

# Sweep-UC: Swapping Coins Privately

Lucjan Hanzlik<sup>1</sup>, Julian Loss<sup>1</sup>, Sri AravindaKrishnan Thyagarajan<sup>2</sup>, and  
Benedikt Wagner<sup>1,3</sup>

<sup>1</sup> CISA Helmholtz Center for Information Security  
`{hanzlik,loss,benedikt.wagner}@cispa.de`

<sup>2</sup> NTT Research

`t.srikrishnan@gmail.com`

<sup>3</sup> 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*<sup>4</sup>, 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 Introduction

One of the most fundamental financial operations is the exchange of 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. Clearly, 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

---

<sup>4</sup> Read as *Sweep Ur Coins*.

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  $B$ ) [26,3,2,28,4,7,9,11]. 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 [3,2,28] use a special type of script called *Hash Timelock Contract* (HTLC). Roughly speaking, Alice can use an HTLC script with hash function  $H$  to freeze some amount of her coins temporarily as follows. The HTLC specifies a value  $h$  such that if Bob presents  $x$  with  $H(x) = h$ , Bob obtains Alice’s coins. On the other hand, the HTLC 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 [11], or smart contracts [26,4,7,9] such as 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 scripts or contracts, e.g., Monero [27].
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 transaction 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 [8].

**Existing Constructions.** To overcome these issues, Thyagarajan, Malavolta and Moreno-Sanchez proposed *universal swaps* [39]. Their protocol enables 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 [20]. This strongly limits the applicability to important systems including Monero or the Chia network [5]. Due to the result of Erwig et al. [20], Chia (and any other system based on unique signatures) *provably* lacks support for adaptor signatures.

Tumblebit [24] and A<sup>2</sup>L [37] are two efficient atomic swap protocols that take an alternate route. In these protocols, Alice changes her coins with an *untrusted intermediate party*, a *tumbler* (in case of Tumblebit), or a *hub* (in case of A<sup>2</sup>L). While the intermediary can deny its service to Alice, atomicity of the exchange

between Alice and the intermediary is guaranteed. Specifically, Alice can make a payment of a coin in currency  $A$  to the intermediary, and in return is guaranteed to get a payment of a coin in currency  $B$  from the intermediary. Relying on such an intermediary has many benefits. For example, Alice no longer has to solve the *bootstrapping problem* [3,2,28,26,4,7,9], 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 of type  $A$  and  $B$  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 [30,35,31,32,29] have shown that mere pseudonyms do not guarantee privacy or anonymity for the users and their coins. Many instances [6] have showcased the importance of privacy and anonymity of coins and there has been considerable effort like CoinJoin [1], CoinShuffle [33,34], among many others to improve coin privacy. Even new currencies with enhanced privacy were developed from scratch [27,10].

Unfortunately, Tumblebit critically relies on the support of HTLC scripts from the underlying blockchains, which results in poor compatibility (with systems like Monero). Usage 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 [22] that there was a gap in their security model which allowed for key recovery attacks on specific instantiations. The authors of [22] also proposed fixes to  $A^2L$  called  $A^2L^+$ , but only prove security in an idealized model (the linear-only encryption model) [23] with game-based security guarantees. They also propose a version called  $A^2L^{UC}$  in the *Universal Composability (UC)* framework [16], that unfortunately requires heavy cryptographic tools like general-purpose two party computation (2PC). This makes the protocol inefficient for immediate use. Moreover, both  $A^2L^+$  and  $A^2L^{UC}$  do not offer compatibility with systems lacking adaptor signature support. We summarize existing solutions in Table 1.

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

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

### 1.1 Our Contribution

We answer the above question positively by presenting Sweep-UC. Like Tumblebit and  $A^2L$  (series), Sweep-UC is bootstrapped with an intermediary called the

Protocol	Scripts	Signature	UC	Comments
Tumblebit [24]	HTLC	ECDSA	(✓)	Security only for parts
A <sup>2</sup> L [37]	Signature verification	Adaptor	✗	Gap in security model
A <sup>2</sup> L <sup>+</sup> [22]	Signature verification	Adaptor	✗	Idealized model
A <sup>2</sup> L <sup>UC</sup> [22]	Signature verification	Adaptor	✓	General-purpose 2PC
Sweep-UC	Signature verification	Adaptor or BLS	✓	

**Table 1.** Comparison of our protocol Sweep-UC with previous protocols. We compare the required scripting functionality and the supported signature schemes, as well as the security that is proven. All protocols additionally require a timelock script, which can be removed using tools from [38].

*sweeper* and can be used to swap (i.e. exchange) coins unlinkably and atomically. We compare our protocol with existing solutions in Table 1. Below, we summarize the properties of our protocol.

**Efficiency and Security.** Sweep-UC achieves the strong notion of UC security. At the same time, in contrast to [39,22], it does not rely on any heavy cryptographic machinery such as general-purpose 2PC. In particular, we thereby solve the challenge raised in [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.

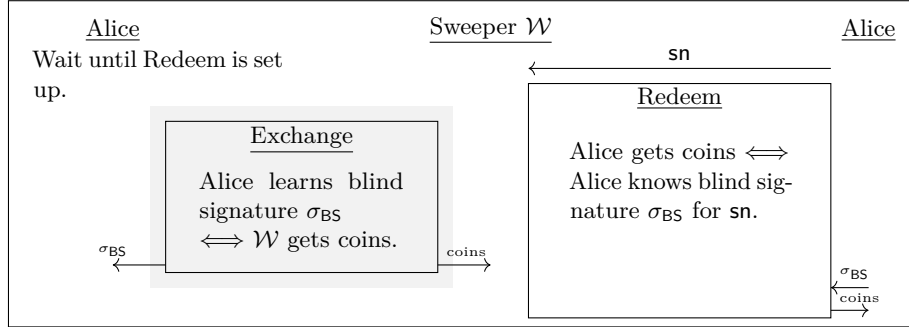
**Compatibility.** To support swaps between currencies  $A$  and  $B$ , Sweep-UC relies only on minimal scripting for verifying signatures<sup>5</sup>. As discussed, this preserves on-chain privacy and fungibility of the currencies involved. In terms of supported signature schemes, Sweep-UC is the first 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 [20], and BLS [14]<sup>6</sup>. 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

<sup>5</sup> Similar to Tumblebit, A<sup>2</sup>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 [38].

<sup>6</sup> 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.

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 the efficiency, one only has to focus on the construction of these game-based sub-protocols, instead of doing a cumbersome UC proof again.



**Fig. 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*  $\mathcal{W}$ <sup>7</sup>. This should be done in an atomic and unlinkable way.

**Blueprint.** Assume Alice owns addresses  $\mathbf{pk}_{in}$  and  $\mathbf{pk}_{out}$  and the sweeper owns  $\mathbf{pk}_{\mathcal{W}}$ . Our goal is to coordinate two payments  $\mathbf{tx}_{out} = \mathbf{pk}_{a,out} \rightarrow \mathbf{pk}_{\mathcal{W}}$  in currency  $A$  and  $\mathbf{tx}_{in} = \mathbf{pk}_{\mathcal{W}} \rightarrow \mathbf{pk}_{a,in}$  in currency  $B$ . To coordinate these payments, Sweep-UC implements a form of Chaum’s E-Cash [17], which is also the common high level structure of previous protocols [24,37,22,25]. In this E-Cash approach, Alice signs  $\mathbf{tx}_{out}$  using her secret key  $\mathbf{sk}_{a,out}$  (associated to  $\mathbf{pk}_{a,out}$ ) and obtains some voucher in exchange. Then, Alice can use that voucher to get a signature (valid with respect to  $\mathbf{pk}_{\mathcal{W}}$ ) for  $\mathbf{tx}_{in}$ . Let us now explain the steps of Sweep-UC in a bit more detail. An overview can be found in Figure 1. We assume that

<sup>7</sup>  $\mathcal{S}$  is reserved for the simulator in the UC-proof.

the sweeper holds the secret key  $sk_{BS}$  for a blind signature scheme  $BS$ , and the corresponding public key  $pk_{BS}$  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  $\sigma_{BS}$  for  $sn$  in the future, she will be able to learn a signature for transaction  $tx_{in}$ . She could then publish this signature, to get coins from  $\mathcal{W}$ . We can ensure that  $\mathcal{W}$  can not spend these coins in the meantime, by locking them in a shared address for a certain time, which is a standard technique<sup>8</sup>.

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 exchange, the signed payment  $tx_{out}$  is published, i.e.  $\mathcal{W}$  gets coins. To ensure fairness, we wrap what we call an *exchange protocol* around the blind signature interaction. Finally (right-hand side), Alice uses the received blind signature  $\sigma_{BS}$  on  $sn$  in the redeem protocol to get a signature on payment  $tx_{in}$ , and publishes the signed payment. One of the major design challenges to be overcome is to set up both the left and the right-hand side in a compatible way. We will come back to the required security guarantees for the exchange and redeem protocols later.

## 2.1 Challenge 1: UC Modeling

Before we start thinking about a UC proof, we need to define an appropriate ideal functionality  $\mathcal{F}_{ux}$ . Our first attempt to do this is to have three interfaces, covering the three phases as above. I.e. we have interfaces where the user can (1) register a receiving key  $pk_{in}$  (right-hand side), (2) add a payment (left-hand side) by specifying  $pk_{out}$  and referring to the registered  $pk_{in}$ , and (3) get the payment for  $pk_{in}$  (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 security of our protocol. This problem, as discussed extensively in [22], arises from the blindness of blind signatures. It is the reason why the UC proof of A<sup>2</sup>L [37] 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  $pk_{in}$ . If blindness of  $BS$  is unconditional, the simulator can not do that, as it can not extract the correct  $pk_{in}$ . For the case of computational blindness, we refer the reader to [22] for a detailed discussion. Namely, based on the common structure of known blind signature schemes, the authors of [22] elaborate that there is only little hope to get a secure system if blindness is computational. The solution taken in [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

<sup>8</sup> This is why our protocol, as well as previous protocols, requires timelocks.

a blind signature. In other words, we cannot rule out that Alice changes the receiving public key  $\text{pk}_{\text{in}}$  after obtaining the blind signature on the left. At the same time, there is no reason why we want to rule this out. Namely, even if Alice changes  $\text{pk}_{\text{in}}$  to  $\text{pk}'_{\text{in}}$  afterwards, this does steal coins from the sweeper, as long as she can not redeem coins (interface (3)) for *both*  $\text{pk}_{\text{in}}$  and  $\text{pk}'_{\text{in}}$ . With this in mind, we add an additional interface `ChangePayment`, that allows the simulator to change  $\text{pk}_{\text{in}}$  to  $\text{pk}'_{\text{in}}$  in case Alice is corrupted and both  $\text{pk}_{\text{in}}, \text{pk}'_{\text{in}}$  have been registered before. Note that the number of coins that the sweeper spends in total stays the same, and 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  $\text{pk}_{\text{in}}$ , and call `ChangePayment` with the correct  $\text{pk}'_{\text{in}}$  afterwards, once it learns  $\text{sn}$  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 [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 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.** Roughly, we want to avoid that  $\mathcal{W}$  loses coins. Intuitively, 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 received 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  $\ell$  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  $\ell$  coins on the left  $\implies \mathcal{R}$  uses signer oracle  $\ell$  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 by sweeper on the left and the number of times

reduction  $\mathcal{R}$  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 usages of the blind signature secret key  $\text{sk}_{\text{BS}}$  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  $\text{sk}_{\text{BS}}$  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  $\text{sk}_{\text{BS}}$ . Once we know the exchange protocol is completed, 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  $\text{sk}_{\text{BS}}$  have to be simulated without using  $\text{sk}_{\text{BS}}$  or any signer oracle.

Now, consider (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<sup>9</sup>, 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 technical contribution. We have constructions for both unique signatures as well as adaptor signatures. In both cases, the blind signature scheme  $\text{BS}$  is the BLS blind signature scheme. Note that  $\text{BS}$  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  $\text{bsm}_1 \in \mathbb{G}$  and  $\text{bsm}_2 = \text{bsm}_1^{\text{sk}_{\text{BS}}} \in \mathbb{G}$  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 of 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  $\sigma$  for transaction  $\text{tx}_{\text{in}} = \text{pk}_{\mathcal{W}} \rightarrow \text{pk}_{a,\text{in}}$  once it knows the blind signature  $\sigma_{\text{BS}}$ . 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  $\sigma$  into a ciphertext  $\text{ct}$  using  $\sigma_{\text{BS}}$  as a symmetric key. To convince Alice that she can really decrypt, i.e.  $\text{ct}$  is well-formed,  $\mathcal{W}$  could append a non-interactive zero-knowledge proof (NIZK)  $\pi$ . That is, to

<sup>9</sup> Otherwise, if a user relied on interaction with the sweeper in this step, then a malicious sweeper could go offline and violate fairness.



set up the redeem protocol,  $\mathcal{W}$  would send a “promise” message  $\text{prom} = (\text{ct}, \pi)$  to Alice. Once Alice verified this message (by verifying  $\pi$ ), Alice can be sure that she can decrypt  $\sigma$  using  $\sigma_{\text{BS}}$ , 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  $\text{ct}$  and  $\pi$  without having access to  $\text{sk}_{\text{BS}}$  or  $\sigma_{\text{BS}}$ . The challenge here is that once the user knows  $\sigma_{\text{BS}}$  (e.g. because it behaves honestly), the ciphertext  $\text{ct}$  should look consistent again. To implement this, we define  $\text{ct} := \text{H}(\sigma_{\text{BS}}) \oplus \sigma$  instead, and use the programmability of the random oracle  $\text{H}$ . Namely, we send a simulated  $\pi$  and a random  $\text{ct}$  first, and program  $\text{H}(\sigma_{\text{BS}}) := \text{ct} \oplus \sigma$  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  $\text{tx}_{\text{out}}$  requires two signatures  $\sigma_{\mathcal{W}}$  and  $\sigma_a$  by  $\mathcal{W}$  and Alice, respectively<sup>10</sup>. We encrypt the blind signature response  $\text{bsm}_2$  using transaction signature  $\sigma_{\mathcal{W}}$  for transaction  $\text{tx}_a$  in the same way, i.e.  $\text{ct} := \text{H}(\sigma_{\mathcal{W}}) \oplus \text{bsm}_2$ . When Alice receives  $\text{ct}$  and a NIZK  $\pi$ , she sends her share  $\sigma_a$  if  $\pi$  verifies. Then, once  $\mathcal{W}$  publishes  $\sigma_{\mathcal{W}}, \sigma_a$ , Alice derives  $\text{bsm}_2$  from  $\text{ct}$ .

The constructions sketched here have a significant shortcoming: We use NIZKs to prove relations defined by random oracle  $\text{H}$ . 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 by suitable modifications and switching roles as in our naive attempt. We also omit some minor details for readability. Roughly, our idea is to use a cut-and-choose technique to implement the proof  $\pi$ . In such a technique,  $\mathcal{W}$  would repeat the naive attempt in  $2\lambda$  instances independently, and has to open  $\lambda$  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  $\sigma_{\text{BS}}$ . A first try to solve this problem using secret sharing is as follows:

1.  $\mathcal{W}$  sends a ciphertext  $\text{ct}_0 = h^{f'(0)} \cdot \sigma$ , and ciphertexts  $\text{ct}_j = \text{H}(\sigma_{\text{BS}}, j) \oplus h^{f'(j)}$ ,  $j \in [2\lambda]$ , where  $h$  is a generator of  $\mathbb{G}$ , and  $f'$  is a random polynomial of degree  $\lambda$  over  $\mathbb{Z}_p$ .  $\mathcal{W}$  also commits to  $f'$  by sending its coefficients in the exponent of a generator.  $\mathcal{W}$  can prove well-formedness of  $\text{ct}_0$  using an efficient NIZK, as the statement is purely algebraic<sup>11</sup>.
2. Alice challenges  $\mathcal{W}$  to open  $\text{ct}_k$  by sending  $\sigma_{\text{BS}}$  and  $f'(k)$  for  $\lambda$  randomly chosen  $k$ <sup>12</sup>.
3. Using the opening, Alice can check consistency by recomputing  $\text{ct}_k$  for all opened  $k$ , and checking in the exponent that  $f'(k)$  indeed lies on the poly-

<sup>10</sup> This can be implemented using a multi-signature address.

<sup>11</sup> The relation is defined by  $h$ , the first committed coefficients of  $f'$ , and  $\text{pk}_{\mathcal{W}}$ .

<sup>12</sup> This can be done non-interactively using the Fiat-Shamir heuristic.

nomial  $f'$ . The cut-and-choose technique guarantees that with overwhelming probability, at least one unopened ciphertext  $\text{ct}_{k^*}$  is also consistent.

4. To redeem, once Alice learns  $\sigma_{\text{BS}}$  from the exchange on the left, she can decrypt  $\text{ct}_{k^*}$  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  $\text{ct}_0$ .

This approach allows the user to check consistency without requiring the NIZK  $\pi$ . 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  $\text{ct}_k$  by sending  $\sigma_{\text{BS}}$  and  $f'(k)$ , the user learns  $\sigma_{\text{BS}}$ , and can therefore redeem its coins without interacting on the left. On a technical level, simulating the promise without knowing  $\sigma_{\text{BS}}$  will fail.

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

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

Then, each ciphertext has the form  $\text{ct}_j = \text{H}(\sigma_j) \oplus h^{f'(j)}$ , and can be opened by sending  $\sigma_j$ . Again, we publish coefficients of  $f$  in the exponent, which allows to publicly compute  $\text{pk}_{\text{BS},j}$ . Now, the user can check consistency of  $\sigma_j$  using  $\text{pk}_{\text{BS},j}$  and BLS verification. Also, note that Alice (computationally) only learns  $\lambda$  points of  $f$  in the exponent of basis  $\text{H}(\text{sn})$ . Once Alice bought the blind signature  $\sigma_{\text{BS}}$  on the left, this serves as the  $(\lambda + 1)$ st share, and she can reconstruct  $f$  in the exponent of basis  $\text{H}(\text{sn})$ , i.e. she learns all  $\sigma_j$ . Then, the argument is as before. Namely, soundness of the cut-and-choose guarantees that there is at least one unopened  $k^*$  for which  $\text{ct}_{k^*}$  is consistent. As in the strawman solution, she can now compute  $h^{f'(0)}$  and therefore  $\sigma$  from  $\text{ct}_0$ . It turns out that, implemented carefully, the random oracle-based simulation strategy works out now. Our simulator can know which indices are opened in advance. Then, without knowing  $\text{sk}_{\text{BS}}$  and  $\sigma_{\text{BS}}$ , it can define the polynomial  $f$  such that  $f(0) = \text{sk}_{\text{BS}}$  implicitly in the exponent, while still knowing  $\lambda$  points of  $f$  over  $\mathbb{Z}_p$ . These can be used to open consistently, and the unopened  $\text{ct}_j$  are sampled at random as in the naive attempt. Then, once Alice queries  $\text{H}(\sigma_j)$  for some unopened  $\sigma_j$ , the simulator can compute  $f$  entirely in the exponent of basis  $\text{H}(\text{tx})$ , and program  $\text{H}(\sigma_j) = \text{ct}_j \oplus h^{f'(j)}$  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  $\rho$  explicit. The notation  $y \in \mathcal{A}(x)$  means that  $y$  is a possible output of  $\mathcal{A}(x)$ . A function  $f : \mathbb{N} \rightarrow \mathbb{R}_+$  is said to be negligible in its input  $\lambda$ , if  $f \in \lambda^{-\omega(1)}$ . The first  $K$  natural numbers are denoted by  $[K] := \{1, \dots, K\}$ . Next, we introduce the cryptographic primitives we use. For formal definitions of the primitives and computational assumptions we refer to Supplementary Material A.

**Digital Signatures.** A signature scheme  $\text{SIG} = (\text{Gen}, \text{Sig}, \text{Ver})$  consists of three PPT algorithms. The key generation algorithm  $\text{Gen}(1^\lambda)$  generates a key pair  $(\text{pk}, \text{sk})$ . We require the public keys  $\text{pk}$  generated by  $\text{Gen}$  to have high entropy. The signing algorithm  $\text{Sig}(\text{sk}, \text{m})$  generates a signature  $\sigma$  on the message  $\text{m}$ . The verification algorithm  $\text{Ver}(\text{pk}, \text{m}, \sigma)$  validates the signature  $\sigma$  with respect to message  $\text{m}$  and public key  $\text{pk}$  and returns either 1 for valid, or 0 for invalid. A signature scheme is said to be *unique* if for any public key  $\text{pk}$  and message  $\text{m}$ , there exists exactly one  $\sigma$  with  $\text{Ver}(\text{pk}, \text{m}, \sigma) = 1$ . The security property of interest is that of *unforgeability*. Here, an adversary without access to the secret key  $\text{sk}$ , should not be able to forge a fresh valid signature on a message even given access to signatures on any arbitrary messages of its choice. Such an unforgeable signature scheme is referred to as being EUF-CMA secure. 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 [17] 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  $\text{BS} = (\text{Gen}, \text{S}, \text{U}, \text{Ver})$ , where  $\text{Gen}$  and  $\text{Ver}$  are as before. Signatures are generated in an interactive protocol between a user  $\text{U}(\text{pk}, \text{m})$  and a signer  $\text{S}(\text{sk})$ . We only consider two-move blind signature schemes, for which the interaction is as follows:  $(\text{bsm}_1, \text{St}) \leftarrow \text{U}_1(\text{pk}, \text{m})$ ,  $\text{bsm}_2 \leftarrow \text{S}(\text{sk}, \text{bsm}_1)$ ,  $\sigma \leftarrow \text{U}_2(\text{St}, \text{bsm}_2)$ . We write  $\sigma \leftarrow \text{BS.Sig}(\text{sk}, \text{m})$  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  $\text{m}$  and the resulting signature  $\sigma$ . For this work, we only need a relaxed version of this property referred to as *weak blindness* where the adversary is not given  $\sigma$ , but only if  $\sigma$  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  $\ell$  interactions with the signer.

**Threshold Secret Sharing.** We make use of Shamir secret sharing [36] 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  $\text{reconst}_p((x_0, y_0), \dots, (x_\lambda, y_\lambda))$  and  $\text{reconst}_{g,z}((x_0, h_0), \dots, (x_\lambda, h_\lambda))$  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  $\ell_j(X) := \prod_{m \in \{0, \dots, \lambda\}, m \neq j} (X - x_m) / (x_j - x_m) \in \mathbb{Z}_p[X]$ . Algorithm  $\text{reconst}_p$  outputs  $L(X) := \sum_{j=0}^\lambda y_j \cdot \ell_j(X) \in \mathbb{Z}_p[X]$ , and  $\text{reconst}_{g,z}$  outputs  $\prod_{j=0}^\lambda h_j^{\ell_j(z)}$ . Further, given  $\lambda$

indices  $(k_j)_{j \in [\lambda]}$  for  $k_j \in [2\lambda]$ , we define algorithm  $\text{polyGen}_{g,p}(\lambda, \text{coeff}_0, (k_j)_{j \in [\lambda]})$  that internally generates a polynomial  $f(X) \in \mathbb{Z}_p[X]$  of degree  $\lambda$  and outputs  $\lambda$  evaluations  $((k_j, s_{k_j} := f(k_j))_{j \in [\lambda]})$  and  $\lambda$  coefficients  $(\text{coeff}_j)_{j \in [\lambda]}$ . For the outputs we have  $g^{f(k_j)} = \prod_{i=0}^{\lambda} (\text{coeff}_i)^{(k_j)^i}$  for all  $j \in [\lambda]$  and  $g^{f(0)} = \text{coeff}_0$ .

## 4 Security Model

In this section, we first discuss the security properties that we want to achieve. Then, we introduce our formal security model.

**Informal Security Properties.** We aim for three security properties that our protocol should satisfy. These are security for users, security for the sweeper, and unlinkability. Let us describe what these goals mean informally. Our protocol should achieve *security for users*, in a 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 payed back by the sweeper, even if for example the sweeper goes offline. On the other hand, our protocol should achieve *security for the sweeper*. This means that colluding users should only be able to get coins from the sweeper, if they payed before. Finally, we aim for *unlinkability*. This property means that if a lot of users interact with the sweeper at the same time, then the neither the sweeper nor any outsider can link the interaction and payment in which the user payed to the sweeper to the interaction and payment in which the sweeper payed 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 composability (UC) framework [16] with static corruptions. In terms of communication, our protocol makes use of secure channels and anonymous channels. Also, similar to other works in this area, e.g. [19,39], we consider a synchronous model of communication. This means that we implicitly assume a global clock functionality, and protocols are executed in rounds. Every party knows the current round. Thus, the parties and functionalities can expect messages to be received at a certain time.

**Ledger Functionality.** As in previous works [19,39], we model the blockchain as a global ledger functionality  $\mathcal{L}^{\text{SIG}}$  parameterized by a signature scheme SIG. We postpone the formal presentation of  $\mathcal{L}^{\text{SIG}}$  to Figure 16. The functionality holds the current balances  $\text{bal}[\text{pk}] \in \mathbb{N}_0$  of public keys  $\text{pk}$ . Parties can call  $\mathcal{L}^{\text{SIG}}.\text{Pay}(\text{pk}_s, \text{pk}_r, c, \text{sk}_s)$  to pay  $c$  coins from address  $\text{pk}_s$  to address  $\text{pk}_r$  using secret key  $\text{sk}_s$ . Further, we allow functionalities to call interfaces  $\mathcal{L}^{\text{SIG}}.\text{Freeze}(\text{pk}, c)$

and  $\mathcal{L}^{\text{SIG}}.\text{Unfreeze}(\text{pk}', c)$  to freeze  $c$  coins of an address  $\text{pk}$  or to unfreeze them into an address  $\text{pk}'$ . Also, our protocol makes use of a functionality  $\mathcal{F}_s$ , formally specified in Figure 17. Via interface  $\mathcal{F}_s.\text{OpenSh}(T, \text{pk}_{in}, \mathcal{P}_b, c, \text{sk}_{in})$  this functionality allows a party  $\mathcal{P}_a$  to open a shared address  $(\text{pk}_a, \text{pk}_b)$  with party  $\mathcal{P}_b$  by paying  $c$  coins from  $\text{pk}_{in}$  into it. As a result,  $\mathcal{P}_a$  gets secret key share  $\text{sk}_a$  and  $\mathcal{P}_b$  gets secret key share  $\text{sk}_b$ . Later, it can be closed using  $\mathcal{F}_s.\text{CloseSh}(\text{pk}_a, \text{pk}_b, \text{pk}_{out}, c, \sigma_a, \sigma_b)$ , where  $\sigma_a, \sigma_b$  are valid signatures on a closing transaction  $\text{tx}$  with respect to  $\text{pk}_a, \text{pk}_b$ , respectively. In this case, the  $c$  coins are transferred to  $\text{pk}_{out}$ . If the shared address is not closed after timeout  $T$ , the coins go back to  $\text{pk}_{in}$ . 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  $\mathcal{L}^{\text{SIG}}.\text{Freeze}$  and  $\mathcal{L}^{\text{SIG}}.\text{Unfreeze}$  are only called by  $\mathcal{F}_s$ , and it is well known [39] 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  $\mathcal{F}_{\text{ux}}$  for unlinkable exchanges. The formal presentation of the functionality is postponed Figure 15. The functionality interacts with  $\mathcal{L}^{\text{SIG}}$ . It is parameterized by a timeout parameter  $T$  and an amount  $\text{amt}$ . All payments will have this fixed amount, which is important to maximize the anonymity set. When a user  $\mathcal{P}$  wants to use  $\mathcal{F}_{\text{ux}}$  to exchange coins with the sweeper  $\mathcal{W}$ , it first calls interface  $\mathcal{F}_{\text{ux}}.\text{Register}(\text{pk}_b)$ , which freezes  $\text{amt}$  coins of some fixed public key  $\text{pk}_{\mathcal{W}}$  of  $\mathcal{W}$ . Here, the adversary learns  $\mathcal{P}, \text{pk}_b$ . Next, party  $\mathcal{P}$  calls  $\mathcal{F}_{\text{ux}}.\text{AddPayment}(\text{pk}_a, \text{sk}_a, \text{pk}_b)$ , which leads to  $\text{amt}$  coins of  $\text{pk}_a$  being transferred to  $\text{pk}_{\mathcal{W}}$ . Here, the adversary only learns  $\text{pk}_a$ , and not  $\mathcal{P}, \text{pk}_b$ . Finally, party  $\mathcal{P}$  calls  $\mathcal{F}_{\text{ux}}.\text{GetPayment}(\text{pk}_b)$ . If the corresponding calls to **Register** and **AddPayment** were issued correctly, this leads to unfreezing the  $\text{amt}$  coins that were frozen in **Register** into address  $\text{pk}_b$ . In this way,  $\mathcal{P}$  payed  $\text{amt}$  coins from address  $\text{pk}_a$  to  $\mathcal{W}$  and received  $\text{amt}$  coins to  $\text{pk}_b$  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  $\text{pk}_b$  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  $\text{pk}_{\mathcal{W}}$  pays stays the same when calling this interface.

Let us argue how the informal security properties discussed above are captured by  $\mathcal{F}_{\text{ux}}$ . 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  $\text{pk}_b$  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  $\text{pk}_b$ , 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  $\mathcal{F}_{\text{ux}}$ . 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 condition 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  $x \in \{r = \text{Register}, a = \text{AddPayment}, g = \text{GetPayment}\}$  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 common phases 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  $\mathcal{L}^{\text{SIG}}$  in `Register` and `GetPayment` are completely independent from the calls to  $\mathcal{L}^{\text{SIG}}$  in `AddPayment`.

## 5 Building Blocks for Sweep-UC

In this section, we focus on the building blocks for our protocol. First, we define an exchange protocol and give different instantiations of it. Then, we define a redeem protocol and present constructions. In a nutshell, using an exchange protocol, a user will buy a blind signature from the sweeper. Then, using the redeem protocol, it can turn it in to get a signed transaction from the sweeper. Throughout, we use the terminology “on the left/right” following Figures 1 and 14.

### 5.1 Exchange Protocol

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

**Setting.** Consider the following scenario for a signature scheme `SIG` and a blind signature scheme `BS`. A buyer and a seller<sup>13</sup> opened a shared address  $(\text{pk}_b, \text{pk}_s)$  for `SIG`, where the buyer knows the secret key  $\text{sk}_b$  corresponding to  $\text{pk}_b$ , and the seller knows the secret key  $\text{sk}_s$  corresponding to  $\text{pk}_s$ . Both parties are aware of a public key  $\text{pk}_{\text{BS}}$  for `BS`, and the seller knows the corresponding secret key  $\text{sk}_{\text{BS}}$ . Assume that the signing protocol of `BS` consists of two messages,  $\text{bsm}_1$  and

<sup>13</sup> 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.

$\text{bsm}_2$ . Then, the buyer has some nonce  $\text{sn}$  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  $\text{tx}$  under the shared address  $(\text{pk}_b, \text{pk}_s)$ . More precisely, first, the buyer sends the first message  $\text{bsm}_1$  of the blind signature interaction. Then, both parties run an exchange protocol to fairly exchange the message  $\text{bsm}_2$  for a signature  $(\sigma_b, \sigma_s)$  on transaction  $\text{tx}$ .

**Syntax.** In our syntax of this exchange, we assume that the overall parameters  $\text{xpar} := (\text{pk}_{\text{BS}}, \text{bsm}_1, \text{pk}_b, \text{pk}_s, \text{tx})$  are known to the seller and the buyer. Then, the seller first sends a message  $\text{xm}_1$  to the buyer, which is computed using the first message  $\text{bsm}_1$  and the secret key  $\text{sk}_{\text{BS}}$ , and may already encapsulate the second message  $\text{bsm}_2$  in some sense. Then, the buyer responds with a message  $\text{xm}_2$ . Now, the seller can derive the signature  $\sigma_b$  from  $\text{xm}_2$ . Whenever the seller publishes  $(\sigma_b, \sigma_s)$ , the buyer can derive a valid second message  $\text{bsm}_2$  from the transcript  $\text{xm}_1, \text{xm}_2$  and  $(\sigma_b, \sigma_s)$ . An overview of this can be found in Figure 9.

