsf{BS}} \leftarrow \mathsf{BS.U}_2(St, \mathsf{bsm}_2)$		

Figure 12: Overview of the sub-protocol `AddPayment` of protocol Sweep-UC. The protocol is run between the sweeper $\mathcal{W}$ and a party $\mathcal{P}_i$.

$\underline{\mathcal{W}(\mathsf{sk}_{\mathsf{BS}}, \mathsf{pk}_{\mathsf{BS}})}$	$\underline{\mathcal{P}_i(\mathsf{pk}_b, \mathsf{pk}_{\mathsf{BS}})}$
	$\sigma_{r,\mathcal{W}} \leftarrow \mathsf{Redeem}(\mathsf{rpar}, \mathsf{prom}, \sigma_{\mathsf{BS}})$
	$\sigma_{r,\mathcal{P}} \leftarrow \mathsf{SIG.Sig}(\bar{\mathsf{sk}}_{r,\mathcal{P}}, \mathsf{tx}_r)$
Receive $(\sigma_{r,\mathcal{W}}, \sigma_{r,\mathcal{P}})$ **from** $\mathcal{F}_s$	$\mathcal{F}_s.\mathtt{CloseSh}(\bar{\mathsf{pk}}_{r,\mathcal{W}}, \bar{\mathsf{pk}}_{r,\mathcal{P}}, \mathsf{pk}_b, \mathsf{amt}, \sigma_{r,\mathcal{W}}, \sigma_{r,\mathcal{P}})$
$\mathsf{Reg} := \mathsf{Reg} \setminus \mathsf{pk}_b$	

Figure 13: Overview of the sub-protocol `GetPayment` of protocol Sweep-UC. The protocol is run between the sweeper $\mathcal{W}$ and a party $\mathcal{P}_i$.

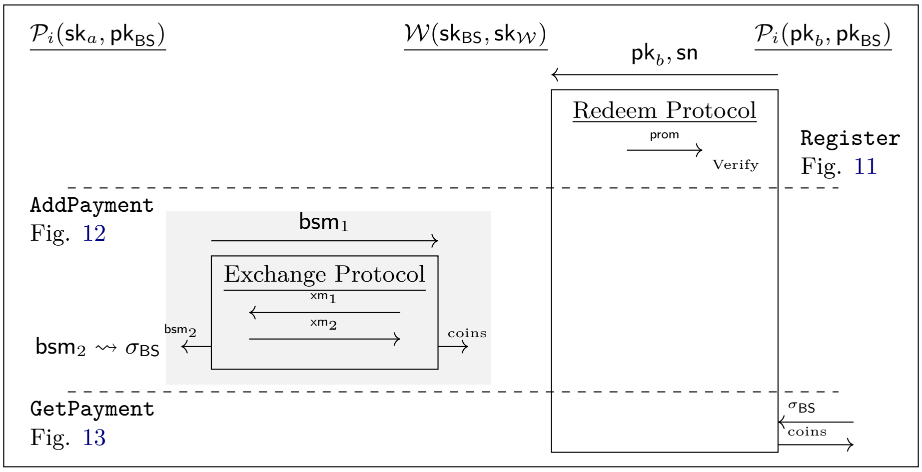

Figure 14: Overview of Sweep-UC, run between the sweeper $\mathcal{W}$ and a party $\mathcal{P}_i$. The gray area stands for an anonymous channel.

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 $\texttt{Register}(\mathsf{pk}_b)$, 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 $\mathcal{L}^{\mathsf{SIG}}.\texttt{Freeze}(\mathsf{pk}_{\mathcal{W}}, \mathsf{amt})$ 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 $\mathcal{L}^{\mathsf{SIG}}.\texttt{Unfreeze}(\mathsf{pk}_{\mathcal{W}}, \mathsf{amt})$ and delete the entry from Reg.