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

- $\text{Setup}(\text{xpar}, \text{sk}_{\text{BS}}, \text{sk}_s) \rightarrow (\text{xm}_1, \text{St})$  takes as input exchange parameters  $\text{xpar}$ , a secret key  $\text{sk}_{\text{BS}}$ , and a secret key  $\text{sk}_s$ , and outputs a message  $\text{xm}_1$  and a state  $\text{St}$ .
- $\text{Buy}(\text{xpar}, \text{sk}_b, \text{xm}_1) \rightarrow \text{xm}_2$  takes as input exchange parameters  $\text{xpar}$ , a secret key  $\text{sk}_b$ , and a message  $\text{xm}_1$ , and outputs a message  $\text{xm}_2$ .
- $\text{Sell}(\text{St}, \text{xm}_2) \rightarrow \sigma_b$  is deterministic, takes as input a state  $\text{St}$  and a message  $\text{xm}_2$ , and outputs a signature  $\sigma_b$ .
- $\text{Get}(\text{xpar}, \text{xm}_1, \text{xm}_2, \sigma_b, \sigma_s) \rightarrow \text{bsm}_2$  is deterministic, takes as input exchange parameters  $\text{xpar}$ , messages  $\text{xm}_1$  and  $\text{xm}_2$ , and signatures  $\sigma_b$  and  $\sigma_s$ , and outputs a message  $\text{bsm}_2$ .

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

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

We require that an exchange protocol has well distributed signatures. That is, signatures on a transaction  $\text{tx}$  obtained from the exchange protocol should be distributed identically to freshly computed signatures. We postpone the formal definition of this property to Supplementary Material B.

**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  $\text{xm}_2$ , the seller should not be able to derive a signature on  $\text{tx}$ . The seller should only be able to derive a signature for the given transaction  $\text{tx}$ . 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  $\sigma_b$ . We formalize this via simulators that do not get  $\text{sk}_{\text{BS}}$  as input. Our definition captures the intuition that the only information about  $\text{sk}_{\text{BS}}$  that is revealed is  $\text{bsm}_2$ , and this is only revealed once the signatures  $\sigma_b, \sigma_s$  are published.

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

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

1. *Run  $\mathcal{A}$  and obtain a public key  $\text{pk}_{\text{BS}}$  and a message  $\text{sn}$  for BS.*
2. *Run  $(\text{bsm}_1, St) \leftarrow U_1(\text{pk}_{\text{BS}}, \text{sn})$ .*
3. *Sample keys  $(\text{pk}_b, \text{sk}_b) \leftarrow \text{SIG.Gen}(1^\lambda)$ .*
4. *Run  $\mathcal{A}$  on input  $\text{pk}_b$  and  $\text{bsm}_1$ . Obtain  $\text{pk}_s, \text{tx}$ , and a message  $\text{xm}_1$  from  $\mathcal{A}$ . Set  $\text{xpar} := (\text{pk}_{\text{BS}}, \text{bsm}_1, \text{pk}_b, \text{pk}_s, \text{tx})$ .*
5. *If  $\text{xm}_1 \neq \perp$ , run  $\text{xm}_2 \leftarrow \text{Buy}(\text{xpar}, \text{sk}_b, \text{xm}_1)$  and give  $\text{xm}_2$  to  $\mathcal{A}$ . Otherwise, give  $\text{xm}_2 := \perp$  to  $\mathcal{A}$ .*
6. *Obtain  $\text{tx}'$  and  $\sigma_b, \sigma_s$  from  $\mathcal{A}$  and run  $\text{bsm}_2 := \text{Get}(\text{xpar}, \text{xm}_1, \text{xm}_2, \sigma_b, \sigma_s)$  and  $\sigma_{\text{BS}} \leftarrow U_2(St, \text{bsm}_2)$ .*
7. *If  $\text{SIG.Ver}(\text{pk}_b, \text{tx}', \sigma_b) = 0$  or  $\text{SIG.Ver}(\text{pk}_s, \text{tx}', \sigma_s) = 0$ , output 0.*
8. *Output 1 if one of the following holds, otherwise output 0:*
  - (a)  $\text{tx} \neq \text{tx}'$ .
  - (b)  $\text{tx} = \text{tx}'$  and  $\text{xm}_2 = \perp$ .
  - (c)  $\text{tx} = \text{tx}'$ ,  $\text{xm}_2 \neq \perp$ , and  $\text{BS.Ver}(\text{pk}_{\text{BS}}, \text{sn}, \sigma_{\text{BS}}) = 0$ .

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

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



1. Sample a key pair  $(pk_{BS}, sk_{BS}) \leftarrow BS.Gen(1^\lambda)$ .
2. Let  $O$  be an oracle that takes as input  $bsm_1$  and returns  $bsm_2 \leftarrow BS.S(sk_{BS}, bsm_1)$ .
3. Run  $\mathcal{A}$  on input  $pk_{BS}$  with access to oracle  $O$  and an interactive oracle  $O^*$ , which is defined as follows:
  - (a) Upon receiving a call, run  $(pk_s, sk_s) \leftarrow SIG.Gen(1^\lambda)$  and return  $pk_s$ .
  - (b) Upon receiving a key  $pk_b$ , a transaction  $tx$ , and a message  $bsm_1$ , set  $xpar := (pk_{BS}, bsm_1, pk_b, pk_s, tx)$ . If  $b = 0$ , run  $(xm_1, St) \leftarrow Setup(xpar, sk_{BS}, sk_s)$ . If  $b = 1$ , run  $xm_1 \leftarrow Sim_1(xpar, sk_s)$ . Return  $xm_1$ .
  - (c) Upon receiving  $xm_2$ , run  $\sigma_s \leftarrow SIG.Sig(sk_s, tx)$ . If  $b = 0$ , run  $\sigma_b := Sell(St, xm_2)$ , and abort if  $SIG.Ver(pk_b, tx, \sigma_b) = 0$ . If  $b = 1$ , abort if  $Sim_2(xm_2) = 0$ . Otherwise, run  $bsm_2 \leftarrow BS.S(sk_{BS}, bsm_1)$  and  $\sigma_b \leftarrow Sim_3(xm_2, bsm_2)$ . Return  $\sigma_b, \sigma_s$ .
4. Obtain a bit  $b'$  from  $\mathcal{A}$ . Output  $b'$ .

Note that algorithms  $Sim_1, Sim_{RO}, Sim_2, Sim_3$  do not have access to oracle  $O$ .

We say that EXC is secure against malicious buyers, if there are PPT algorithms  $Sim_1, Sim_{RO}, Sim_2, Sim_3$  as above, such that for all PPT algorithms  $\mathcal{A}$  the probability that the game with  $b = 0$  outputs 1 and the probability that the game with  $b = 1$  outputs 1 are negligibly close.

**Toy Constructions.** In Supplementary Material C.1 and Supplementary Material C.2, we give two simple constructions of an exchange protocol. The first construction is for any unique signature scheme. The second construction is for any signature scheme supporting adaptor signatures. Their drawback is that we treat a random oracle as a circuit, which has unclear security implications. The reader may use these constructions as a starting point introducing the overall strategy.

**Constructions using Cut-and-Choose.** We give two concrete constructions of an exchange protocol using a cut-and-choose technique, avoiding the need to treat a random oracle as a circuit. In the first construction, the signature scheme  $SIG = (SIG.Gen, SIG.Sig, SIG.Ver)$  is the BLS signature scheme [14]. The second construction uses adaptor signatures for a discrete logarithm relation. Due to space limitations, it is given in Supplementary Material C.3. In both cases, the blind signature scheme BS is the BLS blind signature scheme (see Supplementary Material H). 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  $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$  is a pairing. Let  $\ell = \ell(\lambda)$  denote an upper bound on the bit length of messages  $bsm_2$  sent in signing interactions of BS. We make use of random oracles  $H : \{0, 1\}^* \rightarrow \{0, 1\}^\ell$  and  $H_c : \{0, 1\}^* \rightarrow \{0, 1\}^\lambda$ . The schemes are called  $EXC_{BLS}^{cc}[SIG, BS]$  and  $EXC_a^{cc}[SIG, aSIG, BS]$ , respectively, and given in Figure 2 for BLS and Figure 6 for adaptor signatures. The security proofs are given in Supplementary Material D.

**Lemma 1.** *Assume that the BLS signature scheme SIG is EUF-CMA secure. Then the exchange protocol  $EXC_{BLS}^{cc}[SIG, BS]$  is secure against malicious sellers.*

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

## 5.2 Redeem Protocol

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

**Setting.** Informally, we consider the following scenario. Assume that a service and a user<sup>14</sup> are aware of a public key  $\text{pk}_{\text{BS}}$  for a blind signature scheme BS. The service holds the corresponding secret key  $\text{sk}_{\text{BS}}$ . Further, the service published a public key  $\text{pk}_s$  for signature scheme SIG, for which it knows a secret key  $\text{sk}_s$ . Additionally, both parties agreed on a transaction  $\text{tx}$  and a message  $\text{sn}$ . Then, the goal of both parties is to move towards a state, in which the user can use a blind signature  $\sigma_{\text{BS}}$  that is valid for message  $\text{sn}$  and key  $\text{pk}_{\text{BS}}$ , to obtain a signature  $\sigma_s$  which is valid for  $\text{tx}$  under key  $\text{pk}_s$ . This transformation of  $\sigma_{\text{BS}}$  into  $\sigma_s$  should be possible without any further interaction with the service. Moreover, the service wants to ensure that without knowing the blind signature  $\sigma_{\text{BS}}$ , it should not be possible to obtain  $\sigma_s$ . In other words, both parties want to run a protocol such that afterwards, the user is able to turn in  $\sigma_{\text{BS}}$  non-interactively and get a signature  $\sigma_s$  on the transaction  $\text{tx}$  for it.

**Syntax.** In our syntax, we assume that the parameters  $\text{rpar} := (\text{pk}_{\text{BS}}, \text{pk}_s, \text{tx}, \text{sn})$  are known to both parties. The service first sends a promise message  $\text{prom}$ . This message can be verified by the user using the public key  $\text{pk}_{\text{BS}}$ . Intuitively, this verification step should guarantee that the user can be sure to obtain a valid signature  $\sigma_s$  from  $\text{prom}$  as soon as it knows  $\sigma_{\text{BS}}$ . Finally, the user can use  $\sigma_{\text{BS}}$  and  $\text{prom}$  to derive the signature  $\sigma_s$  on the transaction  $\text{tx}$ . An overview of this can be found in Figure 10.

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

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

Further, it is required that the following completeness property holds: For all transactions  $\text{tx}$ , all messages  $\text{sn}$ , all keys  $(\text{pk}_{\text{BS}}, \text{sk}_{\text{BS}}) \in \text{BS.Gen}(1^\lambda)$ , all  $(\text{pk}_s, \text{sk}_s) \in$

<sup>14</sup> In the context of protocol Sweep-UC, the user Alice will act as the user of the redeem protocol, and the sweeper will have the role of the service.

```

Setup(xpar = (pkBS, bsm1, pkb, pks, tx), skBS, sks)
// Share bsm2 = bsm1skBS and σs
01 r1, ..., rλ ←$ ℤp, r'1, ..., r'λ ←$ ℤp
02 f(X) = skBS + ∑j=1λ rj · Xj ∈ ℤp[X], f'(X) = sks + ∑j=1λ r'j · Xj ∈ ℤp[X]
03 for j ∈ [2λ] : skBS,j := f(j), bsm2,j ← S(skBS,j, bsm1)
04 for j ∈ [2λ] : sks,j := f'(j), σj ← SIG.Sig(sks,j, tx)
05 for j ∈ [λ] : coeffj := g2rj, coeff'j := g2r'j
// Encrypt bsm2,j with σj
06 for j ∈ [2λ] : ctj := H(σj) ⊕ bsm2,j
// Cut-and-choose
07 xm1,1 := ((ctj)j∈[2λ], (coeffj, coeff'j)j∈[λ])
08 b0 ... bλ-1 := Hc(xm1,1), for j ∈ [λ] : kj := 2j - bj-1
09 return (xm1 := (xm1,1, xm1,2 := (σkj)j∈[λ]), St := ⊥)

Buy(xpar = (pkBS, bsm1, pkb, pks, tx), skb, xm1 = (xm1,1, xm1,2))
// Verify cut-and-choose
10 b0 ... bλ-1 := Hc(xm1,1)
11 for j ∈ [λ] :
12   kj := 2j - bj-1, pkBS,kj := pkBS · ∏i=1λ (coeffi)kj, pks,kj := pks · ∏i=1λ (coeff'i)kj
13   bsm2,kj := ctkj ⊕ H(σkj)
14   if e(bsm1, pkBS,kj) ≠ e(bsm2,kj, g2) ∨ SIG.Ver(pks,kj, tx, σkj) = 0 : return ⊥
// Return a signature
15 return xm2 := σb ← SIG.Sig(skb, tx)

Sell(St, xm2 = σb)
16 if SIG.Ver(pkb, tx, σb) = 0 : return ⊥
17 return σb

Get(xpar = (pkBS, bsm1, pkb, pks, tx), xm1, xm2, σb, σs)
18 b0 ... bλ-1 := Hc(xm1,1)
// Reconstruct all shares
19 for j ∈ [λ] : kj := 2j - bj-1, k̄j := 2j - (1 - bj-1), bsm2,kj := ctkj ⊕ H(σkj)
// Find a valid share
20 w := 0
21 for j ∈ [λ] :
22   σ̄kj := reconstg1, k̄j((0, σs), (ki, σki)i∈[λ]), bsm2,k̄j := ctk̄j ⊕ H(σ̄kj)
23   pkBS,k̄j := pkBS · ∏i∈[λ] (coeffi)k̄j
24   if e(bsm1, pkBS,k̄j) = e(bsm2,k̄j, g2) : w := k̄j
25 if w = 0 : return ⊥
// Reconstruct bsm2
26 return bsm2 := reconstg1, 0((w, bsm2,w), (kj, bsm2,kj)j∈[λ])

```

**Fig. 2.** The exchange protocol  $\text{EXC}_{\text{BLS}}^{\text{CS}}[\text{SIG}, \text{BS}] = (\text{Setup}, \text{Buy}, \text{Sell}, \text{Get})$  for BLS signature scheme SIG, and blind BLS signature scheme BS. Here,  $H : \{0, 1\}^* \rightarrow \{0, 1\}^\ell$  and  $H_c : \{0, 1\}^* \rightarrow \{0, 1\}^\lambda$  are random oracles and  $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$  is a pairing.

$\text{SIG.Gen}(1^\lambda)$ , we have

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

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

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

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

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

We say that  $(\text{Sim}, \text{Sim}_{\text{RO}})$  is a simulator against malicious users for  $\text{RP}$ , 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  $\text{Sim}, \text{Sim}_{\text{RO}}, \text{Ext}$ , which may share state, consider the following game:*

1. Sample keys  $(\text{pk}_{\text{BS}}, \text{sk}_{\text{BS}}) \leftarrow \text{BS.Gen}(1^\lambda)$  and initialize an empty list  $\text{DSpend}$  and set  $\text{bad} := 0$ .
2. Let  $\mathcal{O}$  be an interactive oracle that on input  $\text{sn}$  does the following:
  - (a) If  $\text{sn} \in \text{DSpend}$ , abort. Otherwise, add  $\text{sn}$  to  $\text{DSpend}$ .
  - (b) Sample keys  $(\text{pk}_s, \text{sk}_s) \leftarrow \text{SIG.Gen}(1^\lambda)$  and output  $\text{pk}_s$ .
  - (c) Receive  $\text{tx}$  and set  $\text{rpar} := (\text{pk}_{\text{BS}}, \text{pk}_s, \text{tx}, \text{sn})$ .

- (d) Run  $\text{prom} \leftarrow \text{Sim}(\text{rpar}, \text{sk}_s)$  and output  $\text{prom}$ .
  - (e) Get  $\sigma_s$  as input and run  $\sigma_{\text{BS}} \leftarrow \text{Ext}(\text{rpar}, \text{sk}_s, \sigma_s)$ .
  - (f) If  $\text{BS.Ver}(\text{pk}_{\text{BS}}, \text{sn}, \sigma_{\text{BS}}) = 0$  and  $\text{SIG.Ver}(\text{pk}_s, \text{tx}, \sigma_s) = 1$ , set  $\text{bad} := 1$ .
3. Run  $\mathcal{A}$  on input  $\text{pk}_{\text{BS}}, \text{sk}_{\text{BS}}$  with access to oracle  $O$ . During  $\mathcal{A}$ 's execution, use algorithm  $\text{Sim}_{\text{RO}}$  to provide the random oracle.
  4. Output  $\text{bad}$ .

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

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

**Definition 6 (Security Against Malicious Services).** Let  $\text{RP} = (\text{Promise}, \text{VerPromise}, \text{Redeem})$  be a redeem protocol for  $\text{SIG}$  and  $\text{BS}$  as in Definition 4. For any algorithm  $\mathcal{A}$  and any algorithm  $\text{Ext}$ , consider the following game:

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

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

**Toy Construction.** We generically construct a redeem protocol for any signature scheme and any unique blind signature scheme. The drawback of this scheme is that it uses proofs about relations defined by random oracles. We postpone the details to Supplementary Material E.1.

**Constructions using Cut-and-Choose.** We give two constructions of a redeem protocol without relying on proof systems that argue about the random oracle. For the first construction we assume that the signature scheme associated with  $\text{pk}_s$  is the BLS signature scheme  $\text{SIG}$  defined over cyclic groups  $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$  of prime order  $p$  with respective generators  $g_1 \in \mathbb{G}_1, g_2 \in \mathbb{G}_2$ , and  $e(g_1, g_2) \in \mathbb{G}_T$ . The second construction works with a Schnorr signature  $\text{SIG}$  and is postponed to Supplementary Material E.2. In both cases we use the BLS blind signature scheme. Both signature schemes use the random oracle  $\text{H} : \{0, 1\}^* \rightarrow \mathbb{G}_1$ , as the oracle for the BLS and blind BLS signature. Moreover, we let  $\text{H}_c : \{0, 1\}^* \rightarrow \{0, 1\}^\lambda$ ,  $\hat{\text{H}} : \{0, 1\}^* \rightarrow \mathbb{G}_1$ , and  $\text{H}_p : \{0, 1\}^* \rightarrow \mathbb{Z}_p^*$  be random oracles. The resulting schemes  $\text{RP}_{\text{BLS}}^{\text{cc}}[\text{SIG}, \text{BS}]$  and  $\text{RP}_{\text{Schn}}^{\text{cc}}[\text{SIG}, \text{BS}]$  are given in Figure 3 and Figure 8, respectively. The security proofs are given in Supplementary Material F.

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

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

## 6 Sweep-UC: The Complete Protocol

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

**Setup.** Assume that  $\text{BS} = (\text{BS.Gen}, \text{BS.S}, \text{BS.U}, \text{BS.Ver})$  is a two-move<sup>15</sup> blind signature scheme. Let  $\text{EXC} = (\text{Setup}, \text{Buy}, \text{Sell}, \text{Get})$  be an exchange protocol and  $\text{RP} = (\text{Promise}, \text{VerPromise}, \text{Redeem})$  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  $(\mathcal{L}^{\text{SIG}}, \mathcal{F}_s)$ -hybrid model. At setup time, a key pair  $(\text{pk}_{\text{BS}}, \text{sk}_{\text{BS}}) \leftarrow \text{BS.Gen}(1^\lambda)$  is generated. The sweeper  $\mathcal{W}$  is initialized with  $\text{sk}_{\text{BS}}$ . All parties are initialized with the corresponding public key  $\text{pk}_{\text{BS}}$ . Further,  $\mathcal{W}$  holds a secret key  $\text{sk}_{\mathcal{W}}$  for public key  $\text{pk}_{\mathcal{W}}$  for signature scheme SIG, and lists  $\text{Reg}, \text{DSpend}$ , 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.

**Register( $\text{pk}_b$ ):** 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  $\text{sn} \leftarrow_{\mathcal{S}} \{0, 1\}^\lambda$  and sends  $\text{sn}, \text{pk}_b$  to  $\mathcal{W}$ .
2. *Opening a Shared Address:* Then,  $\mathcal{W}$  aborts if  $\text{sn} \in \text{DSpend}$  or  $\text{pk}_b \in \text{Reg}$ . Otherwise, it adds these entries to the respective lists. Then, it calls  $\mathcal{F}_s.\text{OpenSh}(T, \text{pk}_{\mathcal{W}}, \mathcal{P}, \text{amt}, \text{sk}_{\mathcal{W}})$ . As a result,  $\mathcal{W}$  obtains  $(\bar{\text{pk}}_{r, \mathcal{W}}, \bar{\text{pk}}_{r, \mathcal{P}}, \bar{\text{sk}}_{r, \mathcal{W}})$  from  $\mathcal{F}_s$  and  $\mathcal{P}$  obtains  $(\bar{\text{pk}}_{r, \mathcal{W}}, \bar{\text{pk}}_{r, \mathcal{P}}, \bar{\text{sk}}_{r, \mathcal{P}})$  from  $\mathcal{F}_s$ .
3. *Making a Promise:* Both parties  $\mathcal{P}$  and  $\mathcal{W}$  set  $\text{tx}_r := (\bar{\text{pk}}_{r, \mathcal{W}}, \bar{\text{pk}}_{r, \mathcal{P}}, \text{pk}_b, \text{amt})$ . Also, both set the redeem parameters  $\text{rpar} := (\text{pk}_{\text{BS}}, \bar{\text{pk}}_{r, \mathcal{W}}, \text{tx}_r, \text{sn})$ . Then,  $\mathcal{W}$  computes a promise message  $\text{prom} \leftarrow \text{Promise}(\text{rpar}, \text{sk}_{\text{BS}}, \bar{\text{sk}}_{r, \mathcal{W}})$  and sends  $\text{prom}$  to  $\mathcal{P}$ .

<sup>15</sup> We only assume two moves for simplicity of exposition. The construction can naturally be generalized to more moves.

```

Promise(rpar, skBS, sks)
01  $\sigma_s := H(\text{tx})^{\text{sk}_s}$ ,  $h := \hat{H}(\text{sn})$ ,  $s_0 \leftarrow \mathbb{Z}_p$ ,  $\text{ct}_0 := h^{s_0} \cdot \sigma_s$ 
// Share  $\sigma_{\text{BS}}$  and  $h^{s_0}$ 
02  $r_1, \dots, r_\lambda \leftarrow \mathbb{Z}_p$ ,  $r'_0, \dots, r'_\lambda \leftarrow \mathbb{Z}_p$ ,  $\text{coeff}'_0 := g_1^{s_0}$ 
03  $f(X) := \text{sk}_{\text{BS}} + \sum_{j=1}^\lambda r_j \cdot X^j$ ,  $f'(X) := s_0 + \sum_{j=1}^\lambda r'_j \cdot X^j \in \mathbb{Z}_p[X]$ 
04 for  $j \in [2\lambda]$ :  $\text{sk}_j := f(j)$ ,  $s_j := f'(j)$ ,  $\sigma_j := H(\text{sn})^{\text{sk}_j}$ 
05 for  $j \in [\lambda]$ :  $\text{coeff}_j := g_2^{r_j}$ ,  $\text{coeff}'_j := g_1^{r'_j}$ 
// Encrypt  $h^{s_j}$  with  $\sigma_j$ 
06 for  $j \in [2\lambda]$ :  $\text{ct}_j := \hat{H}(\text{sn}, \sigma_j) \cdot h^{s_j}$ 
// Prove that  $\text{ct}_0$  is well-formed
07  $t_0, t_1 \leftarrow \mathbb{Z}_p^*$ ,  $T_0 := h^{t_0} \cdot H(\text{tx})^{t_1}$ ,  $T_1 := g_1^{t_0}$ ,  $T_2 := g_2^{t_1}$ 
08  $e := H_p(T_0, T_1, T_2, h, H(\text{tx}), \text{ct}_0, \text{coeff}'_0, \text{pk}_s)$ ,  $\pi_0 := t_0 + e \cdot s_0$ ,  $\pi_1 := t_1 + e \cdot \text{sk}_s$ 
// Cut-and-choose
09  $\text{prom}_1 := (\text{ct}_0, (\text{ct}_j)_{j \in [2\lambda]}, (\pi_0, \pi_1, e), \text{coeff}'_0, (\text{coeff}_j, \text{coeff}'_j)_{j \in [\lambda]})$ 
10  $b_0 \dots b_{\lambda-1} := H_c(\text{prom}_1)$ , for  $j \in [\lambda]$ :  $k_j := 2j - b_{j-1}$ 
11 return  $\text{prom} := (\text{prom}_1, \text{prom}_2 := (\sigma_{k_j}, s_{k_j})_{j \in [\lambda]})$ 

VerPromise(rpar, prom = (prom1, prom2 = (σBS,kj, skj)j ∈ [λ]))
12  $h := \hat{H}(\text{sn})$ ,  $b_0 \dots b_{\lambda-1} := H_c(\text{prom}_1)$ 
// Verify cut-and-choose
13 for  $j \in [\lambda]$ :
14  $k_j := 2j - b_{j-1}$ ,  $\text{pk}_{\text{BS},k_j} := \text{pk}_{\text{BS}} \cdot \prod_{i=1}^\lambda (\text{coeff}_i)^{k_j^i}$ 
15 if  $\text{ct}_{k_j} \neq \hat{H}(\text{sn}, \sigma_{k_j}) \cdot h^{s_{k_j}} \vee g_1^{s_{k_j}} \neq \prod_{i=0}^\lambda (\text{coeff}'_i)^{k_j^i}$ : return 0
16 if  $\text{BS.Ver}(\text{pk}_{\text{BS},k_j}, \text{sn}, \sigma_{k_j}) = 0$ : return 0
// Verify that  $\text{ct}_0$  is well-formed
17  $\hat{T}_0 := h^{\pi_0} \cdot H(\text{tx})^{\pi_1} \cdot \text{ct}_0^{-e}$ ,  $\hat{T}_1 := g_1^{\pi_0} \cdot (\text{coeff}'_0)^{-e}$ ,  $\hat{T}_2 := g_2^{\pi_1} \cdot (\text{pk}_s)^{-e}$ 
18 if  $e \neq H_p(\hat{T}_0, \hat{T}_1, \hat{T}_2, h, H(\text{tx}), \text{ct}_0, \text{coeff}'_0, \text{pk}_s)$ : return 0
19 return 1

Redeem(rpar, prom = (prom1, prom2), σBS)
20  $h := \hat{H}(\text{sn})$ ,  $b_0 \dots b_{\lambda-1} := H_c(\text{prom}_1)$ 
// Reconstruct all shares
21 for  $j \in [\lambda]$ :
22  $k_j := 2j - b_{j-1}$ ,  $\bar{k}_j := 2j - (1 - b_{j-1})$ ,  $h_{k_j} := h^{s_{k_j}}$ 
23  $\sigma_{\bar{k}_j} := \text{reconst}_{g_1, \bar{k}_j}((0, \sigma_{\text{BS}}), (k_j, \sigma_{k_i})_{i \in [\lambda]})$ ,  $h_{\bar{k}_j} := \text{ct}_{\bar{k}_j} / \hat{H}(\text{sn}, \sigma_{\bar{k}_j})$ 
// Try to decrypt  $\text{ct}_0$ 
24 for  $j \in [\lambda]$ :
25  $h_0 := \text{reconst}_{g_1, 0}((\bar{k}_j, h_{\bar{k}_j}), (k_i, h_{k_i})_{i \in [\lambda]})$ ,  $\sigma_s := \text{ct}_0 / h_0$ 
26 if  $\text{SIG.Ver}(\text{pk}_s, \text{tx}, \sigma_s) = 1$ : return  $\sigma_s$ 
27 return  $\perp$ 

```

**Fig. 3.** The redeem protocol  $\text{RP}_{\text{BLS}}^{\text{cc}}[\text{SIG}, \text{BS}] = (\text{Promise}, \text{VerPromise}, \text{Redeem})$  for the BLS signature scheme SIG and the blind BLS signature scheme BS. Here,  $H : \{0, 1\}^* \rightarrow \mathbb{G}_1$ ,  $H_c : \{0, 1\}^* \rightarrow \{0, 1\}^\lambda$ ,  $H_p : \{0, 1\}^* \rightarrow \mathbb{Z}_p^*$  and  $\hat{H} : \{0, 1\}^* \rightarrow \mathbb{G}_1$  are random oracles.

4. *Verifying the Promise:*  $\mathcal{P}$  runs  $b := \text{VerPromise}(\text{rpar}, \text{prom})$ . If  $b = 0$ , it aborts the entire execution.

**AddPayment**( $\text{pk}_a, \text{sk}_a, \text{pk}_b$ ): 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  $(\text{bsm}_1, St) \leftarrow \text{BS.U}_1(\text{pk}_{\text{BS}}, \text{sn})$ . It sends  $\text{bsm}_1$  to  $\mathcal{W}$ .
2. *Opening a Shared Address:* Then,  $\mathcal{P}$  calls  $\mathcal{F}_s.\text{OpenSh}(T, \text{pk}_a, \mathcal{W}, \text{amt}, \text{sk}_a)$ . As a result,  $\mathcal{W}$  obtains  $(\bar{\text{pk}}_{l,\mathcal{P}}, \bar{\text{pk}}_{l,\mathcal{W}}, \bar{\text{sk}}_{l,\mathcal{W}})$  and  $\mathcal{P}$  obtains  $(\text{pk}_{l,\mathcal{P}}, \bar{\text{pk}}_{l,\mathcal{W}}, \text{sk}_{l,\mathcal{P}})$ .
3. *Running the Exchange:* Both parties define a transaction  $\text{tx}_l := (\text{pk}_{l,\mathcal{P}}, \bar{\text{pk}}_{l,\mathcal{W}}, \text{pk}_{\mathcal{W}}, \text{amt})$  and exchange parameters  $\text{xpar} := (\text{pk}_{\text{BS}}, \text{bsm}_1, \bar{\text{pk}}_{l,\mathcal{P}}, \bar{\text{pk}}_{l,\mathcal{W}}, \text{tx}_l)$ . Then, the sweeper runs  $(\text{xm}_1, St) \leftarrow \text{Setup}(\text{xpar}, \text{sk}_{\text{BS}}, \bar{\text{sk}}_{l,\mathcal{W}})$ . It sends  $\text{xm}_1$  to  $\mathcal{P}$ . Then,  $\mathcal{P}$  runs  $\text{xm}_2 \leftarrow \text{Buy}(\text{xpar}, \text{sk}_{l,\mathcal{P}}, \text{xm}_1)$  and sends  $\text{xm}_2$  to  $\mathcal{W}$ . Then,  $\mathcal{W}$  runs  $\sigma_{l,\mathcal{P}} := \text{Sell}(St, \text{xm}_2)$ . Additionally,  $\mathcal{W}$  computes  $\sigma_{l,\mathcal{W}} \leftarrow \text{SIG.Sig}(\text{sk}_{l,\mathcal{W}}, \text{tx}_l)$ .
4. *Closing the Shared Address:*  $\mathcal{W}$  calls  $\mathcal{F}_s.\text{CloseSh}(\bar{\text{pk}}_{l,\mathcal{P}}, \bar{\text{pk}}_{l,\mathcal{W}}, \text{pk}_{\mathcal{W}}, \text{amt}, \sigma_{l,\mathcal{P}}, \sigma_{l,\mathcal{W}})$ . Then,  $\mathcal{P}$  receives (“closedSharedAddress”,  $\bar{\text{pk}}_{l,\mathcal{P}}, \bar{\text{pk}}_{l,\mathcal{W}}, \text{pk}_{\mathcal{W}}, \text{amt}, \sigma_{l,\mathcal{P}}, \sigma_{l,\mathcal{W}}$ ) from  $\mathcal{F}_s$ . Finally, it computes message  $\text{bsm}_2 := \text{Get}(\text{xpar}, \text{xm}_1, \text{xm}_2, \sigma_{l,\mathcal{P}}, \sigma_{l,\mathcal{W}})$  and the blind signature  $\sigma_{\text{BS}} \leftarrow \text{BS.U}_2(St, \text{bsm}_2)$ .

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

**Security.** We informally argue why Sweep-UC satisfies security for users, security for the sweeper, and user unlinkability. A formal analysis in the UC model can be found in Supplementary Material G. 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. Thus, in a case where users steal 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. Thus, the users must



have broken one-more unforgeability of BS. Finally, unlinkability follows from the blindness of BS and the usage of an anonymous channel.

## 7 Discussion

In this section, we discuss efficiency, practicality, and potential extensions of our results.

**Efficiency.** Both the communication and computational complexity of our protocol is dominated by the exchange and redeem protocols. For the generic constructions without cut-and-choose, the cost is clearly dominated by the costs of the NIZK that is used. Thus, we only go into detail for the constructions based on cut-and-choose for BLS and Schnorr/ECDSA signatures. 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  $\lambda$  evaluations of algorithm `reconst` (see Figure 3, Line 25). We can significantly reduce this to  $O(\lambda^2)$  operations using preprocessing techniques as explained in Supplementary Material I. 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.

**Experimental Evaluation.** To show efficiency in practice, we implemented a simple 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 in combination with the BLS blind signature scheme. Other cut-and-choose variants of our algorithms should be equally practical. More details of our implementation and experimental setup are given in Supplementary Material J. 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 setup 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 on a standard laptop. We expect that this can be optimized further. Also, note that in the case of an honest sweeper, algorithms `EXC.Get` and `RP.Redeem` terminate early and take less time (less than a second) than for a malicious sweeper.

**Redeem Cut-and-Choose for Arbitrary Signature Scheme.** In Section 5.2 we presented two redeem protocols based on a cut-and-choose technique, where the signature scheme `SIG` was instantiated respectively using BLS and Schnorr. On the other hand, our generic redeem protocol supports any signature scheme. We will briefly discuss how to achieve the same for cut-and-choose. The idea is similar to hybrid encryption. In this regard, we will use the BLS-based redeem protocol. Recall, that at the end of the protocol one gets a BLS signature for

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 BLS12-381 curve. For EXC.Get and RP.Redem, the honest case (HC), and the worst case for a malicious sweeper (WC) are listed.

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 [13] and add IBE ciphertexts to the promise. This particular construction for BLS was recently proposed by Döttling et al. [18] and called signature witness encryption (SWE). The primitive they propose allows encrypting an arbitrary message, proving any statement about the message using Bulletproofs [15], 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 new constructions of 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

1. CoinJoin - Bitcoin Forum. <https://bitcointalk.org/?topic=279249> (2013)
2. Submarine swap in lightning network (2018), <https://wiki.ion.radar.tech/tech/research/submarine-swap>
3. What is atomic swap and how to implement it (2019), <https://www.axiomadev.com/blog/what-is-atomic-swap-and-how-to-implement-it/>
4. Uniswap (2020), <https://uniswap.org/whitepaper.pdf>
5. Chia network faq (2022), <https://www.chia.net/faq/>
6. Digital currency donations for freedom convoy evading seizure by authorities (2022), <https://www.cbc.ca/news/canada/ottawa/freedom-convoy-cryptocurrency-asset-seizure-1.6389601>
7. Raiden network (2022), <https://raiden.network/>
8. Understanding bitcoin fungibility (2022), <https://river.com/learn/bitcoin-fungibility/>
9. Baum, C., David, B., Frederiksen, T.K.: P2DEX: Privacy-preserving decentralized cryptocurrency exchange. In: Sako, K., Tippenhauer, N.O. (eds.) ACNS 21, Part I. LNCS, vol. 12726, pp. 163–194. Springer, Heidelberg (Jun 2021)
10. Ben-Sasson, E., Chiesa, A., Garman, C., Green, M., Miers, I., Tromer, E., Virza, M.: Zerocash: Decentralized anonymous payments from bitcoin. In: 2014 IEEE

- Symposium on Security and Privacy. pp. 459–474. IEEE Computer Society Press (May 2014)
11. Bentov, I., Ji, Y., Zhang, F., Breidenbach, L., Daian, P., Juels, A.: Tesseract: Real-time cryptocurrency exchange using trusted hardware. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) ACM CCS 2019. pp. 1521–1538. ACM Press (Nov 2019)
  12. Boldyreva, A.: Threshold signatures, multisignatures and blind signatures based on the gap-Diffie-Hellman-group signature scheme. In: Desmedt, Y. (ed.) PKC 2003. LNCS, vol. 2567, pp. 31–46. Springer, Heidelberg (Jan 2003)
  13. Boneh, D., Franklin, M.K.: Identity-based encryption from the Weil pairing. In: Kilian, J. (ed.) CRYPTO 2001. LNCS, vol. 2139, pp. 213–229. Springer, Heidelberg (Aug 2001)
  14. Boneh, D., Lynn, B., Shacham, H.: Short signatures from the Weil pairing. In: Boyd, C. (ed.) ASIACRYPT 2001. LNCS, vol. 2248, pp. 514–532. Springer, Heidelberg (Dec 2001)
  15. Bünz, B., Bootle, J., Boneh, D., Poelstra, A., Wuille, P., Maxwell, G.: Bulletproofs: Short proofs for confidential transactions and more. In: 2018 IEEE Symposium on Security and Privacy. pp. 315–334. IEEE Computer Society Press (May 2018)
  16. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: 42nd FOCS. pp. 136–145. IEEE Computer Society Press (Oct 2001)
  17. Chaum, D.: Blind signatures for untraceable payments. In: Chaum, D., Rivest, R.L., Sherman, A.T. (eds.) CRYPTO’82. pp. 199–203. Plenum Press, New York, USA (1982)
  18. Döttling, N., Hanzlik, L., Magri, B., Wöhnig, S.: McFly: Verifiable encryption to the future made practical. Cryptology ePrint Archive, Report 2022/433 (2022), <https://eprint.iacr.org/2022/433>
  19. Dziembowski, S., Eckey, L., Faust, S.: FairSwap: How to fairly exchange digital goods. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) ACM CCS 2018. pp. 967–984. ACM Press (Oct 2018)
  20. Erwig, A., Faust, S., Hostáková, K., Maitra, M., Riahi, S.: Two-party adaptor signatures from identification schemes. In: Garay, J. (ed.) PKC 2021, Part I. LNCS, vol. 12710, pp. 451–480. Springer, Heidelberg (May 2021)
  21. Fischlin, M., Schröder, D.: On the impossibility of three-move blind signature schemes. In: Gilbert, H. (ed.) EUROCRYPT 2010. LNCS, vol. 6110, pp. 197–215. Springer, Heidelberg (May / Jun 2010)
  22. Glaeser, N., Maffei, M., Malavolta, G., Moreno-Sanchez, P., Tairi, E., Thyagarajan, S.A.: Foundations of coin mixing services. In: ACM CCS 2022 (to appear) (2022)
  23. Groth, J.: Rerandomizable and replayable adaptive chosen ciphertext attack secure cryptosystems. In: Naor, M. (ed.) TCC 2004. LNCS, vol. 2951, pp. 152–170. Springer, Heidelberg (Feb 2004)
  24. Heilman, E., Alshenibr, L., Baldimtsi, F., Scafuro, A., Goldberg, S.: TumbleBit: An untrusted bitcoin-compatible anonymous payment hub. In: NDSS 2017. The Internet Society (Feb / Mar 2017)
  25. Heilman, E., Baldimtsi, F., Goldberg, S.: Blindly signed contracts: Anonymous on-blockchain and off-blockchain bitcoin transactions. In: Clark, J., Meiklejohn, S., Ryan, P.Y.A., Wallach, D.S., Brenner, M., Rohloff, K. (eds.) FC 2016 Workshops. LNCS, vol. 9604, pp. 43–60. Springer, Heidelberg (Feb 2016)
  26. Herlihy, M.: Atomic cross-chain swaps (2018)
  27. Lai, R.W.F., Ronge, V., Ruffing, T., Schröder, D., Thyagarajan, S.A.K., Wang, J.: Omniring: Scaling private payments without trusted setup. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) ACM CCS 2019. pp. 31–48. ACM Press (Nov 2019)

28. Malavolta, G., Moreno-Sanchez, P., Kate, A., Maffei, M., Ravi, S.: Concurrency and privacy with payment-channel networks. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) ACM CCS 2017. pp. 455–471. ACM Press (Oct / Nov 2017)
29. Möser, M., Soska, K., Heilman, E., Lee, K., Heffan, H., Srivastava, S., Hogan, K., Hennessey, J., Miller, A., Narayanan, A., et al.: An empirical analysis of traceability in the monero blockchain. arXiv preprint arXiv:1704.04299 (2017)
30. Ober, M., Katzenbeisser, S., Hamacher, K.: Structure and anonymity of the bitcoin transaction graph. *Future internet* 5(2), 237–250 (2013)
31. Reid, F., Harrigan, M.: An analysis of anonymity in the bitcoin system. In: *Security and privacy in social networks*, pp. 197–223. Springer (2013)
32. Ron, D., Shamir, A.: Quantitative analysis of the full bitcoin transaction graph. In: *International Conference on Financial Cryptography and Data Security*. pp. 6–24. Springer (2013)
33. Ruffing, T., Moreno-Sanchez, P., Kate, A.: CoinShuffle: Practical decentralized coin mixing for bitcoin. In: Kutyłowski, M., Vaidya, J. (eds.) ESORICS 2014, Part II. LNCS, vol. 8713, pp. 345–364. Springer, Heidelberg (Sep 2014)
34. Ruffing, T., Moreno-Sanchez, P., Kate, A.: P2P mixing and unlinkable bitcoin transactions. In: *NDSS 2017*. The Internet Society (Feb / Mar 2017)
35. Santamaria Ortega, M.: The bitcoin transaction graph anonymity (2013)
36. Shamir, A.: How to share a secret. *Communications of the Association for Computing Machinery* 22(11), 612–613 (Nov 1979)
37. Tairi, E., Moreno-Sanchez, P., Maffei, M.: A<sup>2</sup>L: Anonymous atomic locks for scalability in payment channel hubs. In: *2021 IEEE Symposium on Security and Privacy*. pp. 1834–1851. IEEE Computer Society Press (May 2021)
38. Thyagarajan, S.A.K., Bhat, A., Malavolta, G., Döttling, N., Kate, A., Schröder, D.: Verifiable timed signatures made practical. In: Ligatti, J., Ou, X., Katz, J., Vigna, G. (eds.) ACM CCS 2020. pp. 1733–1750. ACM Press (Nov 2020)
39. Thyagarajan, S.A., Malavolta, G., Moreno-Sanchez, P.: Universal atomic swaps: Secure exchange of coins across all blockchains (2021), <https://ia.cr/2021/1612>

## Supplementary Material

### A Detailed Preliminaries

#### A.1 Digital Signatures

**Definition 7 (Signature Scheme).** A signature scheme SIG is a tuple  $\text{SIG} = (\text{Gen}, \text{Sig}, \text{Ver})$  of PPT algorithms with the following syntax:

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

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

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

**Definition 8 (Unique Signatures).** Let  $\text{SIG} = (\text{Gen}, \text{Sig}, \text{Ver})$  be a signature scheme. We say that  $\text{SIG}$  has unique signatures, if for every public key  $\text{pk}$  (not necessarily output by  $\text{Gen}$ ) and every message  $\text{m}$ , there is exactly one signature  $\sigma$  such that  $\text{Ver}(\text{pk}, \text{m}, \sigma) = 1$ .

**Definition 9 (Smoothness).** Let  $\text{SIG} = (\text{Gen}, \text{Sig}, \text{Ver})$  be a signature scheme. Assume that signatures have length  $\ell = \ell(\lambda)$  bits. We say that  $\text{SIG}$  is smooth, if for every public key  $\text{pk}$  (not necessarily output by  $\text{Gen}$ ) and every message  $\text{m}$ , the following probability is negligible:

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

**Definition 10 (Public Key Entropy).** Let  $\text{SIG} = (\text{Gen}, \text{Sig}, \text{Ver})$  be a signature scheme and  $f : \mathbb{N} \rightarrow \mathbb{R}$  be a function. We say that  $\text{SIG}$  has public key entropy  $f$ , if for all public keys  $\text{pk}_0$  the following holds

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

**Definition 11 (Unforgeability).** Consider a signature scheme  $\text{SIG} = (\text{Gen}, \text{Sig}, \text{Ver})$ . For any algorithm  $\mathcal{A}$ , consider the following game:

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

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

**Definition 12 (Strong Unforgeability).** Let  $\text{SIG} = (\text{Gen}, \text{Sig}, \text{Ver})$  be a signature scheme. For any algorithm  $\mathcal{A}$ , consider the following game:

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

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

## A.2 Blind Signatures

**Definition 13 (Blind Signature Scheme).** A (two-move) blind signature scheme  $\text{BS} = (\text{Gen}, \text{S}, \text{U}, \text{Ver})$  is a quadruple of PPT algorithms with the following syntax:

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

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

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

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

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

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

**Definition 14 (Unique Blind Signatures).** Let  $\text{BS} = (\text{Gen}, \text{S}, \text{U}, \text{Ver})$  be a blind signature scheme. We say that  $\text{BS}$  has unique signatures, if for every public key  $\text{pk}$  (not necessarily output by  $\text{Gen}$ ) and every message  $\text{m}$ , there is exactly one signature  $\sigma$  such that  $\text{Ver}(\text{pk}, \text{m}, \sigma) = 1$ .

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 [21], this is implied by the standard notion of blindness. It is sufficient for our purposes<sup>16</sup>.

**Definition 15 (Weak Blindness).** Let  $\text{BS} = (\text{Gen}, \text{S}, \text{U}, \text{Ver})$  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  $\text{pk}$  and messages  $\text{m}_0, \text{m}_1$ .
2. Run  $(\text{bsm}_1, \text{St}) \leftarrow \text{U}_1(\text{pk}, \text{m}_b)$  and give  $\text{bsm}_1$  to  $\mathcal{A}$ .
3. Get  $\text{bsm}_2$  from  $\mathcal{A}$  and run  $\sigma \leftarrow \text{U}_2(\text{St}, \text{bsm}_2)$ .
4. Give  $\text{Ver}(\text{pk}, \text{m}_b, \sigma)$  to  $\mathcal{A}$  and obtain a bit  $b'$  in return.
5. Output  $b'$ .

<sup>16</sup> 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.

We say that  $\text{BS}$  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 16 (One-More Unforgeability).** Let  $\text{BS} = (\text{Gen}, \text{S}, \text{U}, \text{Ver})$  be a blind signature scheme. For any algorithm  $\mathcal{A}$ , consider the following game:

1. Generate keys  $(\text{pk}, \text{sk}) \leftarrow \text{Gen}(1^\lambda)$ .
2. Let  $O$  be an oracle that on input  $\text{bsm}_1$  returns  $\text{bsm}_2 \leftarrow \text{BS.S}(\text{sk}, \text{bsm}_1)$ .
3. Run  $\mathcal{A}$  on input  $\text{pk}$  with access to oracle  $O$  and obtain  $(\mathbf{m}_1, \sigma_1), \dots, (\mathbf{m}_k, \sigma_k)$ .
4. Let  $\ell$  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  $\mathbf{m}_i \neq \mathbf{m}_j$ .
  - (c) For all  $i \in [k]$  we have  $\text{Ver}(\text{pk}, \mathbf{m}_i, \sigma_i) = 1$ .

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

### A.3 NP-Relations

**Definition 17 (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

$$\mathcal{L}_\lambda := \{ \text{stmt} \in \{0, 1\}^* \mid \exists \text{witn} \in \{0, 1\}^* : (\text{stmt}, \text{witn}) \in \mathcal{R}_\lambda \}.$$

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

- There exists a polynomial  $\text{poly}$ , such that for any  $\text{stmt} \in \mathcal{L}_\lambda$ , we have  $|\text{stmt}| \leq \text{poly}(\lambda)$ .
- 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  $\text{poly}'$  such that for all  $(\text{stmt}, \text{witn}) \in \mathcal{R}_\lambda$  we have  $|\text{witn}| \leq \text{poly}'(|\text{stmt}|)$ .

**Definition 18 (Hard NP-Relation).** Let  $\mathcal{R} = (\mathcal{R}_\lambda)_\lambda$  be an NP-relation. Assume that there is a PPT algorithm  $\mathcal{R}.\text{Gen}$  that on input  $1^\lambda$  outputs tuples  $(\text{stmt}, \text{witn}) \in \mathcal{R}_\lambda$ . We say that  $\mathcal{R}$  is hard relative to  $\mathcal{R}.\text{Gen}$  if for any PPT algorithm  $\mathcal{A}$  the following probability is negligible:

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

**Definition 19 (Unique NP-Relation).** Let  $\mathcal{R} = (\mathcal{R}_\lambda)_\lambda$  be an NP-relation. We say that  $\mathcal{R}$  is unique if for any  $\text{stmt} \in \mathcal{L}_\lambda$  there is exactly one  $\text{witn}$  such that  $(\text{stmt}, \text{witn}) \in \mathcal{R}_\lambda$ .

#### A.4 Adaptor Signatures

**Definition 20 (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  $aSIG = (\text{PreSig}, \text{Adapt}, \text{PreVer}, \text{Ext})$  of PPT algorithms with the following syntax:

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

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

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

**Definition 21 (Adaptability).** Let  $SIG$  be a signature scheme,  $\mathcal{R}$  an NP-relation, and  $aSIG = (\text{PreSig}, \text{Adapt}, \text{PreVer}, \text{Ext})$  be an adaptor signature scheme for  $SIG$  and  $\mathcal{R}$ . We say that  $aSIG$  satisfies adaptability, if for all messages  $\text{m}$ , pairs  $(\text{stmt}, \text{witn}) \in \mathcal{R}_\lambda$ , keys  $\text{pk}$  and pre-signatures  $\tilde{\sigma}$  the following implication holds:

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

**Definition 22 (Witness Extractability).** Let  $SIG$  be a signature scheme,  $\mathcal{R}$  an NP-relation, and  $aSIG = (\text{PreSig}, \text{Adapt}, \text{PreVer}, \text{Ext})$  be an adaptor signature scheme for  $SIG$  and  $\mathcal{R}$ . For any algorithm  $\mathcal{A}$  consider the following game:

1. Sample keys  $(\text{pk}, \text{sk}) \leftarrow \text{Gen}(1^\lambda)$  and initialize  $\mathcal{Q} := \emptyset$ .
2. Let  $SIG, \text{PRESIG}$  be oracles, defined as follows:
  - $SIG(\text{m})$ : Set  $\mathcal{Q} := \mathcal{Q} \cup \{\text{m}\}$  and return  $\text{Sig}(\text{sk}, \text{m})$ .
  - $\text{PRESIG}(\text{m}, \text{stmt})$ : Set  $\mathcal{Q} := \mathcal{Q} \cup \{\text{m}\}$ . Then, return  $\text{PreSig}(\text{sk}, \text{m}, \text{stmt})$ .
3. Run  $\mathcal{A}$  on input  $\text{pk}$  with access to  $SIG, \text{PRESIG}$ . Obtain  $(\text{m}^*, \text{stmt}^*)$  in return.
4. Compute  $\tilde{\sigma} \leftarrow \text{PreSig}(\text{sk}, \text{m}^*, \text{stmt}^*)$  and give  $\tilde{\sigma}$  to  $\mathcal{A}$ . Obtain  $\sigma^*$  in return.
5. Run  $\text{witn} := \text{Ext}(\tilde{\sigma}, \sigma^*)$ .
6. Output 1 if  $\text{Ver}(\text{pk}, \text{m}^*, \sigma^*)$ ,  $\text{m}^* \notin \mathcal{Q}$ , and  $(\text{stmt}^*, \text{witn}) \notin \mathcal{R}_\lambda$ . 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 aEUF-CMA is weaker than the standard notion (e.g. in [20]) in a sense that we do not give the adversary a pre-signature on the message  $\text{m}^*$ .



**Definition 23 (Adaptor Unforgeability).** Let  $\text{SIG}$  be a signature scheme,  $\mathcal{R}$  an NP-relation, and  $\text{aSIG} = (\text{PreSig}, \text{Adapt}, \text{PreVer}, \text{Ext})$  be an adaptor signature scheme for  $\text{SIG}$  and  $\mathcal{R}$ . For any algorithm  $\mathcal{A}$  consider the following game:

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

We say that  $\text{aSIG}$  is aEUF-CMA secure, if for all PPT algorithms  $\mathcal{A}$ , the probability that the above game outputs 1 is 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 [20].

**Definition 24 (Well Adapted Signatures).** Let  $\text{SIG}$  be a signature scheme,  $\mathcal{R}$  an NP-relation, and  $\text{aSIG} = (\text{PreSig}, \text{Adapt}, \text{PreVer}, \text{Ext})$  be an adaptor signature scheme for  $\text{SIG}$  and  $\mathcal{R}$ . We say that  $\text{aSIG}$  has well adapted signatures, if for all keys  $(\text{pk}, \text{sk}) \in \text{Gen}(1^\lambda)$ , all messages  $\text{m}$ , and all pairs  $(\text{stmt}, \text{witn}) \in \mathcal{R}_\lambda$ , the following distributions  $\mathcal{D}_1$  and  $\mathcal{D}_2$  are the same:

$$\mathcal{D}_1 := \left\{ (\text{pk}, \text{sk}, \text{m}, \sigma) \mid \begin{array}{l} \tilde{\sigma} \leftarrow \text{PreSig}(\text{sk}, \text{m}, \text{stmt}), \\ \sigma := \text{Adapt}(\text{pk}, \tilde{\sigma}, \text{witn}) \end{array} \right\},$$

$$\mathcal{D}_2 := \{ (\text{pk}, \text{sk}, \text{m}, \sigma) \mid \sigma \leftarrow \text{Sig}(\text{sk}, \text{m}) \}.$$

## A.5 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  $\text{PSim}$  multiple times without introducing conflicts due to random oracle programming.

**Definition 25 (Non-Interactive Proof System).** Let  $\mathcal{R}$  be an NP-relation. A non-interactive proof system for  $\mathcal{R}$  is a tuple  $\text{PS} = (\text{PProve}, \text{PVer})$  of PPT algorithms with the following syntax:

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

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

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

**Definition 26 (Soundness).** Let  $\mathcal{R}$  be an NP-relation and  $\text{PS} = (\text{PProve}, \text{PVer})$  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:

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

**Definition 27 (Zero-Knowledge).** Consider an NP-relation  $\mathcal{R}$  and a non-interactive proof system  $\text{PS} = (\text{PProve}, \text{PVer})$  for  $\mathcal{R}$ . We say that PS is zero-knowledge, if there exists a PPT algorithm  $\text{PSim}$ , that is allowed to program random oracles, such that for any  $(\text{stmt}, \text{witn}) \in \mathcal{R}_\lambda$ , the following distributions  $\mathcal{D}_1$  and  $\mathcal{D}_2$  are statistically close:

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

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.

## A.6 Computational Assumptions

**Definition 28 (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:

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

**Definition 29 (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:

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

## A.7 Universal Composability Framework

In the universal composability (UC) framework [16], all parties are modelled as interactive Turing machines. For an environment  $\mathcal{Z}$ , an adversary  $\mathcal{A}$ , a protocol  $\pi$ , and a functionality  $\mathcal{G}$ , we write  $\text{Hybrid}_{\mathcal{Z}, \mathcal{A}, \pi}^{\mathcal{G}}$  to denote the output distribution of  $\mathcal{Z}$  in the execution with protocol  $\pi$  and adversary  $\mathcal{A}$ . Here,  $\pi$  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  $\text{Ideal}_{\mathcal{Z}, \mathcal{S}, \mathcal{F}}$  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 30 (UC Security).** *A protocol  $\pi$  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  $\text{Hybrid}_{\mathcal{Z},\mathcal{A},\pi}^{\mathcal{G}}$  and  $\text{Ideal}_{\mathcal{Z},\mathcal{S},\mathcal{F}}$  are computationally indistinguishable.*

## B Omitted Definitions for Exchange Protocols

**Definition 31 (Well Distributed Signatures).** *Let  $\text{EXC} = (\text{Setup}, \text{Buy}, \text{Sell}, \text{Get})$  be an exchange for  $\text{SIG}$  and  $\text{BS}$  as in Definition 1. We say that  $\text{EXC}$  has well distributed signatures, if for all transactions  $\text{tx}$ , all messages  $\text{sn}$ , all keys  $(\text{pk}_{\text{BS}}, \text{sk}_{\text{BS}}) \in \text{BS.Gen}(1^\lambda)$ , all  $(\text{pk}_b, \text{sk}_b) \in \text{SIG.Gen}(1^\lambda)$ , all  $(\text{pk}_s, \text{sk}_s) \in \text{SIG.Gen}(1^\lambda)$ , we have, the following distributions  $\mathcal{D}_1$  and  $\mathcal{D}_2$  are the same:*

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

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

## C Omitted Constructions of Exchange Protocols

### C.1 Generic Construction for Unique Signatures.

Let  $\text{SIG} = (\text{SIG.Gen}, \text{SIG.Sig}, \text{SIG.Ver})$  be a signature scheme and  $\text{BS} = (\text{BS.Gen}, \text{BS.S}, \text{BS.U}, \text{BS.Ver})$  be a two-move blind signature scheme. We assume that  $\text{SIG}$  has unique signatures, and give a generic construction of an exchange protocol  $\text{EXC}_u[\text{SIG}, \text{BS}, \text{PS}] = (\text{Setup}, \text{Buy}, \text{Sell}, \text{Get})$  for  $\text{SIG}$  and  $\text{BS}$ . 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  $\text{bsm}_2$  sent in signing interactions of  $\text{BS}$ . Further, let  $\ell_2 = \ell_2(\lambda)$  denote an upper bound on the number of random bits that algorithm  $\text{S}$  uses. We make use of a random oracle  $\text{H} : \{0, 1\}^* \rightarrow \{0, 1\}^{\ell_1}$  and a NIZK  $\text{PS} = (\text{PProve}, \text{PVer})$  with zero-knowledge simulator  $\text{PS.Sim}$  for the relation

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

The scheme  $\text{EXC}_u[\text{SIG}, \text{BS}, \text{PS}]$  is formally presented in Figure 4. Completeness follows by inspection. As  $\text{SIG}$  has unique signatures,  $\text{EXC}_u[\text{SIG}, \text{BS}, \text{PS}]$  has well distributed signatures. Security proofs are given in Supplementary Material D.

<b>Setup</b> ( $x\text{par}, \text{sk}_{\text{BS}}, \text{sk}_s$ ) 01 $\rho \leftarrow_s \{0, 1\}^{\ell_2}$ 02 $\text{bsm}_2 := \text{S}(\text{sk}_{\text{BS}}, \text{bsm}_1; \rho)$ 03 $\sigma_s \leftarrow \text{SIG.Sig}(\text{sk}_s, \text{tx})$ 04 $\text{ct} := \text{H}(\sigma_s) \oplus \text{bsm}_2$ 05 $\text{stmt} := (\text{pk}_{\text{BS}}, \text{pk}_s, \text{tx}, \text{bsm}_1, \text{ct})$ 06 $\text{witn} := (\sigma_s, \text{sk}_{\text{BS}}, \rho)$ 07 $\pi \leftarrow \text{PProve}(\text{stmt}, \text{witn})$ 08 $x\text{m}_1 := (\text{ct}, \pi)$ 09 <b>return</b> ( $x\text{m}_1, St := x\text{par}$ )	<b>Buy</b> ( $x\text{par}, \text{sk}_b, x\text{m}_1 = (\text{ct}, \pi)$ ) 10 $\text{stmt} := (\text{pk}_{\text{BS}}, \text{pk}_s, \text{tx}, \text{bsm}_1, \text{ct})$ 11 <b>if</b> $\text{PVer}(\text{stmt}, \pi) = 0$ : <b>return</b> $\perp$ 12 <b>return</b> $x\text{m}_2 := \sigma_b \leftarrow \text{SIG.Sig}(\text{sk}_b, \text{tx})$  <b>Sell</b> ( $St, x\text{m}_2 = \sigma_b$ ) 13 <b>if</b> $\text{SIG.Ver}(\text{pk}_b, \text{tx}, \sigma_b) = 0$ : <b>return</b> $\perp$ 14 <b>return</b> $\sigma_b$  <b>Get</b> ( $x\text{par}, x\text{m}_1, x\text{m}_2, \sigma_b, \sigma_s$ ) 15 <b>return</b> $\text{bsm}_2 := \text{ct} \oplus \text{H}(\sigma_s)$
--	---

**Fig. 4.** The exchange protocol  $\text{EXC}_u[\text{SIG}, \text{BS}, \text{PS}] = (\text{Setup}, \text{Buy}, \text{Sell}, \text{Get})$  for a unique signature scheme  $\text{SIG}$  and a blind signature scheme  $\text{BS}$ , where  $\text{PS} = (\text{PProve}, \text{PVer})$  is a NIZK for  $\mathcal{R}$ , and  $\text{H} : \{0, 1\}^* \rightarrow \{0, 1\}^{\ell_1}$  is a random oracle.

**Lemma 5.** *If  $\text{SIG}$  has unique signatures,  $\text{SIG}$  is EUF-CMA secure, and  $\text{PS}$  is sound, then  $\text{EXC}_u[\text{SIG}, \text{BS}, \text{PS}]$  is secure against malicious sellers.*

**Lemma 6.** *If  $\text{SIG}$  has unique signatures,  $\text{SIG}$  is EUF-CMA secure, and  $\text{PS}$  is zero-knowledge, then  $\text{EXC}_u[\text{SIG}, \text{BS}, \text{PS}]$  is secure against malicious buyers.*

## C.2 Generic Construction for Adaptor Signatures

We give a construction of an exchange protocol  $\text{EXC}_a[\text{SIG}, \text{aSIG}, \text{BS}, \text{PS}]$  for a signature scheme  $\text{SIG}$  supporting adaptor signatures. Concretely, let  $\mathcal{R}'$  be a unique **NP**-relation that is hard relative to  $\mathcal{R}'.\text{Gen}$ . Let  $\text{aSIG} = (\text{PreSig}, \text{Adapt}, \text{PreVer}, \text{Ext})$  be an adaptor signature for  $\text{SIG}$  and  $\mathcal{R}'$ . Let  $\ell_1 = \ell_1(\lambda)$  denote an upper bound on the bit length of messages  $\text{bsm}_2$  sent in signing interactions of  $\text{BS}$ . 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  $H : \{0, 1\}^* \rightarrow \{0, 1\}^{\ell_1}$  and a NIZK  $\text{PS} = (\text{PProve}, \text{PVer})$  with zero-knowledge simulator  $\text{PS.Sim}$  for the relation

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

The scheme  $\text{EXC}_a[\text{SIG}, \text{aSIG}, \text{BS}, \text{PS}]$  is presented formally in Figure 5. Completeness follows by the uniqueness of  $\mathcal{R}'$ . The scheme has well distributed signatures if  $\text{aSIG}$  has well adapted signatures. We give the security proofs in Supplementary Material D.

**Lemma 7.** *If  $\text{aSIG}$  is witness extractable and  $\text{aEUF-CMA}$  secure,  $\mathcal{R}'$  is unique, and  $\text{PS}$  is sound, then  $\text{EXC}_a[\text{SIG}, \text{aSIG}, \text{BS}, \text{PS}]$  is secure against malicious sellers.*

**Lemma 8.** *If  $\text{aSIG}$  satisfies adaptability,  $\mathcal{R}'$  is hard relative to  $\mathcal{R}'.\text{Gen}$ , and  $\text{PS}$  is zero-knowledge, then  $\text{EXC}_a[\text{SIG}, \text{aSIG}, \text{BS}, \text{PS}]$  is secure against malicious buyers.*

<u>Setup(xpar, sk<sub>BS</sub>, sk<sub>s</sub>)</u>	<u>Buy(xpar, sk<sub>b</sub>, xm<sub>1</sub> = (stmt', ct, π))</u>
01 $\rho \leftarrow_{\$} \{0, 1\}^{\ell_2}$	11 $\text{stmt} := (\text{pk}_{\text{BS}}, \text{bsm}_1, \text{stmt}', \text{ct})$
02 $\text{bsm}_2 := S(\text{sk}_{\text{BS}}, \text{bsm}_1; \rho)$	12 <b>if</b> $\text{PVer}(\text{stmt}, \pi) = 0$ : <b>return</b> $\perp$
03 $(\text{stmt}', \text{witn}') \leftarrow \mathcal{R}'.\text{Gen}(1^\lambda)$	13 <b>return</b> $\text{xm}_2 := \tilde{\sigma}_b \leftarrow \text{PreSig}(\text{sk}_b, \text{tx}, \text{stmt}')$
04 $\text{ct} := H(\text{witn}') \oplus \text{bsm}_2$	<u>Sell(<math>St = \text{witn}'</math>, <math>\text{xm}_2 = \tilde{\sigma}_b</math>)</u>
05 $\text{stmt} := (\text{pk}_{\text{BS}}, \text{bsm}_1, \text{stmt}', \text{ct})$	14 <b>if</b> $\text{PreVer}(\text{pk}_b, \text{tx}, \text{stmt}', \tilde{\sigma}_b) = 0$ : <b>return</b> $\perp$
06 $\text{witn} := (\text{sk}_{\text{BS}}, \text{witn}', \rho)$	15 <b>return</b> $\sigma_b := \text{Adapt}(\text{pk}_b, \tilde{\sigma}_b, \text{witn}')$
07 $\pi \leftarrow \text{PProve}(\text{stmt}, \text{witn})$	<u>Get(xpar, xm<sub>1</sub>, xm<sub>2</sub>, σ<sub>b</sub>, σ<sub>s</sub>)</u>
08 $\text{xm}_1 := (\text{stmt}', \text{ct}, \pi)$	16 <b>let</b> $\text{xm}_1 = (\text{stmt}', \text{ct}, \pi)$ , $\text{xm}_2 = \tilde{\sigma}_b$
09 $St := \text{witn}'$	17 $\text{witn}' := \text{Ext}(\tilde{\sigma}_b, \sigma_b)$
10 <b>return</b> $(\text{xm}_1, St)$	18 <b>return</b> $\text{bsm}_2 := \text{ct} \oplus H(\text{witn}')$

**Fig. 5.** The exchange protocol  $\text{EXC}_a[\text{SIG}, \text{aSIG}, \text{BS}, \text{PS}] = (\text{Setup}, \text{Buy}, \text{Sell}, \text{Get})$  for a signature scheme  $\text{SIG}$  and an associated adaptor signature scheme  $\text{aSIG}$ , and a blind signature scheme  $\text{BS}$ . Here,  $\text{PS} = (\text{PProve}, \text{PVer})$  is a NIZK for  $\mathcal{R}$ , and  $H : \{0, 1\}^* \rightarrow \{0, 1\}^{\ell_1}$  is a random oracle.