Interface $\texttt{AddPayment}(\mathsf{pk}_a, \mathsf{sk}_a, \mathsf{pk}_b)$, 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 $(\mathsf{pk}_a, \mathsf{sk}_a) \notin \mathsf{SIG}.\mathsf{Gen}(1^\lambda)$, send "failInvalidKey" and return.
03 Send ("addPayment", pk_a) to $\mathcal{S}$.
04 Call $\mathcal{L}^{\mathsf{SIG}}.\texttt{Freeze}(\mathsf{pk}_a, \mathsf{amt})$ 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 $\mathcal{L}^{\mathsf{SIG}}.\texttt{Unfreeze}(\mathsf{pk}_a, \mathsf{amt})$, send "fail" and return.
09 Call $\mathcal{L}^{\mathsf{SIG}}.\texttt{Unfreeze}(\mathsf{pk}_{\mathcal{W}}, \mathsf{amt})$.
10 Append $(\mathcal{P}_i, \mathsf{pk}_a, \mathsf{pk}_b)$ to Pay.

Interface $\texttt{ChangePayment}(\mathsf{pk}_a, \mathsf{pk}_b, \mathsf{pk}_c)$, 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 $\texttt{GetPayment}(\mathsf{pk}_b)$, 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 $\mathcal{L}^{\mathsf{SIG}}.\texttt{Unfreeze}(\mathsf{pk}_b, \mathsf{amt})$.

Figure 15: Ideal functionality $\mathcal{F}_{\mathsf{ux}}$ that models an unlinkable exchange.

Functionality $\mathcal{L}^{\mathsf{SIG}}$

The global functionality interacts with parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$, the environment $\mathcal{Z}$, and ideal adversary $\mathcal{S}$. It is parameterized by a digital signature scheme $\mathsf{SIG} = (\mathsf{Gen}, \mathsf{Sig}, \mathsf{Ver})$. The functionality holds a list $\mathsf{FrozenCoins}$, and a key value table bal. The table bal is publicly accessible to every party.

Interface $\texttt{Update}(\mathsf{pk}, c)$, called by $\mathcal{Z}$:

01 Set $\mathsf{bal}[\mathsf{pk}] := c$.
02 Send ("updatedFunds", pk, c) to every entity.

Interface $\texttt{Pay}(\mathsf{pk}_s, \mathsf{pk}_r, c, \mathsf{sk}_s)$, called by $\mathcal{P}_i$:

01 If $c > \mathsf{bal}[\mathsf{pk}_s]$, send "failNoFunds" and return.
02 If $(\mathsf{pk}_s, \mathsf{sk}_s) \notin \mathsf{SIG.Gen}(1^\lambda)$, send "failInvalidKey" and return.
03 Set $\mathsf{bal}[\mathsf{pk}_s] := \mathsf{bal}[\mathsf{pk}_s] - c, \mathsf{bal}[\mathsf{pk}_r] := \mathsf{bal}[\mathsf{pk}_r] + c$, and $ctr := ctr + 1$.
04 Send ("payed", $\mathsf{pk}_s, \mathsf{pk}_r, c$) to every party.

Interface $\texttt{Freeze}(\mathsf{pk}, c)$, called by an ideal functionality with identifier id:

01 If $c > \mathsf{bal}[\mathsf{pk}]$, send "failNoFunds" and return.
02 Else set $\mathsf{bal}[\mathsf{pk}] := \mathsf{bal}[\mathsf{pk}] - c$ and append (id, c) to $\mathsf{FrozenCoins}$.
03 Send ("frozen", id, pk, c) to every entity.

Interface $\texttt{Unfreeze}(\mathsf{pk}, c)$, called by an ideal functionality with identifier id:

01 If there is no entry (id, c') such that $c' \geq c$ in $\mathsf{FrozenCoins}$, then send "failNoFrozenFunds" and return.
02 Else replace (id, c') in $\mathsf{FrozenCoins}$ with $(id, c' - c)$.
03 If $c' = c$, remove the entry from $\mathsf{FrozenCoins}$.
04 Set $\mathsf{bal}[\mathsf{pk}] := \mathsf{bal}[\mathsf{pk}] + c$.
05 Send ("unfrozen", id, pk, c) to every entity.