### C.3 Construction for Adaptor Signatures using Cut-and-Choose

We give a construction of an exchange protocol using a cut-and-choose technique. We assume that the signature scheme  $\text{SIG}$  has an associated adaptor signature scheme  $\text{aSIG} = (\text{PreSig}, \text{Adapt}, \text{PreVer}, \text{Ext})$  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  $q$ . The blind signature scheme  $\text{BS} = (\text{BS.Gen}, \text{BS.S}, \text{BS.U}, \text{BS.Ver})$  is the BLS blind signature scheme. It is defined over cyclic groups  $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$  of prime order  $p$  with respective generators  $g_1 \in \mathbb{G}_1, g_2 \in \mathbb{G}_2$ , and  $e(g_1, g_2) \in \mathbb{G}_T$ , where  $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$  is a pairing. For completeness, we recall BLS (blind) signatures in Supplementary Material H. Let  $\ell = \ell(\lambda)$  denote an upper bound on the bit length of messages  $\text{bsm}_2$  sent in signing interactions of  $\text{BS}$ . We make use of random oracles  $\text{H} : \{0, 1\}^* \rightarrow \{0, 1\}^\ell$  and  $\text{H}_c : \{0, 1\}^* \rightarrow \{0, 1\}^\lambda$ . The scheme is called  $\text{EXC}_a^{\text{cc}}[\text{SIG}, \text{aSIG}, \text{BS}]$  and given in Figure 6. The security proofs are given in Supplementary Material D.

**Lemma 9.** *Assume that  $\text{aSIG}$  is witness extractable and  $\text{aEUF-CMA}$  secure. Then the exchange protocol  $\text{EXC}_a^{\text{cc}}[\text{SIG}, \text{aSIG}, \text{BS}]$  is secure against malicious sellers.*

**Lemma 10.** *Assume that  $\text{aSIG}$  satisfies adaptability and the DLOG assumption holds in  $\mathbb{G}$ . Then the exchange protocol  $\text{EXC}_a^{\text{cc}}[\text{SIG}, \text{aSIG}, \text{BS}]$  is secure against malicious buyers.*

```

Setup( $xpar = (\mathbf{pk}_{BS}, \mathbf{bsm}_1, \mathbf{pk}_b, \mathbf{pk}_s, \mathbf{tx}), \mathbf{sk}_{BS}, \mathbf{sk}_s$ )
01  $y \leftarrow_{\mathcal{S}} \mathbb{Z}_q, Y := g^y$ 
// Share  $\mathbf{bsm}_2$  and  $y$ 
02  $r_1, \dots, r_\lambda \leftarrow_{\mathcal{S}} \mathbb{Z}_p, r'_1, \dots, r'_\lambda \leftarrow_{\mathcal{S}} \mathbb{Z}_q$ 
03  $f(X) := \mathbf{sk}_{BS} + \sum_{j=1}^{\lambda} r_j \cdot X^j \in \mathbb{Z}_p[X], f'(X) := y + \sum_{j=1}^{\lambda} r'_j \cdot X^j \in \mathbb{Z}_q[X]$ 
04 for  $j \in [2\lambda]$  :  $\mathbf{sk}_j := f(j), y_j := f'(j), \mathbf{bsm}_{2,j} \leftarrow \mathcal{S}(\mathbf{sk}_j, \mathbf{bsm}_1)$ 
05 for  $j \in [\lambda]$  :  $\mathbf{coeff}_j := g_2^{r_j}, \mathbf{coeff}'_j := g^{r'_j}$ 
// Encrypt  $\mathbf{bsm}_{2,j}$  with  $y_j$ 
06 for  $j \in [2\lambda]$  :  $\mathbf{ct}_j := \mathcal{H}(y_j) \oplus \mathbf{bsm}_{2,j}$ 
// Cut-and-choose
07  $\mathbf{xm}_{1,1} := (Y, (\mathbf{ct}_j)_{j \in [2\lambda]}, (\mathbf{coeff}_j, \mathbf{coeff}'_j)_{j \in [\lambda]})$ 
08  $b_0 \dots b_{\lambda-1} := \mathcal{H}_c(\mathbf{xm}_{1,1}), \mathbf{for } j \in [\lambda] : k_j := 2j - b_{j-1}$ 
09 return  $(\mathbf{xm}_1 := (\mathbf{xm}_{1,1}, \mathbf{xm}_{1,2} := (y_{k_j})_{j \in [\lambda]}), St := y)$ 

Buy( $xpar = (\mathbf{pk}_{BS}, \mathbf{bsm}_1, \mathbf{pk}_b, \mathbf{pk}_s, \mathbf{tx}), \mathbf{sk}_b, \mathbf{xm}_1 = (\mathbf{xm}_{1,1}, \mathbf{xm}_{1,2})$ )
// Verify cut-and-choose
10  $b_0 \dots b_{\lambda-1} := \mathcal{H}_c(\mathbf{xm}_{1,1})$ 
11 for  $j \in [\lambda]$  :
12  $k_j := 2j - b_{j-1}, \mathbf{pk}_{BS,k_j} := \mathbf{pk}_{BS} \cdot \prod_{i=1}^{\lambda} (\mathbf{coeff}_i)^{k_j^i}, Y_{k_j} := Y \cdot \prod_{i=1}^{\lambda} (\mathbf{coeff}'_i)^{k_j^i}$ 
13  $\mathbf{bsm}_{2,k_j} := \mathbf{ct}_{k_j} \oplus \mathcal{H}(y_{k_j})$ 
14 if  $e(\mathbf{bsm}_1, \mathbf{pk}_{BS,k_j}) \neq e(\mathbf{bsm}_{2,k_j}, g_2) \vee Y_{k_j} \neq g^{y_{k_j}} : \mathbf{return } \perp$ 
// Return a pre-signature for  $Y$ 
15 return  $\mathbf{xm}_2 := \tilde{\sigma}_b \leftarrow \text{PreSig}(\mathbf{sk}_b, \mathbf{tx}, Y)$ 

Sell( $St = y, \mathbf{xm}_2 = \tilde{\sigma}_b$ )
16 if  $\text{PreVer}(\mathbf{pk}_b, \mathbf{tx}, g^y, \tilde{\sigma}_b) = 0 : \mathbf{return } \perp$ 
17 return  $\sigma_b := \text{Adapt}(\mathbf{pk}_b, \tilde{\sigma}_b, y)$ 

Get( $xpar = (\mathbf{pk}_{BS}, \mathbf{bsm}_1, \mathbf{pk}_b, \mathbf{pk}_s, \mathbf{tx}), \mathbf{xm}_1, \mathbf{xm}_2 = (\tilde{\sigma}_b, \sigma_b, \sigma_s)$ )
18  $y := \text{Ext}(\tilde{\sigma}_b, \sigma_b), b_0 \dots b_{\lambda-1} := \mathcal{H}_c(\mathbf{xm}_{1,1})$ 
// Reconstruct all shares
19 for  $j \in [\lambda]$  :  $k_j := 2j - b_{j-1}, \bar{k}_j := 2j - (1 - b_{j-1}), \mathbf{bsm}_{2,k_j} := \mathbf{ct}_{k_j} \oplus \mathcal{H}(y_{k_j})$ 
20  $f'(X) := \text{reconst}_q((0, y), (k_j, y_{k_j})_{j \in [\lambda]})$ 
// Find a valid share
21  $w := 0$ 
22 for  $j \in [\lambda]$  :
23  $y_{\bar{k}_j} := f'(\bar{k}_j), \mathbf{bsm}_{2,\bar{k}_j} := \mathbf{ct}_{\bar{k}_j} \oplus \mathcal{H}(y_{\bar{k}_j})$ 
24  $\mathbf{pk}_{BS,\bar{k}_j} := \mathbf{pk}_{BS} \cdot \prod_{i \in [\lambda]} (\mathbf{coeff}_i)^{\bar{k}_j^i}$ 
25 if  $e(\mathbf{bsm}_1, \mathbf{pk}_{BS,\bar{k}_j}) = e(\mathbf{bsm}_{2,\bar{k}_j}, g_2) : w := \bar{k}_j$ 
26 if  $w = 0 : \mathbf{return } \perp$ 
// Reconstruct  $\mathbf{bsm}_2$ 
27 return  $\mathbf{bsm}_2 := \text{reconst}_{g_1,0}((w, \mathbf{bsm}_{2,w}), (k_j, \mathbf{bsm}_{2,k_j})_{j \in [\lambda]})$ 

```

**Fig. 6.** The exchange protocol  $\text{EXC}_a^{\text{cc}}[\text{SIG}, \text{aSIG}, \text{BS}] = (\text{Setup}, \text{Buy}, \text{Sell}, \text{Get})$  for a signature scheme SIG and an associated adaptor signature scheme aSIG, and blind BLS signature scheme BS. Here,  $\mathcal{H} : \{0, 1\}^* \rightarrow \{0, 1\}^\ell$  and  $\mathcal{H}_c : \{0, 1\}^* \rightarrow \{0, 1\}^\lambda$  are random oracles and  $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$  is a pairing.

## D 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.

### D.1 Proofs for the Construction for Adaptor Signatures

*Proof (of Lemma 7 (Mal. Seller - Adaptor Signature)).* The proof is very similar to the proof of Lemma 5. Consider an adversary  $\mathcal{A}$  against the security of  $\text{EXC}_a[\text{SIG}, \text{aSIG}, \text{BS}, \text{PS}]$  against malicious sellers. We define three events in the security game, following the three possible ways  $\mathcal{A}$  can win.

- $\text{win}_1$ : This occurs if the security game outputs 1 and  $\text{tx} \neq \text{tx}'$ .
- $\text{win}_2$ : This occurs if the security game outputs 1,  $\text{tx} = \text{tx}'$  and  $\text{xm}_2 = \perp$ .
- $\text{win}_3$ : This occurs if the security game outputs 1,  $\text{tx} = \text{tx}'$ ,  $\text{xm}_2 \neq \perp$ , and  $\text{BS.Ver}(\text{pk}_{\text{BS}}, \text{sn}, \sigma_{\text{BS}}) = 0$ .

First, we bound the probability of  $\text{win}_1 \vee \text{win}_2$ . Intuitively if one of the events occurs, the adversary came up with a valid signature  $\sigma_b$  for a message  $\text{tx}'$ , for which the game did not compute a signature or pre-signature before. Formally, we give a reduction that runs in the aEUF-CMA security game of aSIG. The reduction gets  $\text{pk}$  as input and access to a signing oracle SIG and a pre-signing oracle PRESIG. It runs  $\mathcal{A}$  and obtains a public key  $\text{pk}_{\text{BS}}$  and a message  $\text{sn}$  from  $\mathcal{A}$ . Then, it runs  $(\text{bsm}_1, St) \leftarrow U_1(\text{pk}_{\text{BS}}, \text{sn})$ . It sets  $\text{pk}_b := \text{pk}$ . Next, it gives  $\text{pk}_b$  and  $\text{bsm}_1$  to  $\mathcal{A}$ , which outputs a key  $\text{pk}_s$ , a transaction  $\text{tx}$  and a message  $\text{xm}_1$ . If  $\text{xm}_1 = \perp$  or  $\text{xm}_1 = (\text{stmt}', \text{ct}, \pi)$  but  $\text{PVer}(\text{stmt}, \pi) = 0$  for  $\text{stmt} := (\text{pk}_{\text{BS}}, \text{bsm}_1, \text{stmt}', \text{ct})$ , the reduction sets  $\text{xm}_2 := \perp$ . Otherwise, it uses the oracle PRESIG as  $\tilde{\sigma}_b \leftarrow \text{PRESIG}(\text{tx}, \text{stmt}')$  and sets  $\text{xm}_2 := \tilde{\sigma}_b$ . The reduction gives  $\text{xm}_2$  to  $\mathcal{A}$  and obtains  $\text{tx}'$ ,  $\sigma_b$ , and  $\sigma_s$  in return. If  $\text{win}_1 \vee \text{win}_2$  occurs, it returns  $(\text{tx}', \sigma_b)$  to its game. Otherwise, it aborts. It is clear that the reduction perfectly simulates the game for  $\mathcal{A}$ . Also, note that the pair  $(\text{tx}', \sigma_b)$  that the reduction outputs in the end is valid, i.e.  $\text{SIG.Ver}(\text{pk}, \text{tx}', \sigma_b) = 1$ , by definition of  $\text{win}_1 \vee \text{win}_2$ . Further, note that if  $\text{win}_1$  occurs, the reduction did only query oracle PRESIG on input  $\text{tx} \neq \text{tx}'$ , and not on input  $\text{tx}'$ . Similarly, if  $\text{win}_2$  occurs, the reduction did not query PRESIG at all. In both cases, the reduction did never query oracle SIG. Therefore, the probability of  $\text{win}_1 \vee \text{win}_2$  can be upper bounded by the probability that the reduction wins the aEUF-CMA game. This is negligible by assumption.

It remains to bound the probability of event  $\text{win}_3$ . To do so, we partition  $\text{win}_3$  into two events. Let  $\text{xm}_1 = (\text{stmt}', \text{ct}, \pi)$  and  $\text{xm}_2 = \tilde{\sigma}_b$  be as in the security game against malicious sellers.

- $\text{win}_{3,1}$ : This event occurs, if  $\text{win}_3$  occurs and for  $\text{witn}' := \text{Ext}(\tilde{\sigma}_b, \sigma_b)$  we have  $(\text{stmt}', \text{witn}') \notin \mathcal{R}'$ .



- $\text{win}_{3,2}$ : This event occurs, if  $\text{win}_3$  occurs and for  $\text{witr}' := \text{Ext}(\tilde{\sigma}_b, \sigma_b)$  we have  $(\text{stmt}', \text{witr}') \in \mathcal{R}'$ .

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

We start with event  $\text{win}_{3,1}$ . 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  $\text{pk}$  as input and access to oracles SIG and PRESIG. It runs  $\mathcal{A}$  and obtains a public key  $\text{pk}_{\text{BS}}$  and a message  $\text{sn}$  from  $\mathcal{A}$ . Next, it runs  $(\text{bsm}_1, St) \leftarrow \text{U}_1(\text{pk}_{\text{BS}}, \text{sn})$ , sets  $\text{pk}_b := \text{pk}$ , and gives  $\text{pk}_b$  and  $\text{bsm}_1$  to  $\mathcal{A}$ , which outputs a key  $\text{pk}_s$ , a transaction  $\text{tx}$  and a message  $\text{xm}_1$ . If  $\text{xm}_1 = \perp$  or  $\pi$  does not verify, the reduction aborts. Otherwise, it parses  $\text{xm}_1 = (\text{stmt}', \text{ct}, \pi)$  and outputs  $(\text{tx}, \text{stmt}')$  to its game. It obtains a pre-signature  $\tilde{\sigma}$  in return and sets  $\text{xm}_2 := \tilde{\sigma}_b := \tilde{\sigma}$ . Then, the reduction passes  $\text{xm}_2$  to  $\mathcal{A}$  and obtains  $\text{tx}'$ ,  $\sigma_b$ , and  $\sigma_s$  in return. If  $\text{win}_{3,1}$  occurs, it outputs  $\sigma_b$  to its game. It is easy to see that the witness extractability game outputs 1 if event  $\text{win}_{3,1}$  occurs. Especially, the reduction did not use the oracles SIG and PRESIG at all.

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

$$\text{ct} \oplus \text{H}(\text{witr}'') = \text{BS.S}(\text{sk}_{\text{BS}}, \text{bsm}_1; \rho).$$

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

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

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

*Proof (of Lemma 8 (Mal. Buyer - Adaptor Signature)).* We give algorithms  $\text{Sim}_1$ ,  $\text{Sim}_{\text{RO}}$ ,  $\text{Sim}_2$ ,  $\text{Sim}_3$ , and then we show indistinguishability. The algorithms keep a list  $L$  that holds tuples  $(\text{tx}, \text{stmt}', \text{witr}', \text{pk}_s, \text{ct})$ . Algorithm  $\text{Sim}_1(\text{xpar}, \text{sk}_s)$  is as follows:

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

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

It remains to show that algorithms  $\text{Sim}_1, \text{Sim}_{RO}, \text{Sim}_2, \text{Sim}_3$  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  $(\text{pk}_{BS}, \text{sk}_{BS})$  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  $(\text{pk}_s, \text{sk}_s) \leftarrow \text{SIG.Gen}(1^\lambda)$ , gives  $\text{pk}_s$  to  $\mathcal{A}$  and obtains a key  $\text{pk}_b$ , a transaction  $\text{tx}$ , and a message  $\text{bsm}_1$  in return. Then, it runs algorithm  $\text{Setup}$ . Concretely, it computes  $\text{bsm}_2$ , samples  $\text{witn}'$  and  $\text{stmt}'$ , defines ciphertext  $\text{ct}$ , and computes a proof  $\pi$  as in the scheme. Then, it sets  $\text{xm}_1 := (\text{stmt}', \text{ct}, \pi)$  and sends  $\text{xm}_1$  to  $\mathcal{A}$ . The adversary responds with a message  $\text{xm}_2$ . If  $\text{xm}_2 = \tilde{\sigma}_b$  satisfies  $\text{PreVer}(\text{pk}_b, \text{tx}, \text{stmt}', \tilde{\sigma}_b) = 1$ , the game computes  $\sigma_s$  using  $\text{sk}_s$  and  $\sigma_b$  via  $\sigma_b := \text{Adapt}(\text{pk}_b, \tilde{\sigma}_b, \text{witn}')$ . 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  $\pi$  in message  $\text{xm}_1$  is computed by oracle  $O^*$ . Recall that before, it was computed via  $\pi \leftarrow \text{PProve}(\text{stmt}, \text{witn})$ , where  $\text{stmt}$  and  $\text{witn}$  are as in algorithm  $\text{Setup}$ . In game  $\mathbf{G}_1$ , we simulate it using the zero-knowledge simulator  $\text{PS.PSim}$  via  $\pi \leftarrow \text{PSim}(\text{stmt})$ . Games  $\mathbf{G}_0$  and  $\mathbf{G}_1$  are indistinguishable by the zero-knowledge property of  $\text{PS}$ .

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

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

Here,  $Q$  denotes the number of queries to oracles  $O^*$ , and  $\text{bad}_{1,i}$  (resp.  $\text{bad}_{2,i}$ ) denotes the event that  $\text{bad}_1$  (resp.  $\text{bad}_2$ ) occurs for the entry in  $L$  that is inserted in the  $i$ th query to  $O^*$ . As  $Q$  is polynomially bounded, it is sufficient to bound the probability of event  $\text{bad}_{1,i} \vee \text{bad}_{2,i}$  for all  $i \in [Q]$ . To do so, we give a reduction from the hardness of  $\mathcal{R}'$  relative to  $\mathcal{R}'$ .Gen.

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

- In the  $i$ th call to oracle  $\mathcal{O}^*$ , the reduction sets  $\text{stmt}' := \text{stmt}^*$ , instead of sampling  $(\text{stmt}', \text{witr}') \leftarrow \mathcal{R}'.\text{Gen}(1^\lambda)$ . Then, if for one of the previous random oracle queries  $\mathcal{H}(Z)$  it holds that  $(\text{stmt}^*, Z) \in \mathcal{R}'$ , it outputs  $\text{witr}^* := Z$  and stops (cf. event  $\text{bad}_{1,i}$ ). Otherwise, it samples  $\text{ct} \leftarrow_{\$} \{0, 1\}^{\ell_1}$ . Note that it never needs the witness  $\text{witr}'$ .
- For random oracle queries  $\mathcal{H}(Z)$  after the  $i$ th call to oracle  $\mathcal{O}^*$ , the reduction checks if  $(\text{stmt}^*, Z) \in \mathcal{R}'$ . If this holds, it outputs  $\text{witr}^* := Z$  and stops (cf. event  $\text{bad}_{2,i}$ ).

First, if  $\text{bad}_{1,i}$  occurs, it is clear that the reduction simulates  $\mathbf{G}_1$  perfectly until it stops. Also, if  $\text{bad}_{1,i}$ , it outputs a valid witness  $\text{witr}^*$  for  $\text{stmt}^*$ . Similarly, we see that if event  $\text{bad}_{2,i}$  occurs, then the reduction simulates  $\mathbf{G}_1$  perfectly until it stops and outputs a valid witness  $\text{witr}^*$  for  $\text{stmt}^*$ . We obtain that the probability of  $\text{bad}_{1,i} \vee \text{bad}_{2,i}$  is upper bounded by the advantage of the reduction against the hardness of  $\mathcal{R}'$  relative to  $\mathcal{R}'.\text{Gen}$ , which is negligible by assumption.

**Game  $\mathbf{G}_3$ :** This game is as game  $\mathbf{G}_2$ , but we change how values  $\text{ct}$  contained in messages  $\text{xm}_1$  are computed in executions of  $\mathcal{O}^*$ . Namely, we sample  $\text{ct} \leftarrow_{\$} \{0, 1\}^{\ell_1}$ . Later, before returning signatures  $\sigma_b, \sigma_s$ , we define  $\mathcal{H}(\text{witr}') := \text{ct} \oplus \text{bsm}_2$ , where  $\text{bsm}_2$  is computed using algorithm  $\text{BS.U}_2$  as in algorithm  $\text{Setup}$ . 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  $\text{Sim}_1, \text{Sim}_{RO}, \text{Sim}_2, \text{Sim}_3$ , is the following: In game  $\mathbf{G}_3$ , the oracle  $\mathcal{O}^*$  aborts if  $\text{SIG.Ver}(\text{pk}_b, \text{tx}, \sigma_b) = 0$  for  $\sigma_b := \text{Sell}(St, \text{xm}_2)$ . 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.  $\square$

## D.2 Proofs for the Construction for Unique Signatures

*Proof (of Lemma 5 (Mal. Seller - Unique Signature)).* We consider an adversary  $\mathcal{A}$  against the security of  $\text{EXC}_u[\text{SIG}, \text{BS}, \text{PS}]$  against malicious sellers. We define three events in the security game, according to the three possible ways  $\mathcal{A}$  can win.

- $\text{win}_1$ : This occurs if the security game outputs 1 and  $\text{tx} \neq \text{tx}'$ .
- $\text{win}_2$ : This occurs if the security game outputs 1,  $\text{tx} = \text{tx}'$  and  $\text{xm}_2 = \perp$ .
- $\text{win}_3$ : This occurs if the security game outputs 1,  $\text{tx} = \text{tx}', \text{xm}_2 \neq \perp$ , and  $\text{BS.Ver}(\text{pk}_{\text{BS}}, \text{sn}, \sigma_{\text{BS}}) = 0$ .

First, we bound the probability of  $\text{win}_1 \vee \text{win}_2$ . Intuitively, this follows from EUF-CMA security of SIG, because if one of the events occurs, the adversary came up with a valid signature  $\sigma_b$  for a message  $\text{tx}'$ , 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  $\text{pk}$ , and it

gets access to a signing oracle  $\text{SIG}$ . Then, the reduction runs  $\mathcal{A}$  as in the security game for  $\text{EXC}_u[\text{SIG}, \text{BS}, \text{PS}]$  against malicious sellers. Precisely, it runs  $\mathcal{A}$ , obtains a public key  $\text{pk}_{\text{BS}}$  and a nonce  $\text{sn}$ . Then, it runs  $(\text{bsm}_1, St) \leftarrow \text{U}_1(\text{pk}_{\text{BS}}, \text{sn})$ . It sets  $\text{pk}_b := \text{pk}$ , and passes  $\text{bsm}_1, \text{pk}_b$  to  $\mathcal{A}$ . The adversary outputs  $\text{pk}_s, \text{tx}$ , and a message  $\text{xm}_1$ . If  $\text{xm}_1 = \perp$  or  $\text{xm}_1 = (\text{ct}, \pi)$  and  $\text{PVer}(\text{stmt}, \pi) = 0$  for  $\text{stmt}$  as in algorithm  $\text{Buy}$ , the reduction sends  $\text{xm}_2 := \perp$  to  $\mathcal{A}$ . Otherwise, it queries a signature  $\sigma'_b \leftarrow \text{SIG}(\text{tx})$  from the signing oracle and sets  $\text{xm}_2 := \sigma'_b$ . The reduction passes  $\text{xm}_2$  to  $\mathcal{A}$  and obtains  $\text{tx}', \sigma_b, \sigma_s$  in return. If  $\text{win}_1 \vee \text{win}_2$  occurs, it returns  $(\text{tx}', \sigma_b)$  to its game. Otherwise, it aborts.

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

It remains to bound the probability of event  $\text{win}_3$ . Intuitively, this should follow from the soundness of  $\text{PS}$ . Recall that  $\text{win}_3$  occurs, if  $\text{tx} = \text{tx}', \text{xm}_2 \neq \perp$ , and  $\text{BS.Ver}(\text{pk}_{\text{BS}}, \text{sn}, \sigma_{\text{BS}}) = 0$ . In particular, if  $\text{xm}_2 \neq \perp$ , we know that for  $\text{stmt} = (\text{pk}_{\text{BS}}, \text{pk}_s, \text{tx}, \text{bsm}_1, \text{ct})$  and  $\text{xm}_1 = (\text{ct}, \pi)$  we have  $\text{PVer}(\text{stmt}, \pi) = 1$ . We assume towards contradiction that there exists a witness  $\text{witn}$  such that  $(\text{stmt}, \text{witn}) \in \mathcal{R}$ , i.e.  $\text{stmt}$  is a yes-instance. Then, by definition of  $\mathcal{R}$  and unique signatures, we know that the first component of  $\text{witn}$  is  $\sigma_b$ . and that there is a string  $\rho$  such that  $\text{ct} = \text{H}(\sigma_s) \oplus \text{BS.S}(\text{sk}_{\text{BS}}, \text{bsm}_1; \rho)$ . In combination, we get

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

by definition of algorithm  $\text{Get}$ . Recall that

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

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

In summary, we showed that  $\text{stmt}$  is not a yes-instance, violating soundness of  $\text{PS}$ . Therefore, the probability of  $\text{win}_3$  is negligible.  $\square$

*Proof (of Lemma 6 (Mal. Buyer - Unique Signature)).* We define algorithms  $\text{Sim}_1, \text{Sim}_{\text{RO}}, \text{Sim}_2, \text{Sim}_3$ , and then we show indistinguishability. The algorithms keep a list  $L$  containing tuples of the form  $(\text{tx}, \text{pk}_s, \text{ct})$ . Algorithm  $\text{Sim}_1(\text{xpar}, \text{sk}_s)$  is as follows:

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

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

Next, we present a sequence of games to show that algorithms  $\text{Sim}_1, \text{Sim}_{RO}, \text{Sim}_2, \text{Sim}_3$  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  $(\text{pk}_{BS}, \text{sk}_{BS})$  is sampled. Then, the adversary  $\mathcal{A}$  gets access to a signer oracle  $\mathcal{O}$  and an oracle  $\mathcal{O}^*$ . When called by  $\mathcal{A}$ , oracle  $\mathcal{O}^*$  samples a key pair  $(\text{pk}_s, \text{sk}_s) \leftarrow \text{SIG.Gen}(1^\lambda)$ , gives  $\text{pk}_s$  to  $\mathcal{A}$  and obtains a key  $\text{pk}_b$ , a transaction  $\text{tx}$ , and a message  $\text{bsm}_1$  in return. Then, it runs algorithm **Setup**. Concretely, it computes  $\text{bsm}_2$  and  $\sigma_s$ , defines ciphertext  $\text{ct}$  and computes a proof  $\pi$  as in the scheme. Then, it sets  $\text{xm}_1 := (\text{ct}, \pi)$  and sends  $\text{xm}_1$  to  $\mathcal{A}$ . The adversary responds with a message  $\text{xm}_2$ . If  $\text{xm}_2$  is a valid signature  $\sigma_b$  for  $\text{tx}$  with respect to  $\text{pk}_b$ , the game outputs  $\sigma_b, \sigma_s$ . 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  $\pi$  contained in message  $\text{xm}_1$  is computed by oracle  $\mathcal{O}^*$ . Before, it was computed via  $\pi \leftarrow \text{PProve}(\text{stmt}, \text{witn})$ , where  $\text{stmt}$  and  $\text{witn}$  are as in algorithm **Setup**. In game  $\mathbf{G}_1$ , we compute it using the zero-knowledge simulator  $\text{PS.PSim}$  via  $\pi \leftarrow \text{PSim}(\text{stmt})$ . By the zero-knowledge property of  $\text{PS}$ , games  $\mathbf{G}_0$  and  $\mathbf{G}_1$  are indistinguishable.

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

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

where  $Q$  is the number of queries to oracle  $\mathcal{O}^*$ , and  $\text{bad}_{1,i}$  (resp.  $\text{bad}_{2,i}$ ) denotes the event that  $\text{bad}_1$  (resp.  $\text{bad}_2$ ) occurs for the entry in  $L$  that is inserted in the  $i$ th query to  $\mathcal{O}^*$ . As  $Q$  is polynomial, it is sufficient to bound  $\text{bad}_{1,i} \vee \text{bad}_{2,i}$  for all  $i$ . To this end, we sketch a reduction from the EUF-CMA security of  $\text{SIG}$ . The reduction gets as input a public key  $\text{pk}$  and it gets access to a signing oracle  $\text{SIG}$ . It will not make use of  $\text{SIG}$ . The reduction simulates  $\mathbf{G}_1$  as it is, except for the  $i$ th call to oracle  $\mathcal{O}^*$ , and the random oracle simulation of  $H$ :

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

To argue that the reduction perfectly simulates game  $\mathbf{G}_1$  until it stops, it is sufficient to consider the distribution of  $\text{ct}$ . First, if event  $\text{bad}_{1,i}$  occurs, the simulation is clearly perfect until the reduction terminates. Also, if event  $\text{bad}_{1,i}$  does not occur, in  $\mathbf{G}_1$ , the value  $\text{ct}$  is distributed uniformly. Note that due to uniqueness of signatures, the reduction can efficiently check if  $\text{bad}_{1,i}$  occurs. Finally, we see that if event  $\text{bad}_{1,i} \vee \text{bad}_{2,i}$  occurs, then the reduction outputs a valid forgery  $(\text{tx}, x)$ . As the reduction never used its signing oracle, we obtain that the probability of  $\text{bad}_{1,i} \vee \text{bad}_{2,i}$  is upper bounded by the advantage of the reduction against the EUF-CMA security of  $\text{SIG}$ , which is negligible by assumption.

**Game  $\mathbf{G}_3$ :** This game is as game  $\mathbf{G}_2$ , but we change how ciphertexts  $\text{ct}$  are simulated in executions of oracle  $O^*$ . Namely, we sample  $\text{ct} \leftarrow_{\$} \{0, 1\}^{\ell_1}$ . Later, before the oracle returns signatures  $\sigma_b, \sigma_s$ , it defines  $H(\sigma_s) := \text{ct} \oplus \text{bsm}_2$ , where  $\text{bsm}_2$  is computed using algorithm  $\text{BS.U}_2$  as in algorithm  $\text{Setup}$ . 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  $\text{Sim}_1, \text{Sim}_{RO}, \text{Sim}_2, \text{Sim}_3$ , is exactly the same as  $\mathbf{G}_3$ , finishing the proof.  $\square$

### D.3 Proofs for the BLS Cut-and-Choose Construction

*Proof (of Lemma 1 (Mal. Seller - BLS)).* Consider an adversary  $\mathcal{A}$  against the security of  $\text{EXC}_{\text{BLS}}^{\text{cc}}[\text{SIG}, \text{BS}]$  against malicious sellers. We define three events in the security game, following the three possible ways  $\mathcal{A}$  can win.

- $\text{win}_1$ : This occurs if the security game outputs 1 and  $\text{tx} \neq \text{tx}'$ .
- $\text{win}_2$ : This occurs if the security game outputs 1,  $\text{tx} = \text{tx}'$  and  $\text{xm}_2 = \perp$ .
- $\text{win}_3$ : This occurs if the security game outputs 1,  $\text{tx} = \text{tx}', \text{xm}_2 \neq \perp$ , and  $\text{BS.Ver}(\text{pk}_{\text{BS}}, \text{sn}, \sigma_{\text{BS}}) = 0$ .

First, we bound the probability of  $\text{win}_1 \vee \text{win}_2$ . Intuitively, this follows from EUF-CMA security of  $\text{SIG}$ , because if one of the events occurs, the adversary came up with a valid signature  $\sigma_b$  for a message  $\text{tx}'$ , 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  $\text{pk}$ , and it gets access to a signing oracle  $\text{SIG}$ . Then, the reduction runs  $\mathcal{A}$  as in the security game for  $\text{EXC}_{\text{BLS}}^{\text{cc}}[\text{SIG}, \text{BS}]$  against malicious sellers. Precisely, it runs  $\mathcal{A}$ , obtains a public key  $\text{pk}_{\text{BS}}$  and a nonce  $\text{sn}$ . Then, it runs  $(\text{bsm}_1, St) \leftarrow \text{U}_1(\text{pk}_{\text{BS}}, \text{sn})$ . It

sets  $\mathbf{pk}_b := \mathbf{pk}$ , and passes  $\mathbf{bsm}_1, \mathbf{pk}_b$  to  $\mathcal{A}$ . The adversary outputs  $\mathbf{pk}_s, \mathbf{tx}$ , and a message  $\mathbf{xm}_1$ . If  $\mathbf{xm}_1 = \perp$  the reduction sets  $\mathbf{xm}_2 := \perp$ . Otherwise, if  $\mathbf{xm}_1 = (\mathbf{xm}_{1,1}, \mathbf{xm}_{1,2})$ , the reduction starts running algorithm  $\text{Buy}(\mathbf{xpar}, \mathbf{sk}_b, \mathbf{xm}_1)$ . Concretely, if this algorithm would return  $\mathbf{xm}_2 \neq \perp$ , it uses its signing oracle  $\text{SIG}$  on input  $\mathbf{tx}$  to compute  $\mathbf{xm}_2$ . Otherwise, it continues with  $\mathbf{xm}_2 = \perp$ . The reduction passes  $\mathbf{xm}_2$  to  $\mathcal{A}$  and obtains  $\mathbf{tx}', \sigma_b, \sigma_s$  in return. If  $\text{win}_1 \vee \text{win}_2$  occurs, it returns  $(\mathbf{tx}', \sigma_b)$  to its game. Otherwise, it aborts. It is clear that the reduction perfectly simulates the game for  $\mathcal{A}$ . Also, note that the pair  $(\mathbf{tx}', \sigma_b)$  that the reduction outputs in the end is valid, i.e.  $\text{SIG.Ver}(\mathbf{pk}, \mathbf{tx}', \sigma_b) = 1$ , by definition of  $\text{win}_1 \vee \text{win}_2$ . Further, note that if  $\text{win}_1$  occurs, the reduction did only query oracle  $\text{SIG}$  on input  $\mathbf{tx} \neq \mathbf{tx}'$ , and not on input  $\mathbf{tx}'$ . Similarly, if  $\text{win}_2$  occurs, the reduction did not query  $\text{SIG}$  at all. Therefore, the probability of  $\text{win}_1 \vee \text{win}_2$  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  $\text{win}_3$ . Intuitively, this follows via a statistical argument based on the cut-and-choose technique. Recall that  $\text{win}_3$  occurs, if  $\mathbf{tx} = \mathbf{tx}', \mathbf{xm}_2 \neq \perp$ , and  $\text{BS.Ver}(\mathbf{pk}_{\text{BS}}, \mathbf{sn}, \sigma_{\text{BS}}) = 0$ . We make the following observations.

1. If  $\text{win}_3$  occurs, then algorithm  $\text{Get}$  must have output  $\perp$ . This is because due  $\mathbf{xm}_2 \neq \perp$  we know that  $e(\mathbf{bsm}_1, \mathbf{pk}_{\text{BS}, k_j}) = e(\mathbf{bsm}_{2, k_j}, g_2)$  for all  $j \in [\lambda]$ , for notation as in algorithm  $\text{Buy}$ . Also, assuming  $\text{Get}$  does not output  $\perp$ , we know that  $e(\mathbf{bsm}_1, \mathbf{pk}_{\text{BS}, \bar{k}_j}) = e(\mathbf{bsm}_{2, \bar{k}_j}, g_2)$  for some  $j \in [\lambda]$ , with notation as in  $\text{Get}$ . Correctness of algorithm  $\text{reconst}_{g_1, 0}$  now implies that  $\mathbf{bsm}_2$  as computed by  $\text{Get}$  is a valid second message for the first message  $\mathbf{bsm}_1$ , which has to lead to a valid blind signature  $\sigma_{\text{BS}}$  via algorithm  $\text{U}_2$ .
2. If algorithm  $\text{Get}$  outputs  $\perp$ , then all  $\mathbf{bsm}_{2, \bar{k}_j}$  for  $j \in [\lambda]$  as computed in  $\text{Get}$  are invalid, i.e.  $e(\mathbf{bsm}_1, \mathbf{pk}_{\text{BS}, \bar{k}_j}) \neq e(\mathbf{bsm}_{2, \bar{k}_j}, g_2)$ . This is by definition of  $\text{Get}$ .
3. If  $\text{win}_3$  occurs, then all  $\sigma_{\bar{k}_j}$  for  $j \in [\lambda]$  (as computed in  $\text{Get}$ ) are valid, i.e. for all  $j \in [\lambda]$ ,  $\sigma_{\bar{k}_j}$  is the unique value satisfying  $\text{SIG.Ver}(\mathbf{pk}_{s, \bar{k}_j}, \mathbf{tx}, \sigma_{\bar{k}_j}) = 1$  for  $\mathbf{pk}_{s, \bar{k}_j} := \mathbf{pk}_s \cdot \prod_{i=1}^{\lambda} (\text{coeff}'_i)^{\bar{k}_j^i}$ . This is because all  $\sigma_{k_j}$  are valid in the same sense (due to  $\mathbf{xm}_2 \neq \perp$ ) and due to the correctness of algorithm  $\text{reconst}_{g_1, \bar{k}_j}$ .

Using these three observations, we now finish the statistical argument. For that, consider the moment of the first query of the form  $\text{H}_c(\mathbf{xm}_{1,1})$ . It is clear that  $\mathbf{xm}_{1,1} = ((\text{ct}_j)_{j \in [2\lambda]}, (\text{coeff}_j, \text{coeff}'_j)_{j \in [\lambda]})$  information theoretically determines the polynomials  $f, f'$  and therefore all  $\sigma_j$  and  $\mathbf{pk}_{\text{BS}, j}$  for  $j \in [2\lambda]$ . Therefore,  $\mathbf{xm}_{1,1}$  also determines the values  $\mathbf{bsm}_{2, j} := \text{ct}_j \oplus \text{H}(\sigma_j)$  for all  $j \in [2\lambda]$ . Due to the third observation, these correspond to the values computed in  $\text{Buy}$  and  $\text{Get}$ . Due to the first and second observation, and the fact that  $\text{Buy}$  output  $\mathbf{xm}_2 \neq \perp$  if  $\text{win}_3$  occurs, we therefore have

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

Thus, conditioned on  $\text{win}_3$ , the value  $\mathbf{xm}_{1,1}$  fully determines  $b_0, \dots, b_{\lambda-1}$ . This means that  $\text{win}_3$  can only occur if for some query of the form  $\text{H}_c(\mathbf{xm}_{1,1})$ , the hash

value coincides with the bits  $b_0, \dots, b_{\lambda-1}$  that are determined by  $\mathbf{xm}_{1,1}$ , which happens with probability  $1/2^\lambda$ . As there are at most polynomially many queries of this form, the probability of  $\text{win}_3$  is negligible, which ends the proof.  $\square$

*Proof (of Lemma 2 (Mal. Buyer - BLS)).* Before we provide algorithms  $\text{Sim}_1, \text{Sim}_{RO}, \text{Sim}_2, \text{Sim}_3$ , we give a sequence of hybrid games, starting from the security game against malicious buyers with bit  $b = 0$  (i.e. computing  $\mathbf{xm}_1$  and  $\sigma_b$  honestly via algorithms  $\text{Setup}$  and  $\text{Sell}$ ). 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  $(\mathbf{pk}_{\text{BS}}, \mathbf{sk}_{\text{BS}})$  is sampled. Then,  $\mathbf{pk}_{\text{BS}}$  is given to the adversary. The adversary also gets access to a signer oracle  $\mathcal{O}$  for BS simulating  $\text{BS.S}(\mathbf{sk}_{\text{BS}}, \cdot)$ , and an oracle  $\mathcal{O}^*$  which is as follows. When called, it first samples a key pair  $(\mathbf{pk}_s = g_2^{\mathbf{sk}_s}, \mathbf{sk}_s)$  and outputs  $\mathbf{pk}_s$ . Then, it gets a key  $\mathbf{pk}_b$ , a transaction  $\mathbf{tx}$ , and a message  $\mathbf{bsm}_1 \in \mathbb{G}_1$  from the adversary. It sets  $\mathbf{xpar} := (\mathbf{pk}_{\text{BS}}, \mathbf{bsm}_1, \mathbf{pk}_b, \mathbf{pk}_s, \mathbf{tx})$  and runs  $(\mathbf{xm}_1, St) \leftarrow \text{Setup}(\mathbf{xpar}, \mathbf{sk}_{\text{BS}}, \mathbf{sk}_s)$ . In this scheme,  $\mathbf{xm}_1$  has the form  $\mathbf{xm}_1 = (\mathbf{xm}_{1,1}, \mathbf{xm}_{1,2})$  with  $\mathbf{xm}_{1,1} = ((\mathbf{ct}_j)_{j \in [2\lambda]}, (\text{coeff}_j, \text{coeff}'_j)_{j \in [\lambda]})$  and  $\mathbf{xm}_{1,2} = (\sigma_{k_j})_{j \in [\lambda]}$ . Then, the oracle gives  $\mathbf{xm}_1$  to the adversary, obtains  $\mathbf{xm}_2 = \sigma_b$ , runs  $\text{Sell}$  (which does not do anything for this scheme), and aborts if  $\sigma_b$  is not valid, i.e.  $\text{SIG.Ver}(\mathbf{pk}_b, \mathbf{tx}, \sigma_b) = 0$ . Otherwise, it returns  $\sigma_b, \sigma_s$  to the adversary, where  $\sigma_s \leftarrow \text{SIG.Sig}(\mathbf{sk}_s, \mathbf{tx})$ . In the end, the game outputs whatever the adversary outputs.

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

**Game  $\mathbf{G}_1$ :** In this game, we change the execution of algorithm  $\text{Setup}$  in oracle  $\mathcal{O}^*$ . Namely, in the beginning of the algorithms execution, we now sample uniformly random bits  $b_0, \dots, b_{\lambda-1}$ . Then, we compute  $\mathbf{xm}_{1,1}$  as before, and abort if  $H_c(\mathbf{xm}_{1,1})$  is already defined. Otherwise, we program  $H_c(\mathbf{xm}_{1,1}) := b_0, \dots, b_{\lambda-1}$ , and continue as before. The probability of such an abort is negligible, due to the entropy of  $\text{coeff}'_1$ . 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  $\mathbf{xm}_{1,1}$ .

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

- $\text{bad}_1$ : When the game computes the values  $(\mathbf{ct}_j)_{j \in [2\lambda]}$  during the execution of  $\text{Setup}$ , the hash value  $H(\sigma_{\bar{k}_j})$  is already defined for some  $j \in [\lambda]$ .
- $\text{bad}_2$ : After the game computes the values  $(\mathbf{ct}_j)_{j \in [2\lambda]}$  during the execution of  $\text{Setup}$ , but before the game gives  $\sigma_s$  to the adversary in the same interaction, a query  $H(\sigma_{\bar{k}_j})$  is made for some  $j \in [\lambda]$ .

We have

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



where  $Q$  is the number of queries to oracle  $O^*$ , and the event  $\mathbf{bad}_{1,i}$  (resp.  $\mathbf{bad}_{2,i}$ ) occurs if  $\mathbf{bad}_1$  (resp.  $\mathbf{bad}_2$ ) occurs in the  $i$ th interaction between the adversary and  $O^*$ . As  $Q$  is polynomial, it is sufficient to bound  $\mathbf{bad}_{1,i} \vee \mathbf{bad}_{2,i}$  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  $\mathbf{pk}$  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  $i$ th call to oracle  $O^*$ , and the random oracle simulation of H:

- In the  $i$ th call to oracle  $O^*$ , the reduction sets  $\mathbf{pk}_s := \mathbf{pk}$ , instead of sampling the pair  $(\mathbf{pk}_s, \mathbf{sk}_s)$  on its own.
- This means that it can not define the polynomial  $f'$  as in the game explicitly. Instead, the reduction runs  $((\mathbf{sk}_{s,k_j})_{j \in [\lambda]}, (\mathbf{coeff}'_j)_{j \in [\lambda]}) \leftarrow \mathbf{polyGen}_{g_2,p}(\lambda, \mathbf{pk}_s, (k_j)_{j \in [\lambda]})$ .
- The reduction checks if event  $\mathbf{bad}_{1,i}$  occurs, by checking for each previous random oracle query  $H(x)$  if  $\mathbf{SIG.Ver}(\mathbf{pk}_{s,\bar{k}_j}, \mathbf{tx}, x) = 1$  for some  $j \in [\lambda]$ , where  $\mathbf{pk}_{s,\bar{k}_j} := \mathbf{pk}_s \cdot \prod_{i=1}^{\lambda} (\mathbf{coeff}'_i)^{\bar{k}_j^i}$ . Note that this check is correct due to the uniqueness of SIG. If  $\mathbf{bad}_{1,i}$  occurs, say for  $j^* \in [\lambda]$ , the reduction computes a signature  $\sigma$  for  $\mathbf{tx}$  via  $\sigma := \mathbf{reconst}_{g,0}((j^*, x), (k_i, \sigma_{s,k_i})_{i \in [\lambda]})$ . Then, it outputs  $(\mathbf{tx}, \sigma)$  as a forgery to the EUF-CMA game. If  $\mathbf{bad}_{1,i}$  does not occur, it continues by sampling all  $\mathbf{ct}_{\bar{k}_j}$  at random.
- The reduction can check if event  $\mathbf{bad}_{2,i}$  occurs similar to event  $\mathbf{bad}_{1,i}$  using algorithm  $\mathbf{SIG.Ver}$  whenever the adversary queries H. If  $\mathbf{bad}_{2,i}$  occurs, the reduction computes a signature  $\sigma$  in a similar way as above and outputs  $(\mathbf{tx}, \sigma)$  as a forgery to the EUF-CMA game.
- If the reduction has to output  $\sigma_s$  to the adversary in the  $i$ th 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  $\mathbf{polyGen}_{g_2,p}$ . Also, if  $\mathbf{bad}_{1,i} \vee \mathbf{bad}_{2,i}$  occurs, the reduction does not abort and returns a valid forgery, following from the correctness of algorithm  $\mathbf{reconst}_{g,0}$ . Also, the reduction never uses its signing oracle. This implies that the probability of  $\mathbf{bad}_{1,i} \vee \mathbf{bad}_{2,i}$  is negligible, by the EUF-CMA security of SIG.

**Game  $\mathbf{G}_3$ :** In game  $\mathbf{G}_3$ , we change how the values  $\mathbf{ct}_{\bar{k}_j}$  for  $j \in [\lambda]$  are computed in executions of algorithm Setup in oracle  $O^*$ . Concretely, while they were computed as  $\mathbf{ct}_{\bar{k}_j} = H(\sigma_{\bar{k}_j}) \oplus \mathbf{bsm}_{2,\bar{k}_j}$  before, we now sample them at random as  $\mathbf{ct}_{\bar{k}_j} \leftarrow \{0, 1\}^\ell$ . Later, before giving  $\sigma_s$  to the adversary in the same interaction, we let the game program  $H(\sigma_{\bar{k}_j}) := \mathbf{ct}_{\bar{k}_j} \oplus \mathbf{bsm}_{2,\bar{k}_j}$ . 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  $\mathbf{bsm}_{2,\bar{k}_j}$  to compute  $\mathbf{xm}_1$ , but only once we output  $\sigma_s$ . This will allow us to compute  $\mathbf{xm}_1$  without access to  $\mathbf{sk}_{\mathbf{BS}}$ . Namely, we will now compute the values  $\mathbf{coeff}_j$  used during the computation of  $\mathbf{xm}_1$  as

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

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

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

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

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

#### D.4 Proofs for the Adaptor Cut-and-Choose Construction

*Proof (of Lemma 9 (Mal. Seller - Adaptor CC)).* We consider an adversary  $\mathcal{A}$  against the security of  $\text{EXC}_a^{\text{cc}}[\text{SIG}, \text{aSIG}, \text{BS}]$  against malicious sellers. We define three events in the security game, following the three possible ways  $\mathcal{A}$  can win.

- $\text{win}_1$ : This occurs if the security game outputs 1 and  $\text{tx} \neq \text{tx}'$ .
- $\text{win}_2$ : This occurs if the security game outputs 1,  $\text{tx} = \text{tx}'$  and  $\text{xm}_2 = \perp$ .
- $\text{win}_3$ : This occurs if the security game outputs 1,  $\text{tx} = \text{tx}', \text{xm}_2 \neq \perp$ , and  $\text{BS.Ver}(\text{pk}_{\text{BS}}, \text{sn}, \sigma_{\text{BS}}) = 0$ .

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

It remains to bound the probability of event  $\text{win}_3$ . Recall that  $\text{win}_3$  occurs, if  $\text{tx} = \text{tx}'$ ,  $\text{xm}_2 \neq \perp$ , and  $\text{BS.Ver}(\text{pk}_{\text{BS}}, \text{sn}, \sigma_{\text{BS}}) = 0$ . We bound the probability of event  $\text{win}_3$  by partitioning it into two sub-events.

- $\text{win}_{3,1}$ : This event occurs, if  $\text{win}_3$  occurs, and for  $y := \text{Ext}(\tilde{\sigma}_b, \sigma_b)$  computed in  $\text{Get}$ , we have  $g^y \neq Y$ .
- $\text{win}_{3,2}$ : This event occurs, if  $\text{win}_3$  occurs, and for  $y := \text{Ext}(\tilde{\sigma}_b, \sigma_b)$  computed in  $\text{Get}$ , we have  $g^y = Y$ .

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

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

1. If  $\text{win}_{3,2}$  occurs, then algorithm  $\text{Get}$  must have output  $\perp$ . This is because due  $\text{xm}_2 \neq \perp$  we know that  $e(\text{bsm}_1, \text{pk}_{\text{BS}, k_j}) = e(\text{bsm}_{2, k_j}, g_2)$  for all  $j \in [\lambda]$ , for notation as in algorithm  $\text{Buy}$ . Also, assuming  $\text{Get}$  does not output  $\perp$ , we know that  $e(\text{bsm}_1, \text{pk}_{\text{BS}, \bar{k}_j}) = e(\text{bsm}_{2, \bar{k}_j}, g_2)$  for some  $j \in [\lambda]$ , with notation as in  $\text{Get}$ . Correctness of algorithm  $\text{reconst}_{g_1, 0}$  now implies that  $\text{bsm}_2$  as computed by  $\text{Get}$  is a valid second message for the first message  $\text{bsm}_1$ , which has to lead to a valid blind signature  $\sigma_{\text{BS}}$  via algorithm  $\text{U}_2$ .
2. If algorithm  $\text{Get}$  outputs  $\perp$ , then all  $\text{bsm}_{2, \bar{k}_j}$  for  $j \in [\lambda]$  as computed in  $\text{Get}$  are invalid, i.e.  $e(\text{bsm}_1, \text{pk}_{\text{BS}, \bar{k}_j}) \neq e(\text{bsm}_{2, \bar{k}_j}, g_2)$ . This is by definition of  $\text{Get}$ .
3. If  $\text{win}_{3,2}$  occurs, then the polynomial  $f'$  computed by  $\text{Get}$  is exactly the same polynomial as defined by the values  $\text{coeff}'_j$ . This is because in this event we assume  $g^y = Y$ , and as  $\text{xm}_2 \neq \perp$  we know that  $g^{y_{k_j}} = Y_{k_j}$  for all  $j \in [\lambda]$ . Therefore, correctness of algorithm  $\text{reconst}_q$  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  $\text{H}_c(\text{xm}_{1,1})$ . It is clear that  $\text{xm}_{1,1} = (Y, (\text{ct}_j)_{j \in [2\lambda]}, (\text{coeff}_j, \text{coeff}'_j)_{j \in [\lambda]})$  information theoretically determines the polynomials  $f, f'$  and therefore all  $y_j = f'(j)$  and  $\text{pk}_{\text{BS}, j}$  for  $j \in [2\lambda]$ . Therefore,  $\text{xm}_{1,1}$  also determines the values  $\text{bsm}_{2, j} := \text{ct}_j \oplus \text{H}(y_j)$  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** outputs  $\text{xm}_2 \neq \perp$  if  $\text{win}_{3,2}$  occurs, we therefore have

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

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

*Proof (of Lemma 10 (Mal. Buyer - Adaptor CC)).* Before we provide algorithms  $\text{Sim}_1, \text{Sim}_{RO}, \text{Sim}_2, \text{Sim}_3$ , we give a sequence of hybrid games, starting from the security game against malicious buyers with bit  $b = 0$  (i.e. computing  $\text{xm}_1$  and  $\sigma_b$  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  $(\text{pk}_{\text{BS}}, \text{sk}_{\text{BS}})$  is sampled. Then,  $\text{pk}_{\text{BS}}$  is given to the adversary. The adversary also gets access to a signer oracle  $\text{O}$  for **BS** simulating  $\text{BS.S}(\text{sk}_{\text{BS}}, \cdot)$ , and an oracle  $\text{O}^*$  which is as follows. When called, it first samples a key pair  $(\text{pk}_s, \text{sk}_s) \leftarrow \text{SIG.Gen}(1^\lambda)$  and outputs  $\text{pk}_s$ . Then, it gets a key  $\text{pk}_b$ , a transaction  $\text{tx}$ , and a message  $\text{bsm}_1 \in \mathbb{G}_1$  from the adversary. It sets  $\text{xpar} := (\text{pk}_{\text{BS}}, \text{bsm}_1, \text{pk}_b, \text{pk}_s, \text{tx})$  and runs  $(\text{xm}_1, \text{St}) \leftarrow \text{Setup}(\text{xpar}, \text{sk}_{\text{BS}}, \text{sk}_s)$ . In this scheme,  $\text{xm}_1$  has the form  $\text{xm}_1 = (\text{xm}_{1,1}, \text{xm}_{1,2})$  with  $\text{xm}_{1,1} = (Y = g^y, (\text{ct}_j)_{j \in [2\lambda]}, (\text{coeff}_j, \text{coeff}'_j)_{j \in [\lambda]})$  and  $\text{xm}_{1,2} = (y_{k_j})_{j \in [\lambda]}$ . Then, the oracle gives  $\text{xm}_1$  to the adversary, obtains  $\text{xm}_2 = \tilde{\sigma}_b$ , and runs **Sell**, which aborts if  $\text{PreVer}(\text{pk}_b, \text{tx}, g^y, \tilde{\sigma}_b) = 0$  and computes  $\sigma_b := \text{Adapt}(\text{pk}_b, \tilde{\sigma}_b, y)$ . Further, the oracle aborts if  $\sigma_b$  is not valid, i.e.  $\text{SIG.Ver}(\text{pk}_b, \text{tx}, \sigma_b) = 0$ . From now on, we omit this check, which is redundant due to adaptability of **SIG**. In case there is no abort, the oracle returns  $\sigma_b, \sigma_s$  to the adversary, where  $\sigma_s \leftarrow \text{SIG.Sig}(\text{sk}_s, \text{tx})$ . In the end, the game outputs whatever the adversary outputs.

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

**Game  $\mathbf{G}_1$ :** In this game, we change the execution of algorithm **Setup** in oracle  $\text{O}^*$ . Namely, in the beginning of the algorithms execution, we now sample uniformly random bits  $b_0, \dots, b_{\lambda-1}$ . Then, we compute  $\text{xm}_{1,1}$  as before, and abort if  $H_c(\text{xm}_{1,1})$  is already defined. Otherwise, we program  $H_c(\text{xm}_{1,1}) := b_0, \dots, b_{\lambda-1}$ , 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  $\text{xm}_{1,1}$ .

**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**<sub>1</sub>: When the game computes the values  $(ct_j)_{j \in [2\lambda]}$  during the execution of **Setup**, the hash value  $H(y_{\bar{k}_j})$  is already defined for some  $j \in [\lambda]$ .
- **bad**<sub>2</sub>: After the game computes the values  $(ct_j)_{j \in [2\lambda]}$  during the execution of **Setup**, but before the game gives  $\sigma_s$  to the adversary in the same interaction, a query  $H(y_{\bar{k}_j})$  is made for some  $j \in [\lambda]$ .

We have

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

where  $Q$  is the number of queries to oracle  $O^*$ , and the event **bad**<sub>1, $i$</sub>  (resp. **bad**<sub>2, $i$</sub> ) occurs if **bad**<sub>1</sub> (resp. **bad**<sub>2</sub>) occurs in the  $i$ th interaction between the adversary and  $O^*$ . As  $Q$  is polynomial, it is sufficient to bound **bad**<sub>1, $i$</sub>   $\vee$  **bad**<sub>2, $i$</sub>  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  $i$ th call to oracle  $O^*$ , and the random oracle simulation of  $H$ :

- In the  $i$ th 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  $((y_{k_j})_{j \in [\lambda]}, (\text{coeff}'_j)_{j \in [\lambda]}) \leftarrow \text{polyGen}_{g,q}(\lambda, Y, (k_j)_{j \in [\lambda]})$ .
- The reduction checks if event **bad**<sub>1, $i$</sub>  occurs, by checking for each previous random oracle query  $H(x)$  if  $g^x = Y_{\bar{k}_j}$  for some  $j \in [\lambda]$ , where  $Y_{\bar{k}_j} := Y \cdot \prod_{i=1}^{\lambda} (\text{coeff}'_i)^{\bar{k}_j^i}$ . If **bad**<sub>1, $i$</sub>  occurs, say for  $j^* \in [\lambda]$ , the reduction computes the discrete logarithm  $y$  of  $Y$  via  $f'(X) := \text{reconst}_q((j^*, x), (k_i, y_{k_j})_{i \in [\lambda]})$  and  $y = f'(0)$ . Then, it outputs  $y$  as a DLOG solution. If **bad**<sub>1, $i$</sub>  does not occur, it continues by sampling all  $ct_{\bar{k}_j}$  at random.
- The reduction can check if event **bad**<sub>2, $i$</sub>  occurs similar to event **bad**<sub>1, $i$</sub>  using the check  $g^x = Y_{\bar{k}_j}$  for all  $j \in [\lambda]$  whenever the adversary queries  $H(x)$ . If **bad**<sub>2, $i$</sub>  occurs, the reduction computes  $y$  in a similar way as above and outputs  $y$  as a DLOG solution.
- If the reduction has to output  $\sigma_s$  to the adversary in the  $i$ th 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  $\text{polyGen}_{g,q}$ . Also, if **bad**<sub>1, $i$</sub>   $\vee$  **bad**<sub>2, $i$</sub>  occurs, the reduction does not abort and returns a valid forgery, following from the correctness of algorithm  $\text{reconst}_q$ . This implies that the probability of **bad**<sub>1, $i$</sub>   $\vee$  **bad**<sub>2, $i$</sub>  is negligible, by the DLOG assumption in  $\mathbb{G}$ .

**Game  $\mathbf{G}_3$ :** In game  $\mathbf{G}_3$ , we change how the values  $ct_{\bar{k}_j}$  for  $j \in [\lambda]$  are computed in executions of algorithm **Setup** in oracle  $O^*$ . Concretely, while they were computed as  $ct_{\bar{k}_j} = H(y_{\bar{k}_j}) \oplus \text{bsm}_{2,\bar{k}_j}$  before, we now sample them at random as

$\text{ct}_{\bar{k}_j} \leftarrow_{\$} \{0, 1\}^\ell$ . Later, before giving  $\sigma_s$  to the adversary in the same interaction, we let the game program  $\mathbf{H}(y_{\bar{k}_j}) := \text{ct}_{\bar{k}_j} \oplus \text{bsm}_{2, \bar{k}_j}$ . 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  $\mathbf{O}^*$  again. Namely, note that due to the previous change, we do not need the values  $\text{bsm}_{2, \bar{k}_j}$  to compute  $\text{xm}_1$ , but only once we output  $\sigma_s$ . This will allow us to compute  $\text{xm}_1$  without access to  $\text{sk}_{\text{BS}}$ . Namely, we will now compute the values  $\text{coeff}_j$  used during the computation of  $\text{xm}_1$  as

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

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

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

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

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

## E Omitted Constructions of Redeem Protocols

### E.1 Generic Construction.

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

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

The protocol is presented in Figure 7. Completeness follows from the uniqueness of  $\text{BS}$ . Security proofs are given in Supplementary Material F.

**Lemma 11.** *If  $\text{BS}$  has unique signatures,  $\text{SIG}$  is smooth and  $\text{PS}$  is sound, then  $\text{RP}[\text{SIG}, \text{BS}, \text{PS}]$  is secure against malicious services.*

**Lemma 12.** *Assume that  $\text{PS}$  is zero-knowledge and  $\text{SIG}$  is EUF-CMA secure. Then  $\text{RP}[\text{SIG}, \text{BS}, \text{PS}]$  is secure against malicious users.*

<b>Promise</b> ( $\text{rpar}, \text{sk}_{\text{BS}}, \text{sk}_s$ )	<b>VerPromise</b> ( $\text{rpar}, \text{prom} = (\text{ct}, \pi)$ )
01 $\sigma_{\text{BS}} \leftarrow \text{BS.Sig}(\text{sk}_{\text{BS}}, \text{sn})$	07 $\text{stmt} := (\text{pk}_{\text{BS}}, \text{pk}_s, \text{tx}, \text{sn}, \text{ct})$
02 $\sigma_s \leftarrow \text{SIG.Sig}(\text{sk}_s, \text{tx})$	08 <b>return</b> $\text{PVer}(\text{stmt}, \pi)$
03 $\text{ct} := H(\text{sn}, \sigma_{\text{BS}}) \oplus \sigma_s$	<b>Redeem</b> ( $\text{rpar}, \text{prom} = (\text{ct}, \pi), \sigma_{\text{BS}}$ )
04 $\text{stmt} := (\text{pk}_{\text{BS}}, \text{pk}_s, \text{tx}, \text{sn}, \text{ct})$	09 <b>return</b> $\sigma_s := \text{ct} \oplus H(\text{sn}, \sigma_{\text{BS}})$
05 $\pi \leftarrow \text{PProve}(\text{stmt}, \sigma_{\text{BS}})$	
06 <b>return</b> $\text{prom} := (\text{ct}, \pi)$	

**Fig. 7.** The redeem protocol  $\text{RP}[\text{SIG}, \text{BS}, \text{PS}] = (\text{Promise}, \text{VerPromise}, \text{Redeem})$  for a signature scheme  $\text{SIG}$  and a blind signature scheme  $\text{BS}$ , where  $\text{PS} = (\text{PProve}, \text{PVer})$  is a NIZK for  $\mathcal{R}$  and  $H : \{0, 1\}^* \rightarrow \{0, 1\}^\ell$  and is a random oracle.