Figure 16: Global ideal functionality $\mathcal{L}^{\mathsf{SIG}}$ that models a ledger.

Functionality $\mathcal{F}_s$

The functionality interacts with the functionality $\mathcal{L}^{\mathsf{SIG}}$, parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$, the environment $\mathcal{Z}$, and ideal adversary $\mathcal{S}$.

Interface $\texttt{OpenSh}(T, \mathsf{pk}_{in}, \mathcal{P}_b, c, \mathsf{sk}_{in})$, called by $\mathcal{P}_a$:

01 If $(\mathsf{pk}_{in}, \mathsf{sk}_{in}) \notin \mathsf{SIG.Gen}(1^\lambda)$, send "failInvalidKey" and return.
02 Generate keys $(\mathsf{pk}_a, \mathsf{sk}_a) \leftarrow \mathsf{SIG.Gen}(1^\lambda), (\mathsf{pk}_b, \mathsf{sk}_b) \leftarrow \mathsf{SIG.Gen}(1^\lambda)$.
03 Call the interface $\mathcal{L}^{\mathsf{SIG}}.\texttt{Freeze}(\mathsf{pk}_{in}, c)$. If it replies with "failNoFunds", reply with "failNoFunds" and return. Else, append $(\mathsf{pk}_a, \mathsf{pk}_b, T, \mathcal{P}_a, \mathcal{P}_b, c)$ to $\mathsf{OpenShared}$.
04 After T clock cycles: If this entry $(\mathsf{pk}_a, \mathsf{pk}_b, T, \mathcal{P}_a, \mathcal{P}_b, c)$ is still in $\mathsf{OpenShared}$, then invoke the interface $\mathcal{L}^{\mathsf{SIG}}.\texttt{Unfreeze}(\mathsf{pk}_{in}, c)$ and delete the entry from $\mathsf{OpenShared}$.
05 Send $(\mathsf{pk}_a, \mathsf{pk}_b, \mathsf{sk}_a)$ to $\mathcal{P}_a$ and $(\mathsf{pk}_a, \mathsf{pk}_b, \mathsf{sk}_b)$ to $\mathcal{P}_b$.
06 Send ("openedSharedAddress", $\mathsf{pk}_a, \mathsf{pk}_b, \mathsf{pk}_{in}, c$) to every party.

Interface $\texttt{CloseSh}(\mathsf{pk}_a, \mathsf{pk}_b, \mathsf{pk}_{out}, c, \sigma_a, \sigma_b)$, called by $\mathcal{P}_b$:

01 If there is no entry of the form $(\mathsf{pk}_a, \mathsf{pk}_b, T, \mathcal{P}_a, \mathcal{P}_b, c)$ in the list $\mathsf{OpenShared}$, send "failNoOpenSharedAddress" and return.
02 Let $\mathsf{tx} := (\mathsf{pk}_a, \mathsf{pk}_b, \mathsf{pk}_{out}, c)$.
03 Set $b_a := \mathsf{SIG.Ver}(\mathsf{pk}_a, \mathsf{tx}, \sigma_a)$ and $b_b := \mathsf{SIG.Ver}(\mathsf{pk}_b, \mathsf{tx}, \sigma_b)$.
04 If $b_a = 0$ or $b_b = 0$, then reply with "failInvalidSignature" and return.
05 Call the interface $\mathcal{L}^{\mathsf{SIG}}.\texttt{Unfreeze}(\mathsf{pk}_{out}, c)$ and remove the entry $(\mathsf{pk}_a, \mathsf{pk}_b, T, \mathcal{P}_a, \mathcal{P}_b, c)$ from $\mathsf{OpenShared}$.
06 Send ("closedSharedAddress", $\mathsf{pk}_a, \mathsf{pk}_b, \mathsf{pk}_{out}, c, \sigma_a, \sigma_b$) to every party.

Figure 17: Ideal functionality $\mathcal{F}_s$ that models a the opening and closing of a shared address for a ledger functionality $\mathcal{L}^{\mathsf{SIG}}$.