### E.2 Construction for Schnorr Signatures using Cut-and-Choose

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

**Lemma 13.** *If  $\text{BS}$  has unique signatures, then  $\text{RP}_{\text{Schn}}^{\text{cc}}[\text{SIG}, \text{BS}]$  is secure against malicious services.*

**Lemma 14.** *If the Schnorr signature scheme  $\text{SIG}$  is sEUF-CMA secure, and the DLOG assumption holds in  $\mathbb{G}$ , then  $\text{RP}_{\text{Schn}}^{\text{cc}}[\text{SIG}, \text{BS}]$  is secure against malicious users.*



```

Promise(rpar, skBS, sks)
// Compute Schnorr signature
01  $k \leftarrow \mathbb{Z}_q^*$ ,  $T := g^k$ ,  $e := H_q(T, \text{tx})$ ,  $s := k - e \cdot \text{sk}_s$ 
// Share  $\sigma_{\text{BS}}$  and  $s$ 
02  $r_1, \dots, r_\lambda \leftarrow \mathbb{Z}_p$ ,  $r'_1, \dots, r'_\lambda \leftarrow \mathbb{Z}_q$ ,  $\text{coeff}'_0 := g^s$ 
03  $f(X) = \text{sk}_{\text{BS}} + \sum_{j=1}^{\lambda} r_j \cdot X^j \in \mathbb{Z}_p[X]$ ,  $f'(X) = s + \sum_{j=1}^{\lambda} r'_j \cdot X^j \in \mathbb{Z}_q[X]$ 
04 for  $j \in [2\lambda]$ :  $\text{sk}_j := f(j)$ ,  $s_j := f'(j)$ ,  $\sigma_j := H(\text{sn})^{\text{sk}_j}$ 
05 for  $j \in [\lambda]$ :  $\text{coeff}_j := g_2^{r_j}$ ,  $\text{coeff}'_j := g^{r'_j}$ 
// Encrypt  $s_j$  with  $\sigma_j$ 
06 for  $j \in [2\lambda]$ :  $\text{ct}_j := \hat{H}_q(\text{sn}, \sigma_j) \oplus s_j$ 
// Cut-and-choose
07  $\text{prom}_1 := ((\text{ct}_j)_{j \in [2\lambda]}, (\text{coeff}'_0, e), (\text{coeff}_j, \text{coeff}'_j)_{j \in [\lambda]})$ 
08  $b_0 \dots b_{\lambda-1} := H_c(\text{prom}_1)$ 
09 for  $j \in [\lambda]$ :  $k_j := 2j - b_{j-1}$ 
10 return  $\text{prom} := (\text{prom}_1, \text{prom}_2 := (\sigma_{k_j}, s_{k_j})_{j \in [\lambda]})$ 

VerPromise(rpar,  $\text{prom} = (\text{prom}_1, \text{prom}_2 = (\sigma_{\text{BS}, k_j}, s_{k_j})_{j \in [\lambda]})$ )
// Verify cut-and-choose
11  $b_0 \dots b_{\lambda-1} := H_c(\text{prom}_1)$ 
12 for  $j \in [\lambda]$ :
13  $k_j := 2j - b_{j-1}$ ,  $\text{pk}_{\text{BS}, k_j} := \text{pk}_{\text{BS}} \cdot \prod_{i=1}^{\lambda} (\text{coeff}_i)^{k_j^i}$ 
14 if  $\text{ct}_{k_j} \neq \hat{H}_q(\text{sn}, \sigma_{k_j}) \oplus s_{k_j} \vee g^{s_{k_j}} \neq \prod_{i=0}^{\lambda} (\text{coeff}'_i)^{k_j^i}$ : return 0
15 if  $\text{BS.Ver}(\text{pk}_{\text{BS}, k_j}, \text{sn}, \sigma_{k_j}) = 0$ : return 0
// Verify Schnorr signature in the exponent
16  $T := \text{coeff}'_0 \cdot (\text{pk}_s)^e$ 
17 if  $e \neq H_q(T, \text{tx})$ : return 0
18 return 1

Redeem(rpar,  $\text{prom} = (\text{prom}_1, \text{prom}_2)$ ,  $\sigma_{\text{BS}}$ )
19  $b_0 \dots b_{\lambda-1} := H_c(\text{prom}_1)$ 
// Reconstruct all shares
20 for  $j \in [\lambda]$ :
21  $k_j := 2j - b_{j-1}$ ,  $\bar{k}_j := 2j - (1 - b_{j-1})$ 
22  $\sigma_{\bar{k}_j} := \text{reconst}_{g_1, \bar{k}_j}((0, \sigma_{\text{BS}}), (k_i, \sigma_{k_i})_{i \in [\lambda]})$ 
23  $s_{\bar{k}_j} := \text{ct}_{\bar{k}_j} \oplus \hat{H}_q(\text{sn}, \sigma_{\bar{k}_j})$ 
// Try to find correct  $s$ 
24 for  $j \in [\lambda]$ :
25  $s := \text{reconst}_q((\bar{k}_j, s_{\bar{k}_j}), (k_i, s_{k_i})_{i \in [\lambda]})$ 
26 if  $\text{coeff}'_0 = g^s$ : return  $\sigma_s := (s, e)$ 
27 return  $\perp$ 

```

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

## F 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.

### F.1 Proofs for the Generic Construction

*Proof (of Lemma 11 (Mal. Service - Generic)).* To prove the claim, we present an algorithm `Ext` that takes as input parameters `rpar`, a promise message `prom = (ct, π)`, and a list  $\mathcal{Q}$  of random oracle queries and outputs a blind signature  $\sigma_{BS}$ . Algorithm `Ext(rpar, prom, Q)` is as follows:

1. Parse `rpar = (pkBS, pks, tx, sn)`.
2. Find an entry  $((sn, \sigma_{BS}), H(sn, \sigma_{BS}))$  in  $\mathcal{Q}$ , such that  $BS.Ver(pk_{BS}, sn, \sigma_{BS}) = 1$ .
3. If such an entry is found, return  $\sigma_{BS}$ . 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 `win1` which occurs if  $VerPromise(rpar, prom) = 1$  and `Ext` outputs  $\perp$ . We also define the event `win2` which occurs if  $VerPromise(rpar, prom) = 1$ , algorithm `Ext` outputs a valid blind signature  $\sigma_{BS}$ , but for  $\sigma_s \leftarrow Redeem(rpar, prom, \sigma_{BS})$  we have  $SIG.Ver(pk_s, tx, \sigma_s) = 0$ . 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 `win1` or `win2` occurs.

First, we upper bound the probability of `win1`. If `win1` occurs, we have  $PVer(stmt, \pi) = 1$  for `stmt = (pkBS, pks, tx, sn, ct)`. Further, if `Ext` outputs  $\perp$ , then  $H(sn, \sigma_{BS})$  is not yet defined, where  $\sigma_{BS}$  is the unique signature that satisfies  $BS.Ver(pk_{BS}, sn, \sigma_{BS}) = 1$ . Therefore, the value  $H(sn, \sigma_{BS}) \oplus ct$  is uniformly random at this point. By smoothness of `SIG`, we therefore know that the probability that  $SIG.Ver(pk_s, tx, ct \oplus H(sn, \sigma_{BS})) = 1$  is negligible. Thus, assuming `win1` occurs, we have `stmt`  $\notin \mathcal{L}_\lambda$  with overwhelming probability, violating soundness of `PS`. Therefore, the probability of `win1` is negligible.

Next, we upper bound the probability of `win2`. Note that by definition of algorithm `Redeem`, if `win2` occurs, we have that

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

where  $\sigma_{BS}$  is output by `Ext` and satisfies  $BS.Ver(pk_{BS}, sn, \sigma_{BS}) = 1$ . Due to uniqueness of `BS`, this implies that `stmt`  $\notin \mathcal{L}_\lambda$ , violating the soundness of `PS`. Therefore, the probability of `win2` is also negligible.  $\square$

*Proof (of Lemma 12 (Mal. User - Generic)).* In order to prove the statement, we provide algorithms `Sim`, `SimRO` and `Ext` that share state.

*Simulatability.* Algorithms `Sim`, `SimRO` simulate promise messages `prom = (ct, π)` and the random oracle `H`. The algorithms share a list  $L$ , that stores tuples  $(sn, ct, \sigma_s)$ . The list is initially empty. Algorithm `Sim(rpar, sk_s)` is as follows:

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

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

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

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

*Extractability.* We provide algorithm  $\text{Ext}$  that shares state with algorithms  $\text{Sim}$  and  $\text{Sim}_{\text{RO}}$  as above, and extracts blind signatures  $\sigma_{\text{BS}}$  from signatures  $\sigma_s$  that are computed from a (simulated) promise message. Algorithm  $\text{Ext}(\text{rpar}, \text{sk}_s, \sigma_s)$  searches for a query  $(\text{sn}, \sigma_{\text{BS}})$  for which  $\text{H}(\text{sn}, \sigma_{\text{BS}})$  is defined and it holds that  $\text{BS.Ver}(\text{pk}_{\text{BS}}, \text{sn}, \sigma_{\text{BS}}) = 1$ . If it finds such a query, it returns  $\sigma_{\text{BS}}$ . 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  $\text{bad}$  defined in the security game. Recall that this bad event occurs, if after getting message  $\text{prom}$ , the adversary  $\mathcal{A}$  sends  $\sigma_s$  to oracle  $\text{O}$  such that  $\text{BS.Ver}(\text{pk}_{\text{BS}}, \text{sn}, \sigma_{\text{BS}}) = 0$  and  $\text{SIG.Ver}(\text{pk}_s, \text{tx}, \sigma_s) = 1$ , where  $\sigma_{\text{BS}} \leftarrow \text{Ext}(\text{rpar}, \text{sk}_s, \sigma_s)$ . Due to the definition of algorithm  $\text{Ext}$  this means that the hash value  $\text{H}(\text{sn}, \sigma_{\text{BS}})$  is not defined, where  $\sigma_{\text{BS}}$  is the unique signature satisfying  $\text{BS.Ver}(\text{pk}_{\text{BS}}, \text{sn}, \sigma_{\text{BS}}) = 1$ . The probability that this bad event occurs in the  $i$ -th interaction with oracle  $\text{O}$  can be bounded using a reduction from the EUF-CMA security of  $\text{SIG}$ .

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

## F.2 Proofs for the Schnorr Cut-and-Choose Construction

*Proof (of Lemma 13 (Mal. Service - Schnorr)).* To prove the claim, we present an algorithm  $\text{Ext}$ . It takes as input parameters  $\text{rpar}$ , a promise message  $\text{prom}$ , and a list  $\mathcal{Q}$  of random oracle queries and outputs a blind signature  $\sigma_{\text{BS}}$ . Algorithm  $\text{Ext}(\text{rpar}, \text{prom}, \mathcal{Q})$  is as follows:

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

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

Otherwise, return  $\perp$ .

It remains to prove that for this algorithm  $\text{Ext}$ , the probability that the security game outputs 1 is negligible. In the security game, we define the event  $\text{win}_1$  which occurs if  $\text{VerPromise}(\text{rpar}, \text{prom}) = 1$  and  $\text{Ext}$  outputs  $\perp$ . We also define the event  $\text{win}_2$  which occurs if  $\text{VerPromise}(\text{rpar}, \text{prom}) = 1$ , algorithm  $\text{Ext}$  outputs a valid blind signature  $\sigma_{\text{BS}}$  for  $\text{sn}$ , but for  $\sigma_s \leftarrow \text{Redeem}(\text{rpar}, \text{prom}, \sigma_{\text{BS}})$  we have  $\text{SIG.Ver}(\text{pk}_s, \text{tx}, \sigma_s) = 0$ . Note that whenever algorithm  $\text{Ext}$  does not output  $\perp$ , it outputs a valid blind signature for  $\text{sn}$ . Therefore, the game outputs 1 if and only if  $\text{win}_1$  or  $\text{win}_2$  occurs.

First, we upper bound the probability of  $\text{win}_1$ . To this end, consider the following two events partitioning  $\text{win}_1$ :

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

Clearly, we can bound the probability of  $\text{win}_1$  by bounding the probability of  $\text{win}_{1,1}$  and  $\text{win}_{1,2}$  separately. We start with event  $\text{win}_{1,1}$ . We can assume that  $\text{VerPromise}(\text{rpar}, \text{prom}) = 1$  and therefore  $g^{s_{k_j}} = \prod_{i=0}^{\lambda} (\text{coeff}'_j)^{k_j^i}$  for all  $j \in [\lambda]$ . Note that when the adversary queries  $\text{H}_c(\text{prom}_1)$ , the values  $s_{k_j}$  and  $\text{pk}_{\text{BS}, k_j}$  are information theoretically fixed by the values  $\text{coeff}'_0, (\text{coeff}'_j)_j$  and  $\text{pk}_{\text{BS}}, (\text{coeff}_j)_j$ , respectively. Therefore, the query  $\text{H}_c(\text{prom}_1)$  also fixes the value of  $\Delta := \text{ct}_{k_j} \oplus s_{k_j}$ . If  $\text{VerPromise}(\text{rpar}, \text{prom}) = 1$ , this value must be equal to  $\hat{\text{H}}_q(\text{sn}, \sigma_{k_j})$ . The probability that after  $\Delta$  is fixed, any of the polynomial many queries to  $\hat{\text{H}}_q$  evaluates to  $\Delta$  is negligible. Thus, the probability of  $\text{win}_{1,1}$  is negligible. Next, we bound the probability of event  $\text{win}_{1,2}$ . If this event occurs, we know that at the moment where the adversary queries  $\text{H}_c(\text{prom}_1)$ , it holds that

for all  $j \in [\lambda]$ ,  $\hat{H}_q(\text{sn}, \sigma_{k_j})$  has been queried, and  $\hat{H}_q(\text{sn}, \sigma_{\bar{k}_j})$  has not been queried (due to the definition of algorithm `Ext` and `win1`). Thus, the bits  $b_0, \dots, b_{\lambda-1}$  are fixed before  $H_c(\text{prom}_1)$  is queried, and  $H_c(\text{prom}_1) = b_0, \dots, b_{\lambda-1}$ . This happens with negligible probability  $1/2^\lambda$ .

Next, we bound the probability of event `win2`. By definition of algorithm `VerPromise` we know that  $H_q(\text{coeff}'_0 \cdot (\text{pk}_s)^e, \text{tx}) = e$ . Thus, if `win2` occurs, we know that `Redeem` did not return  $(s, e)$  such that  $g^s = \text{coeff}'_0$ . 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  $\text{coeff}'_0, (\text{coeff}'_j)_j$ . As  $\sigma_{\text{BS}}$  is output by `Ext` and satisfies  $\text{BS.Ver}(\text{pk}_{\text{BS}}, \text{sn}, \sigma_{\text{BS}}) = 1$ , we know that the values  $\sigma_{\bar{k}_j}$  computed in `Redeem` are the unique values satisfying  $\text{BS.Ver}(\text{pk}_{\text{BS}, \bar{k}_j}, \text{sn}, \sigma_{\bar{k}_j}) = 1$ . This means that both the values  $s_{k_j} = \text{ct}_{k_j} \oplus \hat{H}_q(\text{sn}, \sigma_{k_j})$  and  $s_{\bar{k}_j} = \text{ct}_{\bar{k}_j} \oplus \hat{H}_q(\text{sn}, \sigma_{\bar{k}_j})$  are information theoretically fixed at the first time  $H_c(\text{prom}_1)$  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  $b_0, \dots, b_{\lambda-1}$ . Thus, the probability that `win2,1` occurs is at most the probability that  $H_c(\text{prom}_1) = b_0, \dots, b_{\lambda-1}$ , which is negligible.  $\square$

*Proof (of Lemma 14 (Mal. User - Schnorr)).* To prove the claim, we need provide algorithms `Sim`, `SimRO` and `Ext` that share state.

*Simulatability.* Before we provide algorithms `Sim`, `SimRO`, we give a sequence of hybrid games, starting from the simulatability game with bit  $b = 0$  (i.e. computing `prom` via algorithm `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  $(\text{pk}_{\text{BS}} = g_2^{\text{sk}_{\text{BS}}}, \text{sk}_{\text{BS}})$  is sampled and given to the adversary. Then, the adversary gets access to an oracle  $\mathcal{O}$  that on input `sn` aborts if `sn` has already been submitted. Otherwise, it samples Schnorr signing keys  $(\text{pk}_s = g^{\text{sk}_s}, \text{sk}_s)$  and gives  $\text{pk}_s$  to the adversary, receiving `tx` in return. It then defines  $\text{rpar} := (\text{pk}_{\text{BS}}, \text{pk}_s, \text{tx}, \text{sn})$  and outputs  $\text{prom} \leftarrow \text{Promise}(\text{rpar}, \text{sk}_{\text{BS}}, \text{sk}_s)$ . For this scheme, `prom` has the form  $\text{prom} := (\text{prom}_1, \text{prom}_2 := (\sigma_{k_j}, s_{k_j})_{j \in [\lambda]})$  with  $\text{prom}_1 := ((\text{ct}_j)_{j \in [2\lambda]}, (\text{coeff}'_0, e), (\text{coeff}'_j, \text{coeff}'_j)_{j \in [\lambda]})$ . Additionally, the adversary gets access to random oracles  $\hat{H}_q, H, H_c, H_q$  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 `Promise`, the game samples random bits  $b_0, \dots, b_{\lambda-1} \leftarrow \{0, 1\}$ . Then, it computes `prom1` as before. If  $H_c(\text{prom}_1)$  is already defined, the game aborts. Otherwise, it sets  $H_c(\text{prom}_1) := b_0, \dots, b_{\lambda-1}$  and continues the computation of `prom` as before. Note that the probability of such an abort is negligible, due to the entropy of  $\text{coeff}'_0 = g^k \cdot \text{pk}_s^{-e}$ . 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 `prom1`.

**Game  $\mathbf{G}_2$ :** We change how the values  $\text{ct}_{\bar{k}_j}$  for  $j \in [\lambda]$  are computed. Namely, note that they were defined as  $\text{ct}_{\bar{k}_j} := \hat{H}_q(\text{sn}, \sigma_{\bar{k}_j}) \oplus s_{\bar{k}_j}$  before, where  $s_{\bar{k}_j} := f'(\bar{k}_j)$ , and  $\sigma_{\bar{k}_j}$  is the unique value satisfying  $\text{BS.Ver}(\text{pk}_{\text{BS}, \bar{k}_j}, \text{sn}, \sigma_{\bar{k}_j}) = 1$ . From now on,

the game first checks if  $\hat{H}_q(\text{sn}, \sigma_{\bar{k}_j})$  is already defined. Note that the game can do that without knowing  $\text{sk}_{\text{BS}}$  or  $\sigma_{\bar{k}_j}$ , just by iterating over all queries and running  $\text{BS.Ver}$ . If it is already defined, the game sets  $\text{ct}_{\bar{k}_j} := \hat{H}_q(\text{sn}, \sigma_{\bar{k}_j}) \oplus s_{\bar{k}_j}$ . Otherwise, it samples a random  $\text{ct}_{\bar{k}_j} \leftarrow \mathbb{Z}_p$ , and for any subsequent random oracle query  $\hat{H}_q(\text{sn}, \sigma_{\bar{k}_j})$  with  $\text{BS.Ver}(\text{pk}_{\text{BS}, \bar{k}_j}, \text{sn}, \sigma_{\bar{k}_j}) = 1$ , it sets  $\hat{H}_q(\text{sn}, \sigma_{\bar{k}_j}) := \text{ct}_{\bar{k}_j} \oplus s_{\bar{k}_j}$ . It is easy to see that this does not change the view of the adversary. Note that from now on, the values  $\text{sk}_{\text{BS}}, (\text{sk}_{\bar{k}_j})_j$  are no longer needed, except for the computation of  $\text{coeff}_j$ .

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

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

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

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

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

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

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  $\text{prom}$ , oracle  $\mathcal{O}$  gets  $\sigma_s$  in return. The game outputs 1 if in any of these interactions, the event  $\text{bad}$  occurs, i.e. it holds that  $\text{BS.Ver}(\text{pk}_{\text{BS}}, \text{sn}, \sigma_{\text{BS}}) = 0$  and  $\text{SIG.Ver}(\text{pk}_s, \text{tx}, \sigma_s) = 1$ , where  $\sigma_{\text{BS}} \leftarrow \text{Ext}(\text{rpar}, \text{sk}_s, \sigma_s)$ . We distinguish two cases. In the first case, the adversary reuses the exact signature  $(s, e)$  that the game computes during the generation of  $\text{prom}$ . In this case, the adversary implicitly breaks the DLOG assumption by extracting  $s$  from  $\text{coeff}'_0 = g^s$ . 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  $\text{bad}$  into the following two sub-events:

- $\text{bad}_1$ :  $\text{bad}$  occurs and  $\sigma_s$  is sent to  $\mathcal{O}$  by  $\mathcal{A}$ , initiated with  $\text{sn}$  and there exists an entry such that  $\sigma_s = (s, e)$ .
- $\text{bad}_2$ :  $\text{bad}$  occurs and the returned signature  $\sigma_s$  is fresh, i.e.  $\sigma_s \neq (s, e)$ .

We first bound the the probability that event  $\text{bad}_2$  occurs in the  $i$ th interaction with oracle  $\mathcal{O}$ . This is done using a reduction from the  $\text{sEUF-CMA}$  security of  $\text{SIG}$ . We sketch the reduction. The reduction gets as input a public key  $\text{pk}_s^*$  and access to a signing oracle. It simulates the security game honestly, except for the  $i$ th interaction. In this interaction, it uses  $\text{pk}_s := \text{pk}_s^*$  instead of sampling a fresh key pair  $(\text{pk}_s, \text{sk}_s)$ . It also gets the Schnorr signature  $(s, e)$  using the signing oracle. Finally, if event  $\text{bad}_2$  occurs, the reduction can return  $(\text{tx}, \sigma_s)$ , which is a valid forgery. Note that in such a case we have  $\sigma_s \neq (s, e)$ . Therefore, the reduction breaks  $\text{sEUF-CMA}$  security of  $\text{SIG}$ .

Next, we want to bound the probability of event  $\text{bad}_1$ . 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  $q_{\mathcal{O}}$  queries to the oracle  $\mathcal{O}$ , the game picks an index  $i \leftarrow [q_{\mathcal{O}}]$  and aborts in case the event  $\text{bad}_1$  does not occur in the  $i$ th query to  $\mathcal{O}$ . As  $q_{\mathcal{O}}$  is polynomial and the view of the adversary is independent of  $i$ , it is sufficient to bound the probability of  $\text{bad}_1$  in game  $\mathbf{G}_4$ .

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

We show that the probability of event  $\text{bad}_1$  occurring in game  $\mathbf{G}_5$  is negligible, using a reduction to the  $\text{DLOG}$  assumption. We sketch the reduction. It gets as input the instance  $Y = g^\alpha$ . It simulates game  $\mathbf{G}_5$  honestly, except for the  $i$ th interaction of  $\mathcal{A}$  with the oracle  $\mathcal{O}$ . In this interaction, it sets  $\text{coeff}'_0 := Y$  and continues the simulation as in game  $\mathbf{G}_5$ . Note that the polynomial  $f'$  and the discrete logarithm of  $\text{coeff}'_0$  is never needed for that, due to the previous change. In the end, the adversary returns a signature  $\sigma_s$  for which we know that  $\text{SIG.Ver}(\text{pk}_s, \text{tx}, \sigma_s) = 1$  and because of event  $\text{bad}_1$  we know that  $\sigma_s = (\alpha, e)$ . The reduction can return  $\alpha$  as the solution.  $\square$

### F.3 Proofs for the BLS Cut-and-Choose Construction

**Lemma 15.** *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

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

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  $(\mathbf{ct}_0, \mathbf{coeff}'_0, \mathbf{pk}_s) \in \mathbb{G}_1^2 \times \mathbb{G}_2$  and  $(T_1, T_2, T_3) \in \mathbb{G}_1^2 \times \mathbb{G}_2$  from  $\mathcal{A}$ .
3. If  $(\mathbf{ct}_0, \mathbf{coeff}'_0, \mathbf{pk}_s) \in F_{h,\bar{h}}(\mathbb{Z}_p^2)$ , 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  $T_0 = h^{\pi_0} \cdot \bar{h}^{\pi_1} \cdot \mathbf{ct}_0^{-e}$ ,  $T_1 = g_1^{\pi_0} \cdot (\mathbf{coeff}'_0)^{-e}$ , and  $T_2 = g_2^{\pi_1} \cdot (\mathbf{pk}_s)^{-e}$ . 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  $(\mathbf{ct}_0, \mathbf{coeff}'_0, \mathbf{pk}_s)$  which is not in the image of  $F_{h,\bar{h}}$ . We consider two cases. In the first case, assume that for each tuple  $(T_1, T_2, T_3) \in \mathbb{G}_1^2 \times \mathbb{G}_2$ , 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  $(T_1, T_2, T_3) \in \mathbb{G}_1^2 \times \mathbb{G}_2$ , 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,  $(\mathbf{ct}_0, \mathbf{coeff}'_0, \mathbf{pk}_s)$  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

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

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  $(\mathbf{ct}_0, \mathbf{coeff}'_0, \mathbf{pk}_s)$  under  $F_{h,\bar{h}}$ .  $\square$

*Proof (of Lemma 3 (Mal. Service - BLS)).* The proof is almost identical to the proof of Lemma 13, 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  $\sigma_{\text{BS}}$ . The algorithm is as follows:

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



3. Compute  $b_0 \dots b_{\lambda-1} := H_c(\text{prom}_1)$  and for all  $j \in [\lambda]$  compute  $\bar{k}_j := 2j - (1 - b_{j-1})$ .
4. For each  $j \in [\lambda]$  compute  $\text{pk}_{\text{BS}, \bar{k}_j} := \text{pk}_{\text{BS}} \cdot \prod_{i=1}^{\lambda} (\text{coeff}_j)^{\bar{k}_j^i}$ .
5. Find an index  $j^* \in [\lambda]$  and an entry  $((\text{sn}, \sigma_{\bar{k}_{j^*}}), \hat{H}(\text{sn}, \sigma_{\bar{k}_{j^*}}))$  in the list  $\mathcal{Q}$ , such that  $\text{BS.Ver}(\text{pk}_{\text{BS}, \bar{k}_{j^*}}, \text{sn}, \sigma_{\bar{k}_{j^*}}) = 1$ .
6. If such a  $\sigma_{\bar{k}_{j^*}}$  is found for some  $j^* \in [\lambda]$ , return

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

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  $\text{win}_1$  which occurs if  $\text{VerPromise}(\text{rpar}, \text{prom}) = 1$  and `Ext` outputs  $\perp$ . We also define the event  $\text{win}_2$  which occurs if  $\text{VerPromise}(\text{rpar}, \text{prom}) = 1$ , algorithm `Ext` outputs a valid blind signature  $\sigma_{\text{BS}}$  for  $\text{sn}$ , but for  $\sigma_s \leftarrow \text{Redeem}(\text{rpar}, \text{prom}, \sigma_{\text{BS}})$  we have  $\text{SIG.Ver}(\text{pk}_s, \text{tx}, \sigma_s) = 0$ . Note that whenever algorithm `Ext` does not output  $\perp$ , it outputs a valid blind signature for  $\text{sn}$ . Therefore, the game outputs 1 if and only if  $\text{win}_1$  or  $\text{win}_2$  occurs.

First, we upper bound the probability of  $\text{win}_1$ . To this end, consider the following two events partitioning  $\text{win}_1$ :

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

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

Next, we bound the probability of event  $\text{win}_2$ . Consider the values  $h_{k_j}, h_{\bar{k}_j}$  for  $j \in [\lambda]$  as in the definition of algorithm `Redeem`. We partition  $\text{win}_2$  into the following sub-events:

- win<sub>2,1</sub>: win<sub>2</sub> occurs and  $\text{ct}_0 = h^{f'(0)} \cdot \text{H}(\text{tx})^{\text{sk}_s}$ .
- win<sub>2,2</sub>: win<sub>2</sub> occurs and  $\text{ct}_0 \neq h^{f'(0)} \cdot \text{H}(\text{tx})^{\text{sk}_s}$ .

First, assume that win<sub>2,1</sub> 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  $\text{coeff}'_j$  that are contained in **prom**. We know that  $\sigma_{\text{BS}}$  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  $\text{pk}_{\text{BS},k_j}$ . Note that this means that both the values  $h_{k_j} = \text{ct}_{k_j} / \hat{\text{H}}(\text{sn}, \sigma_{k_j})$  and  $h_{\bar{k}_j} = \text{ct}_{\bar{k}_j} / \hat{\text{H}}(\text{sn}, \sigma_{\bar{k}_j})$  are information theoretically fixed at the first time  $\text{H}_c(\text{prom}_1)$  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 win<sub>2,1</sub> occurs is at most the probability that  $\text{H}_c(\text{prom}_1) = b_0, \dots, b_{\lambda-1}$ , which is negligible. Finally, we can bound the probability of win<sub>2,2</sub> by Lemma 15.  $\square$

*Proof (of Lemma 4 (Mal. User - BLS)).* To prove the claim, we need provide algorithms **Sim**, **Sim<sub>RO</sub>** and **Ext** that share state.

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

**Game G<sub>0</sub>:** We start with game **G<sub>0</sub>**, which is the simulatability game with  $b = 0$ . To recall, in this game, a pair of blind signature keys ( $\text{pk}_{\text{BS}} = g_2^{\text{sk}_{\text{BS}}}, \text{sk}_{\text{BS}}$ ) 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 ( $\text{pk}_s = g_2^{\text{sk}_s}, \text{sk}_s$ ) and gives  $\text{pk}_s$  to the adversary, receiving **tx** in return. It then defines  $\text{rpar} := (\text{pk}_{\text{BS}}, \text{pk}_s, \text{tx}, \text{sn})$  and outputs  $\text{prom} \leftarrow \text{Promise}(\text{rpar}, \text{sk}_{\text{BS}}, \text{sk}_s)$ . For this scheme, **prom** has the form  $\text{prom} := (\text{prom}_1, \text{prom}_2 := (\sigma_{k_j}, s_{k_j})_{j \in [\lambda]})$  with  $\text{prom}_1 := (\text{ct}_0, (\text{ct}_j)_{j \in [2\lambda]}, (\pi_0, \pi_1, e), \text{coeff}'_0, (\text{coeff}'_j)_{j \in [\lambda]})$ . Additionally, the adversary gets access to random oracles  $\hat{\text{H}}, \text{H}, \text{H}_c, \text{H}_p$  provided in the standard lazy manner.

**Game G<sub>1</sub>:** In this game, we change how the proofs  $\pi_0, \pi_1, e$  are computed. Namely, they are from now on simulated by sampling  $\pi_0, \pi_1, e \leftarrow \mathbb{Z}_p^*$ , setting  $T_0 := h^{\pi_0} \cdot \text{H}(\text{tx})^{\pi_1} \cdot \text{ct}_0^{-e}$ ,  $T_1 := g_1^{\pi_0} \cdot (\text{coeff}'_0)^{-e}$ , and  $T_2 := g_2^{\pi_1} \cdot (\text{pk}_s)^{-e}$ , and then aborting if  $\text{H}_p(T_0, T_1, T_2, h, \text{H}(\text{tx}), \text{ct}_0, \text{coeff}'_0, \text{pk}_s)$  is already defined, and setting  $\text{H}_p(T_0, T_1, T_2, h, \text{H}(\text{tx}), \text{ct}_0, \text{coeff}'_0, \text{pk}_s) := e$  otherwise. Due to the entropy of  $T_1$ , the probability of a potential abort is negligible. This implies that **G<sub>0</sub>** and **G<sub>1</sub>** are indistinguishable.

**Game G<sub>2</sub>:** We change how queries of the form  $\hat{\text{H}}(\text{sn})$  are answered. Namely, from now on, whenever the hash value is not yet defined, the game first samples a random  $h_{\text{sn}} \leftarrow \mathbb{Z}_p$ , and then sets  $\hat{\text{H}}(\text{sn}) := g_1^{h_{\text{sn}}}$ . Clearly, this does not change the view of the adversary.

**Game G<sub>3</sub>:** We change how the component  $\text{ct}_0$  of **prom** is computed. Namely, note that  $\text{ct}_0$  has been computed via

$$\text{ct}_0 = h^{s_0} \cdot \sigma_s = \hat{\text{H}}(\text{sn})^{s_0} \cdot \sigma_s = g_1^{h_{\text{sn}} s_0} \cdot \sigma_s = \text{coeff}'_0^{h_{\text{sn}}} \cdot \sigma_s.$$

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

**Game  $\mathbf{G}_4$ :** We add another change to the computation of  $\text{prom}$ . Namely, we now sample bits  $b_0, \dots, b_{\lambda-1} \leftarrow_{\text{s}} \{0, 1\}$  in the beginning of algorithm  $\text{Promise}$ . Then, we compute  $\text{prom}_1$  as before and abort if  $H_c(\text{prom}_1)$  is already defined. Otherwise, we set  $H_c(\text{prom}_1) := b_0, \dots, b_{\lambda-1}$  and continue. Note that the probability of such an abort is negligible, due to the entropy of  $\pi_0$ . 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  $\text{prom}_1$ .

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

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

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

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

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

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

1. Let  $\text{sn}, \text{prom}_1, \text{prom}_2, b_0 \dots b_{\lambda-1}$  be as in the execution of  $\text{Sim}$  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  $\text{pk}_{\text{BS}, \bar{k}_j} := \text{pk}_{\text{BS}} \cdot \prod_{i=1}^{\lambda} (\text{coeff}_i)^{\bar{k}_i}$
4. Find an index  $j^* \in [\lambda]$  and an entry  $(\text{sn}, \sigma_{\bar{k}_{j^*}})$  in the list of queries to  $\hat{H}$  such that  $\text{BS.Ver}(\text{pk}_{\text{BS}, \bar{k}_{j^*}}, \text{sn}, \sigma_{\text{BS}, \bar{k}_{j^*}}) = 1$ .

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

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

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 EUF-CMA security of SIG. To this end, our sequence of hybrids eliminates the dependency on  $\text{sk}_s$ .

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

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

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

**Game  $\mathbf{G}_{10}$ :** In this game, we change how the promise message  $\text{prom}$  for the  $i$ th query with  $\text{sn}$  of the adversary to  $\text{O}$ . Precisely, we change the way we compute ciphertext  $\text{ct}_0$  to  $\text{ct}_0 := K$ , for a random  $K \leftarrow_s \mathbb{G}_1$ . 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  $\text{prom}$  honestly as defined in Game  $\mathbf{G}_9$ , but for the  $i$ th interaction it sets  $K := g_1^\gamma \cdot \sigma_s$  and  $\text{coeff}'_0 := g_1^\beta$ . Moreover, the reduction changes the way oracle  $\hat{\text{H}}$  is simulated in the  $i_h$ th query. Namely, for this query, it sets  $h := g_1^\alpha$ . Note that the only place where value  $h_{\text{sn}} = \alpha$  is used is in if the adversary makes query  $\hat{\text{H}}(\text{sn}, \sigma_{\bar{k}_j})$  with  $\text{BS.Ver}(\text{pk}_{\text{BS}, \bar{k}_j}, \text{sn}, \sigma_{\bar{k}_j}) = 1$  for some  $j \in [\lambda]$ . However, if  $\text{bad}$  occurs, this will never happen. If  $\text{bad}$  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  $\text{bad}$  the reduction simulates  $\mathbf{G}_9$  and  $\mathbf{G}_{10}$  otherwise.

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

We sketch the reduction. The reduction gets as input a public key  $\text{pk}_s^*$ . It simulates the security game honestly as in  $\mathbf{G}_{10}$ . In the  $i$ th interaction, it uses  $\text{pk}_s := \text{pk}_s^*$  instead of sampling a fresh key pair  $(\text{pk}_s, \text{sk}_s)$ . The corresponding secret key and a signature  $\sigma_s$  is never needed as already mentioned. Now in case

event **bad** occurs, the reduction can return  $(\text{tx}, \sigma_s)$ . Note that the reduction never used its signing oracle. Therefore, the forgery  $(\text{tx}, \sigma_s)$  is fresh.  $\square$

## G Security Proof of Sweep-UC

**Definition 32.** Let EXC be an exchange for SIG and BS as in Definition 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 33.** Let RP be an redeem protocol for SIG and BS as in Definition 4. 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 1.** 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  $\mathcal{F}_{ux}$  in the synchronous  $(\mathcal{L}^{\text{SIG}}, \mathcal{F}_s)$ -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  $\mathcal{F}_{ux}$ . Especially, a dummy party corresponding to the sweeper  $\mathcal{W}$  forwards messages that are exchanged between  $\mathcal{F}_{ux}$  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  $\mathcal{F}_{ux}$ . Additionally, the ideal functionality  $\mathcal{F}_{ux}$  communicates with the global ledger functionality  $\mathcal{L}^{\text{SIG}}$ . Also, corrupted parties may call this functionality  $\mathcal{L}^{\text{SIG}}$ . 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  $\text{sk}_{\text{BS}}$  to compute the promise message  $\text{prom}$  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  $\text{sk}_{\text{BS}}$  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  $\text{pk}_b$  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  $\text{RP.Sim}$ ,  $\text{RP.Sim}_{\text{RO}}$ ,  $\text{RP.Ext}$  for the redeem protocol RP and simulators  $\text{EXC.Sim}_1$ ,  $\text{EXC.Sim}_{\text{RO}}$ ,  $\text{EXC.Sim}_2$ ,  $\text{EXC.Sim}_3$  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  $\text{sn}$  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  $(\mathcal{P}, \text{pk}_b)$  to tuples  $(\bar{\text{pk}}_{r,\mathcal{W}}, \bar{\text{pk}}_{r,\mathcal{P}}, \bar{\text{sk}}_{r,\mathcal{W}}, \bar{\text{sk}}_{r,\mathcal{P}}, \text{sn})$ . It is used by  $\mathcal{S}$  to store information about the **Register**( $\text{pk}_b$ ) sub-protocol.

- **List Open**: This list contains tuples  $(\text{pk}_a, \text{pk}_c)$ . Whenever a corrupted party  $\mathcal{P}$  completes the **AddPayment** sub-protocol with  $\mathcal{S}$  (in the role of  $\mathcal{W}$ ) for public key  $\text{pk}_a$ , the simulator samples a random key  $\text{pk}_c$  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<sub>1</sub>**: This event occurs if a random nonce is used twice, i.e. an honest party  $\mathcal{P}$  (simulated by  $\mathcal{S}$ ) samples a nonce  $\text{sn}$  in sub-protocol **Register** that is already in **DSpend**.
- **bad<sub>2</sub>**: 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  $\sigma_{\text{BS}}$  on message  $\text{sn}$  for public key  $\text{pk}_{\text{BS}}$  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  $\text{sn}$  is the nonce sent by the adversary in the corresponding execution of sub-protocol **Register**.
- **bad<sub>3</sub>**: This event occurs if the simulator samples a key  $\text{pk}_c$  randomly when a corrupted party interacts in **AddPayment** with  $\mathcal{W}$ , and after that the environment calls **GetPayment**( $\text{pk}_c$ ).
- **bad<sub>4</sub>**: 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<sub>4</sub>** 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  $\mathcal{L}^{\text{SIG}}$ . If on the other hand, these calls are related to protocol interactions, the calls to  $\mathcal{L}^{\text{SIG}}$  are omitted. Here, calls are related to protocol interactions if they are with respect to shared addresses that are used in interactions.

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

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

Register, Corrupted Party  $\mathcal{P}$ :



1. Assume a corrupted  $\mathcal{P}$  with  $\mathcal{S}$ , which plays the role of  $\mathcal{W}$ , and sends  $\text{sn}, \text{pk}_b$  to  $\mathcal{S}$ . Then,  $\mathcal{S}$  first checks if  $\text{sn}$  is already in list  $\text{DSpend}$ . If it is, it aborts this interaction as the honest sweeper would do. Otherwise, it adds  $\text{sn}$  to  $\text{DSpend}$ , and calls the ideal functionality  $\mathcal{F}_{\text{ux}}$  on interface  $\text{Register}(\text{pk}_b)$ . The functionality  $\mathcal{F}_{\text{ux}}$  sends (“register”,  $\mathcal{P}, \text{pk}_b$ ) to  $\mathcal{S}$ , which responds with “noabort”. Then, if  $\mathcal{F}_{\text{ux}}$  responds with “failDoubleRegister” or “failNoFunds”, the simulator aborts the interaction.
2. Otherwise, it simulates opening a shared address for  $\mathcal{P}$ . Concretely, it generates  $(\bar{\text{pk}}_{r,\mathcal{W}}, \bar{\text{sk}}_{r,\mathcal{W}}) \leftarrow \text{SIG.Gen}(1^\lambda)$  and  $(\bar{\text{pk}}_{r,\mathcal{P}}, \bar{\text{sk}}_{r,\mathcal{P}}) \leftarrow \text{SIG.Gen}(1^\lambda)$  on behalf of functionality  $\mathcal{F}_s$ . Then, it sends  $(\bar{\text{pk}}_{r,\mathcal{W}}, \bar{\text{pk}}_{r,\mathcal{P}}, \bar{\text{sk}}_{r,\mathcal{P}})$  to  $\mathcal{P}$  and the message (“openedSharedAddress”,  $\bar{\text{pk}}_{r,\mathcal{W}}, \bar{\text{pk}}_{r,\mathcal{P}}, \text{pk}_{\mathcal{W}}, \text{amt}$ ) on behalf of  $\mathcal{F}_s$  to all parties.
3. Next, it simulates the promise message  $\text{prom}$  for  $\mathcal{P}$ . To do so, it sets a transaction  $\text{tx}_r := (\bar{\text{pk}}_{r,\mathcal{W}}, \bar{\text{pk}}_{r,\mathcal{P}}, \text{pk}_b, \text{amt})$  and redeem parameters  $\text{rpar} := (\text{pk}_{\text{BS}}, \bar{\text{pk}}_{r,\mathcal{W}}, \text{tx}_r, \text{sn})$  as in the protocol. Then, it computes a promise  $\text{prom}$  via  $\text{prom} \leftarrow \text{RP.Sim}(\text{rpar}, \bar{\text{sk}}_{r,\mathcal{W}})$ . From now on, it uses algorithm  $\text{RP.Sim}_{RO}$  to simulate the random oracle related to  $\text{RP}$ .
4. Finally, it sets  $\text{Shared}[\mathcal{P}, \text{pk}_b] := (\bar{\text{pk}}_{r,\mathcal{W}}, \bar{\text{pk}}_{r,\mathcal{P}}, \bar{\text{sk}}_{r,\mathcal{W}}, \bar{\text{sk}}_{r,\mathcal{P}}, \text{sn})$ .

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

1. When the environment calls  $\mathcal{F}_{\text{ux}}$  on interface  $\text{AddPayment}$  via a dummy party,  $\mathcal{S}$  receives a message (“addPayment”,  $\text{pk}_a$ ) from  $\mathcal{F}_{\text{ux}}$ . When  $\mathcal{S}$  receives (“addPaymentFreeze”,  $\text{pk}_a$ ) from  $\mathcal{F}_{\text{ux}}$ , it responds with “noabort”.
2. Then, it generates a shared address as follows: It generates key  $(\bar{\text{pk}}_{l,\mathcal{P}}, \bar{\text{sk}}_{l,\mathcal{P}}) \leftarrow \text{SIG.Gen}(1^\lambda)$  and  $(\bar{\text{pk}}_{l,\mathcal{W}}, \bar{\text{sk}}_{l,\mathcal{W}}) \leftarrow \text{SIG.Gen}(1^\lambda)$ . It sends message (“openedSharedAddress”,  $\bar{\text{pk}}_{l,\mathcal{P}}, \bar{\text{pk}}_{l,\mathcal{W}}, \text{pk}_a, \text{amt}$ ) 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  $\text{tx}_l := (\bar{\text{pk}}_{l,\mathcal{P}}, \bar{\text{pk}}_{l,\mathcal{W}}, \text{pk}_{\mathcal{W}}, \text{amt})$ . Then, it executes  $\sigma_{l,\mathcal{P}} \leftarrow \text{SIG.Sig}(\bar{\text{sk}}_{l,\mathcal{P}}, \text{tx}_l)$  and  $\sigma_{l,\mathcal{W}} \leftarrow \text{SIG.Sig}(\bar{\text{sk}}_{l,\mathcal{W}}, \text{tx}_l)$ . Finally, it sends a message (“closedSharedAddress”,  $\bar{\text{pk}}_{l,\mathcal{P}}, \bar{\text{pk}}_{l,\mathcal{W}}, \text{pk}_{\mathcal{W}}, \text{amt}, \sigma_{l,\mathcal{P}}, \sigma_{l,\mathcal{W}}$ ) on behalf of  $\mathcal{F}_s$  to all parties.

AddPayment, Corrupted Party  $\mathcal{P}$ :

1. Assume a corrupted party sends a message  $\text{bsm}_1$  via an anonymous channel to  $\mathcal{S}$  (which plays the role of  $\mathcal{W}$ ) and opens a shared address using a call  $\mathcal{F}_s.\text{OpenSh}(T, \text{pk}_a, \mathcal{W}, \text{amt}, \text{sk}_a)$ . Then,  $\mathcal{S}$  calls the ideal functionality  $\mathcal{F}_{\text{ux}}$  via interface  $\text{AddPayment}(\text{pk}_a, \text{sk}_a, \text{pk}_c)$  for an arbitrary corrupted party, for some fresh key  $(\text{pk}_c, \text{sk}_c) \leftarrow \text{SIG.Gen}(1^\lambda)$ . If the environment ever queries  $\text{GetPayment}(\text{pk}_c)$  via a dummy party afterwards, the simulator sets  $\text{bad}_3 := 1$  and aborts the entire execution.
2. If  $\mathcal{F}_{\text{ux}}$  sends “failInvalidKey”,  $\mathcal{S}$  sends “failInvalidKey” on behalf of  $\mathcal{F}_s$ . Similarly, if  $\mathcal{F}_{\text{ux}}$  aborts with “failNoFunds”,  $\mathcal{S}$  sends message “failNoFunds” on behalf of  $\mathcal{F}_s$ .

3. If  $\mathcal{F}_{\text{ux}}$  sends (“addPaymentFreeze”,  $\text{pk}_a$ ) to  $\mathcal{S}$ , then  $\mathcal{S}$  computes message  $\text{xm}_1$  using the simulator  $\text{EXC.Sim}_1$ , i.e. it runs  $\text{xm}_1 \leftarrow \text{EXC.Sim}_1(\text{xpar}, \bar{\text{sk}}_{l,\mathcal{W}})$  for  $\text{tx}_l := (\bar{\text{pk}}_{l,\mathcal{P}}, \bar{\text{pk}}_{l,\mathcal{W}}, \text{pk}_{\mathcal{W}}, \text{amt})$  and exchange parameters  $\text{xpar} := (\text{pk}_{\text{BS}}, \text{bsm}_1, \bar{\text{pk}}_{l,\mathcal{P}}, \bar{\text{pk}}_{l,\mathcal{W}}, \text{tx}_l)$ . It sends  $\text{xm}_1$  to the corrupted party.
4. When the corrupted party responds with  $\text{xm}_2$ , the simulator  $\mathcal{S}$  runs  $\sigma_{l,\mathcal{W}} \leftarrow \text{SIG.Sig}(\bar{\text{sk}}_{l,\mathcal{W}}, \text{tx}_l)$  as in the protocol. If  $\text{EXC.Sim}_2(\text{xm}_2) = 0$ , it sends “abort” to  $\mathcal{F}_{\text{ux}}$ . Otherwise, it runs  $\text{bsm}_2 \leftarrow \text{BS.S}(\text{sk}_{\text{BS}}, \text{bsm}_1)$  and  $\sigma_{l,\mathcal{P}} \leftarrow \text{EXC.Sim}_3(\text{xm}_2, \text{bsm}_2)$ , and sends “noabort” to  $\mathcal{F}_{\text{ux}}$ . It inserts  $(\text{pk}_a, \text{pk}_c)$  into list `Open` and sends (“closedSharedAddress”,  $\bar{\text{pk}}_{l,\mathcal{P}}, \bar{\text{pk}}_{l,\mathcal{W}}, \text{pk}_{\mathcal{W}}, \text{amt}, \sigma_{l,\mathcal{P}}, \sigma_{l,\mathcal{W}}$ ) on behalf of  $\mathcal{F}_s$  to all parties.

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

1. When  $\mathcal{Z}$  calls  $\mathcal{F}_{\text{ux}}$  on interface `Register` via a dummy party,  $\mathcal{S}$  receives a notification message (“getPayment”,  $\mathcal{P}, \text{pk}_b$ ) from  $\mathcal{F}_{\text{ux}}$ .
2. Once  $\mathcal{S}$  receives (“gotPayment”,  $\mathcal{P}, \text{pk}_b$ ) from  $\mathcal{F}_{\text{ux}}$ , 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  $(\bar{\text{pk}}_{r,\mathcal{W}}, \bar{\text{pk}}_{r,\mathcal{P}}, \bar{\text{sk}}_{r,\mathcal{W}}, \bar{\text{sk}}_{r,\mathcal{P}}, \text{sn}) := \text{Shared}[\mathcal{P}, \text{pk}_b]$ . Then, it computes a blind signature  $\sigma_{\text{BS}} \leftarrow \text{BS.Sig}(\text{sk}_{\text{BS}}, \text{sn})$ . Next, it runs  $\sigma_{r,\mathcal{W}} \leftarrow \text{Redeem}(\text{rpar}, \text{prom}, \sigma_{\text{BS}})$  and  $\sigma_{r,\mathcal{P}} \leftarrow \text{SIG.Sig}(\text{sk}_{r,\mathcal{P}}, \text{tx}_r)$ . Finally, it sends (“closedSharedAddress”,  $\bar{\text{pk}}_{r,\mathcal{W}}, \bar{\text{pk}}_{r,\mathcal{P}}, \text{pk}_b, \text{amt}, \sigma_{r,\mathcal{W}}, \sigma_{r,\mathcal{P}}$ ) on behalf of  $\mathcal{F}_s$  to all parties.

GetPayment, Corrupted Party  $\mathcal{P}$ :

1. Suppose a corrupted  $\mathcal{P}$  calls interface  $\mathcal{F}_s.\text{CloseSh}(\bar{\text{pk}}_{r,\mathcal{W}}, \bar{\text{pk}}_{r,\mathcal{P}}, \text{pk}_b, \text{amt}, \sigma_{r,\mathcal{W}}, \sigma_{r,\mathcal{P}})$ . If the first two components of  $\text{Shared}[\mathcal{P}, \text{pk}_b]$  is not equal to  $\bar{\text{pk}}_{r,\mathcal{W}}, \bar{\text{pk}}_{r,\mathcal{P}}$ , then  $\mathcal{S}$  processes this call as  $\mathcal{F}_s$  would do, including the calls to  $\mathcal{L}^{\text{SIG}}$ .
2. Otherwise, it restores entry  $(\bar{\text{pk}}_{r,\mathcal{W}}, \bar{\text{pk}}_{r,\mathcal{P}}, \bar{\text{sk}}_{r,\mathcal{W}}, \bar{\text{sk}}_{r,\mathcal{P}}, \text{sn}) := \text{Shared}[\mathcal{P}, \text{pk}_b]$ . Then,  $\mathcal{S}$  sets  $\text{tx}_r := (\bar{\text{pk}}_{r,\mathcal{W}}, \bar{\text{pk}}_{r,\mathcal{P}}, \text{pk}_b, \text{amt})$  and  $\text{rpar} := (\text{pk}_{\text{BS}}, \bar{\text{pk}}_{r,\mathcal{W}}, \text{tx}_r, \text{sn})$ . It extracts a blind signature via  $\sigma_{\text{BS}} \leftarrow \text{RP.Ext}(\text{rpar}, \bar{\text{sk}}_{r,\mathcal{W}}, \sigma_{r,\mathcal{W}})$  from  $\sigma_{r,\mathcal{W}}$ . If  $\text{BS.Ver}(\text{pk}_{\text{BS}}, \text{sn}, \sigma_{\text{BS}}) = 0$ , the simulator  $\mathcal{S}$  sets  $\text{bad}_2 := 1$  and aborts the entire execution.
3. Otherwise, if the list `Open` is empty, it sets  $\text{bad}_3 := 1$  and aborts the entire execution. Otherwise, let  $(\text{pk}_a, \text{pk}_c)$  be an arbitrary entry in `Open` (e.g. the first). Then,  $\mathcal{S}$  removes the entry  $(\text{pk}_a, \text{pk}_c)$  from `Open` and calls the interface `ChangePayment` $(\text{pk}_a, \text{pk}_c, \text{pk}_b)$  of ideal functionality  $\mathcal{F}_{\text{ux}}$ . Note that this interface will not abort, as the party for which the simulator called `AddPayment` $(\text{pk}_a, \cdot, \text{pk}_c)$  must be corrupted.
4. Finally, it calls `GetPayment` $(\text{pk}_b)$ . When it receives (“gotPayment”,  $\mathcal{P}, \text{pk}_b$ ) from  $\mathcal{F}_{\text{ux}}$ , it sends the message (“closedSharedAddress”,  $\bar{\text{pk}}_{r,\mathcal{W}}, \bar{\text{pk}}_{r,\mathcal{P}}, \text{pk}_b, \text{amt}, \sigma_{r,\mathcal{W}}, \sigma_{r,\mathcal{P}}$ ) 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  $\text{bad}_1$ . More precisely, the execution aborts if an honest party  $\mathcal{P}$  samples a nonce  $\text{sn}$  in sub-protocol **Register**, which is already in list  $\text{DSpend}$ .
- $\mathcal{H}_2$ : 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  $\text{sn}, \text{pk}_b$ , instead of computing and sending the promise message  $\text{prom}$  as in the protocol, the message  $\text{prom}$  is now computed as follows: A transaction  $\text{tx}_r := (\bar{\text{pk}}_{r,\mathcal{W}}, \bar{\text{pk}}_{r,\mathcal{P}}, \text{pk}_b, \text{amt})$  and redeem parameters  $\text{rpar} := (\text{pk}_{\text{BS}}, \bar{\text{pk}}_{r,\mathcal{W}}, \text{tx}_r, \text{sn})$  are set as in the protocol. Then,  $\text{prom}$  is computed as  $\text{prom} \leftarrow \text{RP.Sim}(\text{rpar}, \bar{\text{sk}}_{r,\mathcal{W}})$ , and to answer random oracle queries for the redeem protocol, algorithm  $\text{RP.Sim}_{RO}$  is used. Also, we make the change that details about the **Register** protocol are now stored in the map  $\text{Shared}$ , as in the description 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  $(\bar{\text{pk}}_{r,\mathcal{W}}, \bar{\text{pk}}_{r,\mathcal{P}})$  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  $\text{Shared}$ . In this case, the execution runs  $\sigma_{\text{BS}} \leftarrow \text{RP.Ext}(\text{rpar}, \bar{\text{sk}}_{r,\mathcal{W}}, \sigma_{r,\mathcal{W}})$ , where  $\text{rpar}$  and  $\bar{\text{sk}}_{r,\mathcal{W}}$  are restored using  $\text{Shared}$ . Then, it runs  $b := \text{BS.Ver}(\text{pk}_{\text{BS}}, \text{sn}, \sigma_{\text{BS}})$ . If  $b = 0$ , we say that event  $\text{bad}_2$  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  $\sigma_{\text{BS}}$  is used to derive the signature  $\sigma_{r,\mathcal{W}}$  using algorithm **Redeem** from the promise message  $\text{prom}$ . Here,  $\text{prom}$  has been sent from  $\mathcal{W}$  to  $\mathcal{P}$  in sub-protocol **Register** and  $\sigma_{\text{BS}}$  is generated during the sub-protocol **AddPayment**. We make the following change. In this hybrid, we now no longer use  $\sigma_{\text{BS}}$  that was generated in **AddPayment**, but instead generate  $\sigma_{\text{BS}}$  directly via  $\sigma_{\text{BS}} \leftarrow \text{BS.Sig}(\text{sk}_{\text{BS}}, \text{sn})$ , where  $\text{sn}$  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  $\bar{\text{sk}}_{l,\mathcal{P}}$ . More precisely, the execution first generates the keys  $(\bar{\text{pk}}_{l,\mathcal{P}}, \bar{\text{pk}}_{l,\mathcal{W}}, \bar{\text{sk}}_{l,\mathcal{P}}, \bar{\text{sk}}_{l,\mathcal{W}})$  as before. Then, it computes  $\sigma_{l,\mathcal{P}}$  via  $\sigma_{l,\mathcal{P}} \leftarrow \text{SIG.Sig}(\bar{\text{sk}}_{l,\mathcal{P}}, \text{tx}_l)$ , where  $\text{tx}_l$  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  $\sigma_{\text{BS}}$  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  $\text{bsm}_1$  via an anonymous channel to  $\mathcal{W}$ , and opens a shared address using a call  $\mathcal{F}_s.\text{OpenSh}(T, \text{pk}_a, \mathcal{W}, \text{amt}, \text{sk}_a)$ . Then, the sweeper  $\mathcal{W}$  does not compute  $\text{xm}_1$  using algorithm **EXC.Setup** anymore, but instead it uses the algorithms  $\text{EXC.Sim}_1, \text{EXC.Sim}_{RO}, \text{EXC.Sim}_2, \text{EXC.Sim}_3$ . Concretely, it runs

- $xm_1 \leftarrow \text{EXC.Sim}_1(\text{xpar}, \bar{\text{sk}}_{l,\mathcal{W}})$  for  $\text{xpar}$  as before. Then, it sends  $xm_1$  to the corrupted party. When it receives  $xm_2$  in return, it runs  $\sigma_{l,\mathcal{W}} \leftarrow \text{SIG.Sig}(\bar{\text{sk}}_{l,\mathcal{W}}, \text{tx}_l)$  as in the protocol. If  $\text{EXC.Sim}_2(xm_2) = 0$ , it aborts. Otherwise, it runs  $\text{bsm}_2 \leftarrow \text{BS.S}(\text{sk}_{\text{BS}}, \text{bsm}_1)$  and  $\sigma_{l,\mathcal{P}} \leftarrow \text{EXC.Sim}_3(xm_2, \text{bsm}_2)$ . 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  $\text{bsm}_1$  via an anonymous channel to  $\mathcal{W}$  and opens a shared address using a call  $\mathcal{F}_s.\text{OpenSh}(T, \text{pk}_a, \mathcal{W}, \text{amt}, \text{sk}_a)$ , the execution generates  $(\text{pk}_c, \text{sk}_c) \leftarrow \text{SIG.Gen}(1^\lambda)$ . 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  $(\text{pk}_a, \text{pk}_c)$  is inserted into list `Open`. Then, if the environment ever calls `GetPayment`( $\text{pk}_c$ ) afterwards, we say that event  $\text{bad}_3$  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  $\mathcal{F}_s.\text{CloseSh}(\bar{\text{pk}}_{r,\mathcal{W}}, \bar{\text{pk}}_{r,\mathcal{P}}, \text{pk}_b, \text{amt}, \sigma_{r,\mathcal{W}}, \sigma_{r,\mathcal{P}})$ . 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  $(\text{pk}_a, \text{pk}_c)$  from list `Open`. If this fails because the list is empty, we say that  $\text{bad}_4$  occurs and the execution aborts.
  - $\mathcal{H}_9$ : This is the ideal world simulation using simulator  $\mathcal{S}$  as described above.

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

*Proof.* Note that the distinguishing probability of these hybrids can be bounded by the probability of event  $\text{bad}_1$ . As nonces  $\text{sn}$  sampled by honest parties have  $\lambda$  bits of entropy, event  $\text{bad}_1$  can only occur with negligible probability.  $\square$

*Claim.*  $\mathcal{H}_1$  and  $\mathcal{H}_2$  are indistinguishable, if  $(\text{RP.Sim}, \text{RP.Sim}_{RO})$  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  $\text{pk}_{\text{BS}}, \text{sk}_{\text{BS}}$  as input and access to an oracle  $\mathcal{O}$ . It uses  $\text{sk}_{\text{BS}}$  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  $\mathcal{O}$  to simulate message `prom`. Concretely, assume that  $\text{sn}$  is not yet in `DSpend`. Then, to compute message `prom`, the reduction sends  $\text{sn}$  to  $\mathcal{O}$  and gets a key  $\bar{\text{pk}}_{r,\mathcal{W}}$  in return. It generates  $\bar{\text{pk}}_{r,\mathcal{P}}$  and sets  $\text{tx}_r$  as in the protocol. Then, it sends  $\text{tx}_r$  to  $\mathcal{O}$  and obtains `prom` from  $\mathcal{O}$ . 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  $\mathcal{Z}$ .  $\square$

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

*Proof.* To show the claim, we sketch a reduction from the extractability game of  $\text{RP}$ . The reduction gets  $\text{pk}_{\text{BS}}, \text{sk}_{\text{BS}}$  as input and access to an oracle  $\text{O}$ . 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  $\text{O}$  as follows. When  $\mathcal{P}$  sends a nonce  $\text{sn}$  and a public key  $\text{pk}_{b_2}$  the reduction passes  $\text{sn}$  to  $\text{O}$ . It obtains a key  $\bar{\text{pk}}_{r,\mathcal{W}}$  in return, and generates  $\text{pk}_{r,\mathcal{P}}$  and sets  $\text{tx}_r$  as in the protocol. It sends  $\text{tx}_r$  to  $\text{O}$ , and obtains message  $\text{prom}$  in return. The reduction sends  $\text{prom}$  to  $\mathcal{P}$ , as in the protocol. Later, when a party closes the shared address  $(\bar{\text{pk}}_{r,\mathcal{W}}, \text{pk}_{r,\mathcal{P}})$  using signatures  $\sigma_{r,\mathcal{W}}, \sigma_{r,\mathcal{P}}$ , the reduction passes  $\sigma_{r,\mathcal{W}}$  to oracle  $\text{O}$ . 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  $\text{bad}$  defined in the extractability game of  $\text{RP}$  is set to 1 if and only if event  $\text{bad}_2$  occurs. Thus, we can bound the probability of event  $\text{bad}_2$  by the advantage of the above reduction. Clearly, the distinguishing advantage is upper bounded by the probability of  $\text{bad}_2$ .  $\square$

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

*Proof.* As  $\text{BS}$  has unique signatures, the distribution of  $\sigma_{\text{BS}}$  computed directly (as in  $\mathcal{H}_4$ ) is the same as the distribution of  $\sigma_{\text{BS}}$  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.  $\square$

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

*Proof.* This follows directly from the definition of well distributed signatures.  $\square$

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

*Proof.* Note that due to the previous changes, the secret key  $\text{sk}_{\text{BS}}$  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  $\text{EXC}$  that interpolates between  $\mathcal{H}_5$  and  $\mathcal{H}_6$ . The reduction gets  $\text{pk}_{\text{BS}}$  as input and access to an oracle  $\text{O}^*$  and a signing oracle  $\text{O}$ . 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  $\sigma_{\text{BS}}$  is computed using the signing oracle  $\text{O}$ . Second, when a corrupted party interacts with  $\mathcal{W}$  in **AddPayment**, the oracle  $\text{O}^*$  is used to simulate the exchange. Concretely, when the corrupted party sends  $\text{bsm}_1$  to  $\mathcal{W}$  and opens a shared address, the reduction calls oracle  $\text{O}^*$  and obtains a key  $\bar{\text{pk}}_{l,\mathcal{W}}$ . This key is then used as part of the shared address  $(\bar{\text{pk}}_{l,\mathcal{P}}, \bar{\text{pk}}_{l,\mathcal{W}})$ . Then, the reduction defines a transaction  $\text{tx}_l$  as in the protocol and sends  $\bar{\text{pk}}_{l,\mathcal{P}}, \text{tx}_l$  and  $\text{bsm}_1$  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.  $\square$

*Claim.*  $\mathcal{H}_6$  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  $\text{bad}_3$ . Note that the environment obtains no information about the key  $\text{pk}_c$ . Therefore, the probability that the environment queries **GetPayment** for that key is negligible, by the assumption about entropy of public keys.  $\square$

*Claim.*  $\mathcal{H}_7$  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  $\text{bad}_4$ . We bound the probability of  $\text{bad}_4$  using a reduction against the one-more unforgeability of BS. The reduction gets  $\text{pk}_{\text{BS}}$  as input and access to a signer oracle  $O$ . It simulates  $\mathcal{H}_7$ , with the following modifications: First, to compute the blind signature  $\sigma_{\text{BS}}$  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  $\text{bsm}_1$ , obtaining  $\text{bsm}_2$  in return. We call these queries *queries of the second kind*. Then, it runs  $\sigma_{l,\mathcal{P}} \leftarrow \text{EXC.Sim}_3(xm_2, \text{bsm}_2)$  as in  $\mathcal{H}_7$ . When event  $\text{bad}_4$  occurs, let  $\Sigma_{\text{hon}}$  denote the list of pairs  $(\text{sn}, \sigma_{\text{BS}})$  that are computed by honest parties. Let  $\Sigma_{\text{corr}}$  denote the list of pairs  $(\text{sn}, \sigma_{\text{BS}})$ , for which the execution extracted the blind signature  $\sigma_{\text{BS}}$  for  $\text{sn}$  when a corrupted party closed a shared address that has been opened in **Register**. The reduction outputs  $\Sigma_{\text{hon}} \cup \Sigma_{\text{corr}}$ .

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  $\text{bad}_4$  occurs. To see that, note that due to the usage of list **DSpend** and the event  $\text{bad}_1$ , we know that all  $\text{sn}$  in the reductions final output are distinct. Further, all  $\sigma_{\text{BS}}$  are valid. This is because  $\sigma_{\text{BS}}$  in  $\Sigma_{\text{hon}}$  are computed honestly, and  $\sigma_{\text{BS}}$  in  $\Sigma_{\text{corr}}$  are valid by the definition of  $\text{bad}_2$ . It remains to argue that the reduction returned more pairs than the number of queries to the signer oracle  $O$ .

Let  $k_{\text{add}}$  denote the number of entries that are added to list **Open**, and  $k_{\text{rem}}$  the number of times the reduction tried to remove an entry from list **Open**. If  $\text{bad}_4$  occurs, we have

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

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  $|\Sigma_{\text{hon}}|$ . Therefore, the number of queries that the reduction made is

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

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

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

□

*Claim.*  $\mathcal{H}_8$  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  $\mathcal{F}_{ux}$  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  $\mathcal{F}_{ux}$ . 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  $\mathcal{F}_{ux}$  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  $\mathbf{pk}_a$  pays, but it does not see which dummy party has been called and which public key  $\mathbf{pk}_b$  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 transferred to  $\mathbf{pk}_b$ . 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  $\mathcal{F}_{\text{ux}}$  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  $\text{sn}$  to derive the transaction signature, where  $\text{sn}$  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  $\text{sn}$ . 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  $\text{prom}$  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  $\text{sn}$  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  $(\mathcal{P}, \text{pk}_b)$  to tuples  $(\bar{\text{pk}}_{r,\mathcal{W}}, \bar{\text{pk}}_{r,\mathcal{P}}, \bar{\text{sk}}_{r,\mathcal{W}}, \bar{\text{sk}}_{r,\mathcal{P}}, \text{sn}, \sigma_{r,\mathcal{W}})$ . It is used by  $\mathcal{S}$  to store information about the **Register**( $\text{pk}_b$ ) 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**<sub>1</sub>: 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  $\text{sn}$  in sub-protocol **Register** that is already in **DSPend**.
- **bad**<sub>2</sub>: This event occurs if the algorithm **RP.Ext** can not extract a valid blind signature  $\sigma_{\text{BS}}$  from the promise message  $\text{prom}$  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  $\text{sn}, \text{pk}_b$ , and  $\mathcal{W}$  sends  $\text{prom}$ , let  $\sigma_{\text{BS}} \leftarrow \text{RP.Ext}(\text{rpar}, \text{prom}, \mathcal{Q})$  and  $\sigma_{r,\mathcal{W}} \leftarrow \text{Redeem}(\text{rpar}, \text{prom}, \sigma_{\text{BS}})$ , where  $\mathcal{Q}$  is the list of random oracle queries that corrupted parties made. Then, the bad event occurs, if we have  $\text{BS.Ver}(\text{pk}_{\text{BS}}, \text{sn}, \sigma_{\text{BS}}) = 0$  or  $\text{SIG.Ver}(\bar{\text{pk}}_{r,\mathcal{W}}, \text{tx}_r, \sigma_{r,\mathcal{W}}) = 0$ . Here,  $\bar{\text{pk}}_{r,\mathcal{W}}, \text{tx}_r$ , and  $\text{rpar}$  are as in the protocol.
- **bad**<sub>3,1</sub>: 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  $\text{sn}$  into **BS.U**<sub>1</sub> and sends the resulting message  $\text{bsm}_1$  to  $\mathcal{W}$ . Next, it opens a shared address  $(\bar{\text{pk}}_{l,\mathcal{P}}, \bar{\text{pk}}_{l,\mathcal{W}})$



- 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  $(\bar{pk}_{l,\mathcal{P}}, \bar{pk}_{l,\mathcal{W}})$  using signatures  $(\sigma_{l,\mathcal{P}}, \sigma_{l,\mathcal{W}})$ . Honest party  $\mathcal{P}$  runs  $\text{bsm}_2 := \text{Get}(xpar, xm_1, xm_2, \sigma_{l,\mathcal{P}}, \sigma_{l,\mathcal{W}})$  and computes  $\sigma_{BS}$  from  $\text{bsm}_2$  using algorithm  $\text{BS.U}_2$ . Then, the bad event occurs if  $\text{BS.Ver}(pk_{BS}, sn, \sigma_{BS}) = 0$ .
- $\text{bad}_{3,2}$ : This event occurs if in the same situation as for  $\text{bad}_{3,1}$ ,  $\mathcal{W}$  closes the shared address  $(\bar{pk}_{l,\mathcal{P}}, \bar{pk}_{l,\mathcal{W}})$  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  $\mathcal{L}^{\text{SIG}}$ . 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  $\mathcal{L}^{\text{SIG}}$  are omitted (this is because in such a case these calls are issued by functionality  $\mathcal{F}_{ux}$ ). 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  $\mathcal{F}_{ux}$  on interface **Register** via a dummy party,  $\mathcal{S}$  receives a notification message (“register”,  $\mathcal{P}, pk_b$ ) from  $\mathcal{F}_{ux}$ . Then, it samples a random nonce  $sn \leftarrow_{\$} \{0, 1\}^\lambda$ . If  $sn$  is already in list  $\text{DSpend}$ , it sets  $\text{bad}_1 := 1$  and aborts the execution. Otherwise, it adds  $sn$  to list  $\text{DSpend}$  and sends  $sn, pk_b$  to the corrupted  $\mathcal{W}$ .
2. When  $\mathcal{W}$  calls  $\mathcal{F}_s.\text{OpenSh}(T, pk_{\mathcal{W}}, \mathcal{P}, amt, sk_{\mathcal{W}})$ , the simulator  $\mathcal{S}$  simulates the interface **OpenSh**, except for the calls to  $\mathcal{L}^{\text{SIG}}$ . During this simulation, it generates  $(pk_{r,\mathcal{W}}, sk_{r,\mathcal{W}}) \leftarrow \text{SIG.Gen}(1^\lambda)$  and  $(\bar{pk}_{r,\mathcal{P}}, \bar{sk}_{r,\mathcal{P}}) \leftarrow \text{SIG.Gen}(1^\lambda)$  on behalf of  $\mathcal{F}_s$ . Once  $\mathcal{S}$  receives (“registered”,  $\mathcal{P}, pk_b$ ) from  $\mathcal{F}_{ux}$ , it sends (“openedSharedAddress”,  $\bar{pk}_{r,\mathcal{W}}, \bar{pk}_{r,\mathcal{P}}, pk_{\mathcal{W}}, amt$ ) on behalf of  $\mathcal{F}_s$  to all parties.
3. The simulator  $\mathcal{S}$  sets  $tx_r := (\bar{pk}_{r,\mathcal{W}}, \bar{pk}_{r,\mathcal{P}}, pk_b, amt)$  and  $rpar := (pk_{BS}, \bar{pk}_{r,\mathcal{W}}, tx_r, sn)$  as an honest party would do in the protocol. Then, when  $\mathcal{W}$  sends the promise message  $\text{prom}$ , the simulator  $\mathcal{S}$  checks if  $\text{VerPromise}(rpar, \text{prom}) = 1$ . If this does not hold, it sends “abort” to  $\mathcal{F}_{ux}$ .
4. Otherwise,  $\mathcal{S}$  runs  $\sigma_{BS} \leftarrow \text{RP.Ext}(rpar, \text{prom}, \mathcal{Q})$  and  $\sigma_{r,\mathcal{W}} \leftarrow \text{Redeem}(rpar, \text{prom}, \sigma_{BS})$ , where  $\mathcal{Q}$  is the list of random oracle queries that corrupted parties made so far. Then, if  $\text{BS.Ver}(pk_{BS}, sn, \sigma_{BS}) = 0$  or  $\text{SIG.Ver}(\bar{pk}_{r,\mathcal{W}}, tx_r, \sigma_{r,\mathcal{W}}) = 0$ , the simulator sets  $\text{bad}_2 := 1$  and aborts the execution.
5. The simulator  $\mathcal{S}$  sets  $\text{Shared}[\mathcal{P}, pk_b] := (\bar{pk}_{r,\mathcal{W}}, \bar{pk}_{r,\mathcal{P}}, \bar{sk}_{r,\mathcal{W}}, \bar{sk}_{r,\mathcal{P}}, sn, \sigma_{r,\mathcal{W}})$ .

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

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

2. The simulator  $\mathcal{S}$  samples  $\text{sn}' \leftarrow_{\mathcal{S}} \{0, 1\}^\lambda$ , runs  $(\text{bsm}_1, St) \leftarrow \text{BS.U}_1(\text{pk}_{\text{BS}}, \text{sn}')$  and sends  $\text{bsm}_1$  to  $\mathcal{W}$  via the anonymous channel.
3. When  $\mathcal{S}$  receives (“addPaymentFreeze”,  $\text{pk}_a$ ) from  $\mathcal{F}_{\text{ux}}$ , it simulates the opening of a shared address as follows: It generates keys  $(\text{pk}_{l,\mathcal{P}}, \bar{\text{sk}}_{l,\mathcal{P}}) \leftarrow \text{SIG.Gen}(1^\lambda)$  and  $(\bar{\text{pk}}_{l,\mathcal{W}}, \bar{\text{sk}}_{l,\mathcal{W}}) \leftarrow \text{SIG.Gen}(1^\lambda)$ . It sends (“openedSharedAddress”,  $\bar{\text{pk}}_{l,\mathcal{P}}, \bar{\text{pk}}_{l,\mathcal{W}}, \text{pk}_a, \text{amt}$ ) on behalf of the functionality  $\mathcal{F}_s$  to all parties.
4. If this shared address  $(\bar{\text{pk}}_{l,\mathcal{P}}, \bar{\text{pk}}_{l,\mathcal{W}})$  is closed by a corrupted party before the message  $\text{xm}_2$  (see below) is sent,  $\mathcal{S}$  sets  $\text{bad}_{3,2} := 1$  and aborts the entire execution. If  $\mathcal{W}$  does not send  $\text{xm}_1$ , then  $\mathcal{S}$  sends “abort” to  $\mathcal{F}_{\text{ux}}$ .
5. The simulator  $\mathcal{S}$  sets  $\text{tx}_l := (\bar{\text{pk}}_{l,\mathcal{P}}, \bar{\text{pk}}_{l,\mathcal{W}}, \text{pk}_{\mathcal{W}}, \text{amt})$  and  $\text{xpar} := (\text{pk}_{\text{BS}}, \text{bsm}_1, \bar{\text{pk}}_{l,\mathcal{P}}, \bar{\text{pk}}_{l,\mathcal{W}}, \text{tx}_l)$  as in the protocol. When  $\mathcal{W}$  sends  $\text{xm}_1$ , the simulator runs  $\text{xm}_2 \leftarrow \text{Buy}(\text{xpar}, \bar{\text{sk}}_{l,\mathcal{P}}, \text{xm}_1)$  and sends  $\text{xm}_2$  to  $\mathcal{W}$ .
6. When  $\mathcal{W}$  closes the shared address  $(\bar{\text{pk}}_{l,\mathcal{P}}, \bar{\text{pk}}_{l,\mathcal{W}})$  via  $\mathcal{F}_s.\text{CloseSh}(\bar{\text{pk}}_{l,\mathcal{P}}, \bar{\text{pk}}_{l,\mathcal{W}}, \text{pk}_{\mathcal{W}}, \text{amt}, \sigma_{l,\mathcal{P}}, \sigma_{l,\mathcal{W}})$ ,  $\mathcal{S}$  simulates  $\text{CloseSh}$  except for calls to  $\mathcal{L}^{\text{SIG}}$ , and sends “noabort” to  $\mathcal{F}_{\text{ux}}$ . During that, it also sends (“closedSharedAddress”,  $\bar{\text{pk}}_{l,\mathcal{P}}, \bar{\text{pk}}_{l,\mathcal{W}}, \text{pk}_{\mathcal{W}}, \text{amt}, \sigma_{l,\mathcal{P}}, \sigma_{l,\mathcal{W}}$ ) on behalf of  $\mathcal{F}_s$  to all parties. Then, it runs  $\text{bsm}_2 := \text{Get}(\text{xpar}, \text{xm}_1, \text{xm}_2, \sigma_{l,\mathcal{P}}, \sigma_{l,\mathcal{W}})$  and  $\sigma_{\text{BS}} \leftarrow \text{BS.U}_2(St, \text{bsm}_2)$ . It sets  $\text{bad}_{3,1} := 1$  and aborts the entire execution if  $\text{BS.Ver}(\text{pk}_{\text{BS}}, \text{sn}, \sigma_{\text{BS}}) = 0$ .

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

1. When  $\mathcal{Z}$  calls  $\mathcal{F}_{\text{ux}}$  on interface **GetPayment** via a dummy party,  $\mathcal{S}$  receives a notification message (“getPayment”,  $\mathcal{P}_i, \text{pk}_b$ ) from  $\mathcal{F}_{\text{ux}}$ .
2. Once  $\mathcal{S}$  receives (“gotPayment”,  $\mathcal{P}, \text{pk}_b$ ) from  $\mathcal{F}_{\text{ux}}$ , it sets  $(\bar{\text{pk}}_{r,\mathcal{W}}, \bar{\text{pk}}_{r,\mathcal{P}}, \bar{\text{sk}}_{r,\mathcal{W}}, \bar{\text{sk}}_{r,\mathcal{P}}, \text{sn}, \sigma_{r,\mathcal{W}}) := \text{Shared}[\mathcal{P}, \text{pk}_b]$ . It computes  $\sigma_{r,\mathcal{P}} \leftarrow \text{SIG.Sig}(\bar{\text{sk}}_{r,\mathcal{P}}, \text{tx}_r)$ .
3. Finally, it sends (“closedSharedAddress”,  $\bar{\text{pk}}_{r,\mathcal{W}}, \bar{\text{pk}}_{r,\mathcal{P}}, \text{pk}_b, \text{amt}, \sigma_{r,\mathcal{W}}, \sigma_{r,\mathcal{P}}$ ) 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  $\text{DSpend}$  denote the list of nonces  $\text{sn}$  used by honest parties, as it is used by  $\mathcal{S}$ .
- $\mathcal{H}_1$ : In this hybrid, the execution aborts whenever event  $\text{bad}_1$  occurs. That is, if an honest party samples a nonce  $\text{sn}$  that is already in list  $\text{DSpend}$ .
- $\mathcal{H}_2$ : In this hybrid, we change how **Register** is executed for honest parties  $\mathcal{P}$ . Namely, when  $\mathcal{W}$  sends the promise  $\text{prom}$ , the execution runs  $\sigma_{\text{BS}} \leftarrow \text{RP.Ext}(\text{rpar}, \text{prom}, \mathcal{Q})$  and  $\sigma_{r,\mathcal{W}} \leftarrow \text{Redeem}(\text{rpar}, \text{prom}, \sigma_{\text{BS}})$ , where  $\mathcal{Q}$  is the list of random oracle queries that corrupted parties made so far. If  $\text{BS.Ver}(\text{pk}_{\text{BS}}, \text{sn}, \sigma_{\text{BS}}) = 0$  or  $\text{SIG.Ver}(\bar{\text{pk}}_{r,\mathcal{W}}, \text{tx}_r, \sigma_{r,\mathcal{W}}) = 0$ , we say that the event  $\text{bad}_2$  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  $\mathbf{bad}_{3,1}$  or  $\mathbf{bad}_{3,2}$  occur. Concretely, in an execution of the sub-protocol **AddPayment** for honest party  $\mathcal{P}$ , the event  $\mathbf{bad}_{3,1}$  occurs if no valid blind signature  $\sigma_{\text{BS}}$  can be obtained from the signatures  $(\sigma_{l,\mathcal{P}}, \sigma_{l,\mathcal{W}})$  using algorithms **Get** and **BS.U<sub>2</sub>**. The event  $\mathbf{bad}_{3,2}$  occurs if a corrupted party closes the shared address  $(\bar{\mathbf{pk}}_{l,\mathcal{P}}, \bar{\mathbf{pk}}_{l,\mathcal{W}})$  before the honest party  $\mathcal{P}$  sends  $\mathbf{xm}_2$ .
- $\mathcal{H}_4$ : 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  $\mathbf{sn}$  are blindly signed in executions of **AddPayment** for honest parties  $\mathcal{P}$ . Recall that in previous hybrids, party  $\mathcal{P}$  runs  $(\mathbf{bsm}_1, St) \leftarrow \text{BS.U}_1(\mathbf{pk}_{\text{BS}}, \mathbf{sn})$ , sends  $\mathbf{bsm}_1$  to  $\mathcal{W}$  and interacts in the exchange protocol with  $\mathcal{W}$ . Here,  $\mathbf{sn}$  is the random nonce sampled by  $\mathcal{P}$  in the corresponding execution of **Register**. In this hybrid,  $\mathcal{P}$  instead samples a random  $\mathbf{sn}' \leftarrow_{\$} \{0, 1\}^\lambda$  and computes  $(\mathbf{bsm}_1, St) \leftarrow \text{BS.U}_1(\mathbf{pk}_{\text{BS}}, \mathbf{sn}')$ . Later, to check if event  $\mathbf{bad}_{3,1}$  occurs, nonce  $\mathbf{sn}'$  is also used instead of  $\mathbf{sn}$ .
- $\mathcal{H}_6$ : This is the ideal world simulation using simulator  $\mathcal{S}$  as described above.

*Claim.*  $\mathcal{H}_0$  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  $\mathbf{bad}_1$ . As nonces  $\mathbf{sn}$  are sampled uniformly at random in  $\{0, 1\}^\lambda$ , the probability of  $\mathbf{bad}_1$  is negligible.  $\square$

*Claim.*  $\mathcal{H}_1$  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  $\mathbf{bad}_2$  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  $\mathbf{pk}_{r,\mathcal{W}}, \mathbf{tx}_r, \mathbf{sn}, \mathbf{pk}_{\text{BS}}$  and **prom** to its game.

It is clear that the reduction perfectly simulates  $\mathcal{H}_{1,i-1}$ . Also, the conditions defining  $\mathbf{bad}_2$  are exactly the winning conditions in the security game of **RP**.  $\square$

*Claim.*  $\mathcal{H}_2$  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  $\text{bad}_{3,1}$  or  $\text{bad}_{3,2}$  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  $\text{pk}_{\text{BS}}$  and  $\text{sn}$  to the security game. Then, it obtains a key  $\bar{\text{pk}}_{l,\mathcal{P}}$  and a message  $\text{bsm}_1$  in return. It simulates the opening of a shared address  $(\bar{\text{pk}}_{l,\mathcal{P}}, \bar{\text{pk}}_{l,\mathcal{W}})$ , using the key that it got from the game. Then, it sends  $\text{bsm}_1$  to  $\mathcal{W}$  as in the protocol. If the reduction did not receive  $\text{xm}_1$  from  $\mathcal{W}$ , it sets  $\text{xm}_1 := \perp$ . This includes the case where a corrupted party already closed the shared address (cf. event  $\text{bad}_{3,2}$ ). Then, the reduction sends  $\bar{\text{pk}}_{l,\mathcal{W}}$ ,  $\text{tx}_l$ , and  $\text{xm}_1$  to the game, where  $\text{tx}_l$  is as in the protocol. It obtains  $\text{xm}_2$  in return. If  $\text{xm}_2 \neq \perp$ , it sends  $\text{xm}_2$  to  $\mathcal{W}$ . Once a corrupted party (e.g.  $\mathcal{W}$ ) closes the shared address  $(\bar{\text{pk}}_{l,\mathcal{P}}, \bar{\text{pk}}_{l,\mathcal{W}})$  using signatures  $(\sigma_{l,\mathcal{P}}, \sigma_{l,\mathcal{W}})$ , the reduction returns  $\text{tx}_l$  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  $\text{bad}_{3,1}$  and  $\text{bad}_{3,2}$ , the security game of **EXC** outputs 1 if one of these events occurs in the  $i$ -th interaction.  $\square$

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

*Proof.* Note that the difference between both hybrids is how the blind signature  $\sigma_{\text{BS}}$  that is input into algorithm **Redeem** is computed by honest parties. In both hybrids,  $\sigma_{\text{BS}}$  is a valid blind signature for nonce  $\text{sn}$  with respect to public key  $\text{pk}_{\text{BS}}$ . 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.  $\square$

*Claim.*  $\mathcal{H}_4$  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  $i \in \{0, \dots, Q\}$ , 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  $\text{bad}_{3,1}$ ). 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  $\text{sn}' \leftarrow_s \{0, 1\}^\lambda$  and

outputs  $\text{pk}_{\text{BS}}, \mathbf{m}_0 := \text{sn}$  and  $\mathbf{m}_1 := \text{sn}'$  to its game. Here,  $\text{sn}$  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  $\text{bsm}_1$  to the reduction. Then, the reduction continues the simulation of the **AddPayment** interaction as in  $\mathcal{H}_4$ , using this message  $\text{bsm}_1$ . When a corrupted party closes the shared address and event  $\text{bad}_{3,2}$  did not happen, the reduction extracts  $\text{bsm}_2$  using algorithm **Get**. Then, the reduction outputs  $\text{bsm}_2$  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  $\text{bad}_{3,1} := 1$  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  $b = 1$ .  $\square$

*Claim.*  $\mathcal{H}_5$  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  $\mathcal{F}_{\text{ux}}$  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.  $\square$

$\square$

## H BLS Signatures and Blind Signatures

For completeness, we recall the BLS signature scheme [14] and its blind version [12]. We denote the signature scheme by  $\text{SIG} = (\text{Gen}, \text{SIG.Sig}, \text{Ver})$  and the blind signature scheme by  $\text{BS} = (\text{Gen}, \text{BS.S}, \text{BS.U}, \text{Ver})$ . 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  $e : \mathbb{G}_1 \times \mathbb{G}_2$  is a pairing. Also, they require a random oracle  $H : \{0, 1\}^* \rightarrow \mathbb{G}_1$ .

Algorithm  $\text{Gen}(1^\lambda)$  first generates such parameters, then it samples a secret key  $\text{sk} \leftarrow_{\$} \mathbb{Z}_p$ , and defines the public key  $\text{pk} := g_2^{\text{sk}}$ . Then it returns  $(\text{pk}, \text{sk})$ . Signatures are computed via

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

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

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

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

## I Interpolation with Preprocessing

We sketch how to improve computation costs of interpolation in the exponent (i.e. algorithm  $\text{reconst}_{g,z}$ ), 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  $\text{reconst}_{g,z}((x_0, h_0), \dots, (x_\lambda, h_\lambda))$  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  $\ell_j$  such that

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

Then, for each  $i \in \{0, \dots, \text{secpair}\}$  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  $\text{reconst}_{g,z}$  by

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

**Multiple Last Samples.** Suppose we know  $\lambda$  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  $\text{reconst}_{g,z}$  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  $(x_\lambda, h_\lambda)$ . 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  $\ell_\lambda$  is defined as

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

## J Details on Experiments

We focused on the Schnorr variant of our cut-and-choose approach 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 open source implementation of the BLS12-381 pairing friendly curve<sup>17</sup> with slight modifications to allow lower-level EC operations. Using this library allows to easily implement the blind signature part. To simplify the implementation, we reused the group  $\mathbb{G}_1$  for the Schnorr signature since it is a standard elliptic curve. 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, but the goal of our prototype implementation was just to show practically, and we leave an optimized implementation as future work.

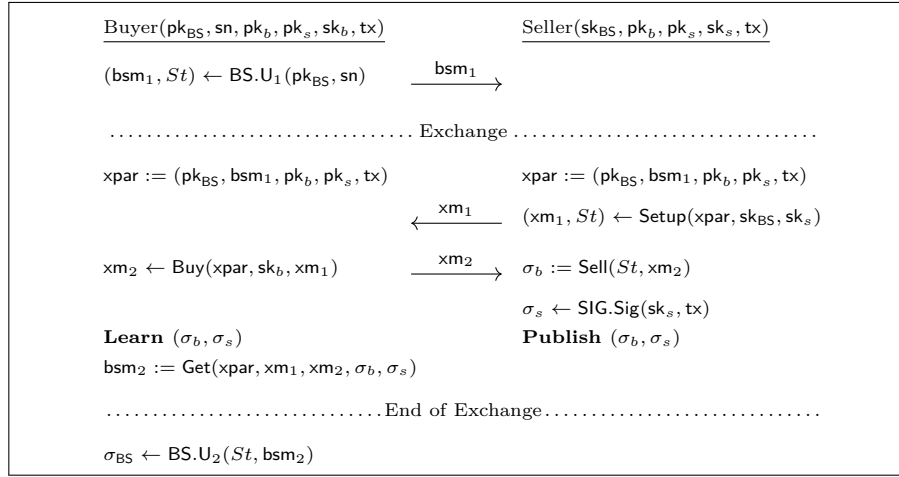
We evaluated our implementation on a MacbookPro with Intel i7@2.3 GHz and 16 GB RAM. The Intel i7 has four physical cores, so we used 16 workers at a time for the parallel execution. The Benchmark of the prototype given in Table 2 is an average of over 100 tests. The results clearly show that our solution is practical. In particular, the sweeper can setup an exchange and create a promise in less than a second. In practice, the code of the sweeper will be executed on a server with more power and physical cores, significantly reducing this time. We did not include `EXC.Sell` in Table 2 since it consists just of on-chain signature verification, which is already considered practical and used in practice.

The most time-consuming operation for the exchange and redeem protocol are the buying process and the promise verification. In both cases, the user verifies the cut-and-choose proof (algorithms `EXC.Buy` and `RP.VerPromise`) created by the sweeper. Fortunately, as shown in Table 2, both take around 5 seconds on a standard laptop. It is worth noting that despite this check, the sweeper's undisclosed values are not necessarily correct, and it is ensured that among the  $\lambda$  undisclosed values, there is at least one correctly created one. If the sweeper is honest, all values of the cut-and-choose will be correct. For example, the check in Figure 3, Line 26 will pass in the first iteration. Thus, for an honest sweeper, algorithms `EXC.Get` and `RP.Redeem` terminate early and take less time (less than a second). Moreover, even if the sweeper is malicious, users can still finalize the exchange/redeeming in less than half a minute.

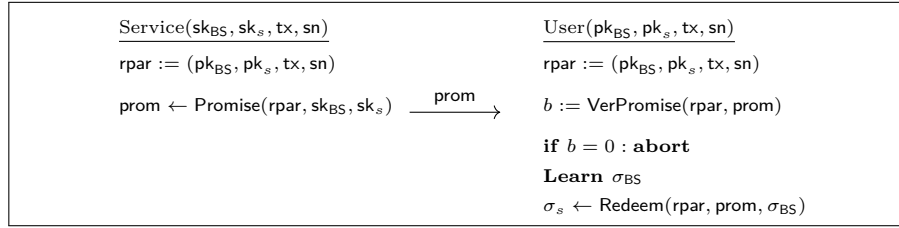
---

<sup>17</sup> See <https://github.com/Chia-Network/bls-signatures>

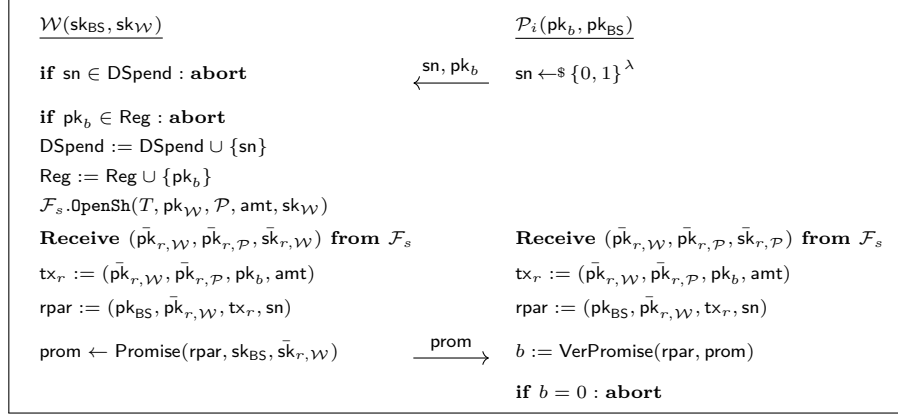




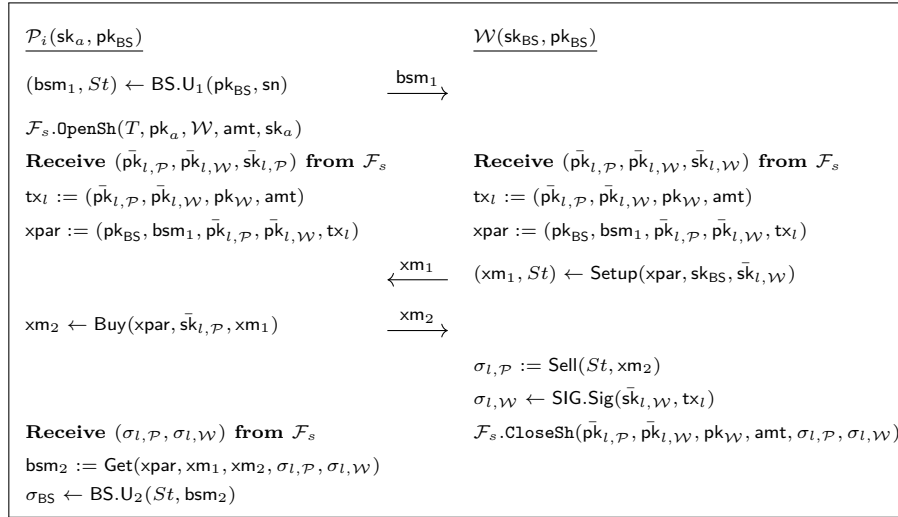
**Fig. 9.** Schematic Overview of an exchange protocol  $EXC = (Setup, Buy, Sell, Get)$  for a signature scheme  $SIG = (SIG.Gen, SIG.Sig, SIG.Ver)$  and a blind signature scheme  $BS = (BS.Gen, BS.S, BS.U, BS.Ver)$ .



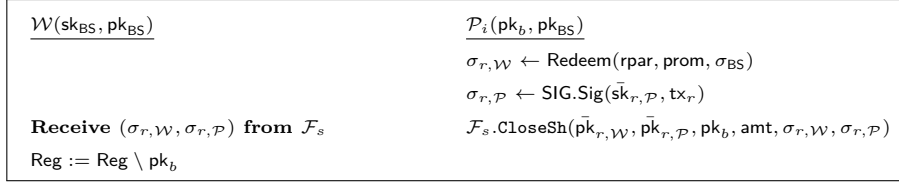
**Fig. 10.** Schematic overview of a redeem protocol  $RP = (Promise, VerPromise, Redeem)$  for a signature scheme  $SIG = (SIG.Gen, SIG.Sig, SIG.Ver)$  and a blind signature scheme  $BS = (BS.Gen, BS.S, BS.U, BS.Ver)$ .



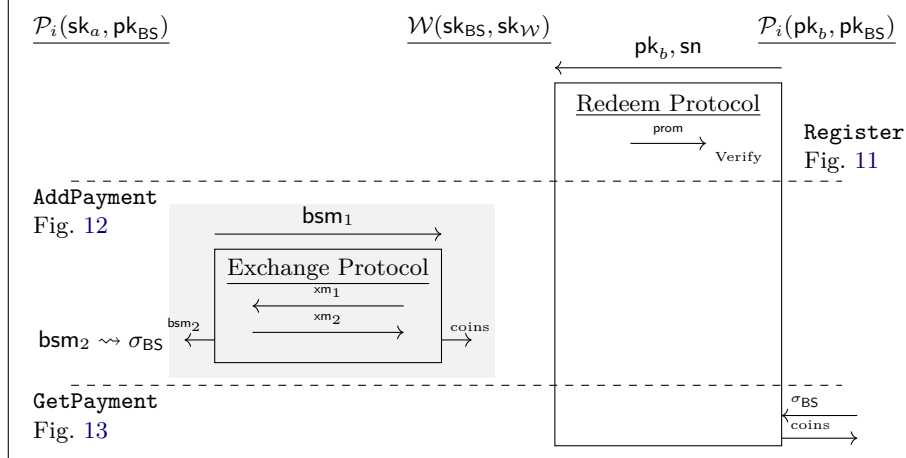
**Fig. 11.** Overview of the sub-protocol **Register** of protocol Sweep-UC. The protocol is run between the sweeper  $\mathcal{W}$  and a party  $\mathcal{P}_i$ .



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



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



**Fig. 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  $\mathcal{F}_{ux}$** 

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

**Interface Register**( $\text{pk}_b$ ), called by  $\mathcal{P}_i$ :

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

**Interface AddPayment**( $\text{pk}_a, \text{sk}_a, \text{pk}_b$ ), called by  $\mathcal{P}_i$ :

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

**Interface ChangePayment**( $\text{pk}_a, \text{pk}_b, \text{pk}_c$ ), called by  $\mathcal{S}$ :

- 01 Search for entry  $(\mathcal{P}_i, \text{pk}_a, \text{pk}_b)$  in  $\text{Pay}$ . 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  $(\mathcal{P}_i, \text{pk}_a, \text{pk}_b)$  in  $\text{Pay}$  with  $(\mathcal{P}_i, \text{pk}_a, \text{pk}_c)$ .

**Interface GetPayment**( $\text{pk}_b$ ), called by  $\mathcal{P}_i$ :

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

**Fig. 15.** Ideal functionality  $\mathcal{F}_{ux}$  that models an unlinkable exchange.

**Functionality  $\mathcal{L}^{\text{SIG}}$**

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

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

- 01 Set  $\text{bal}[\text{pk}] := c$ .
- 02 Send (“updatedFunds”,  $\text{pk}, c$ ) to every entity.

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

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

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

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

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

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

**Fig. 16.** Global ideal functionality  $\mathcal{L}^{\text{SIG}}$  that models a ledger.

**Functionality  $\mathcal{F}_s$**

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

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

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

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

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

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