A Formal Treatment of Deterministic Wallets

Poulami Das*[1] Sebastian Faust†[1] Julian Loss‡ [2]

[1] TU Darmstadt, Germany

[2] Ruhr University Bochum, Germany and University of Maryland, USA

Abstract

In cryptocurrencies such as Bitcoin or Ethereum, users control funds via secret keys. To transfer funds from one user to another, the owner of the money signs a new transaction that transfers the funds to the new recipient. This makes secret keys a highly attractive target for attacks and has led to prominent examples where millions of dollars worth in cryptocurrency were stolen. To protect against these attacks, a widely used approach are so-called hot/cold wallets. In a hot/cold wallet system, the hot wallet is permanently connected to the network, while the cold wallet stores the secret key and is kept without network connection. In this work, we propose the first comprehensive security model for hot/cold wallets and develop wallet schemes that are provably secure within these models. At the technical level, our main contribution is to provide a new, provably secure ECDSA-based hot/cold wallet scheme that can be integrated into legacy cryptocurrencies such as Bitcoin. Our scheme makes several subtle changes to the BIP32 proposal and requires a technically involved security analysis.

Keywords: Wallets; cryptocurrencies; foundations

1 Introduction

In decentralized cryptocurrencies such as Bitcoin or Ethereum, the money mechanics (e.g., who owns what and how money is transferred) are controlled by a network of miners. To this end, the miners agree via a consensus protocol about the current balance that each party has in the system. Changes to these balances are validated by the miners according to well-specified rules. In most cryptocurrencies, balance updates are executed via *transactions*. A transaction transfers money between *addresses*, which is the digital identity of a party and technically is represented by a public key of a digital signature scheme.[1] For better illustration, consider the example where Alice wants to send some of her coins – say 1 $\mathtt{BTC}$ – from her address $pk_{\mathtt{A}}$ to Bob's address $pk_{\mathtt{B}}$. To this end, she creates a transaction $\mathtt{tx}_{\mathtt{AB}}$ that informally says: "Transfer 1 $\mathtt{BTC}$ from $pk_{\mathtt{A}}$ to $pk_{\mathtt{B}}$". To ensure that only Alice can send her coins to Bob, we require that $\mathtt{tx}_{\mathtt{AB}}$ is accompanied by a valid signature of $\mathsf{H}(\mathtt{tx}_{\mathtt{AB}})$. Since only the owner of the corresponding $sk_{\mathtt{A}}$ – here Alice – can produce a valid signature, control over $sk_{\mathtt{A}}$ implies full control over the funds assigned to $pk_{\mathtt{A}}$. This makes secret keys a highly attractive target for attacks. Unsurprisingly, there are countless examples of spectacular hacks where the attacker was able to steal millions of dollars by breaking into a system and extracting the secret key [Ske18, Blo18]. According to the cryptocurrency research firm CipherTrace, in 2018 alone, attackers managed to steal more than USD 1 billion worth in cryptocurrency [Bit18].

One reason for many of these attacks is that large amounts of funds are often controlled by so-called *hot wallets*. A hot wallet is a piece of software that runs on a computer or a smart phone and has a direct connection to the Internet. This make hot wallets very convenient to use since they can move funds around easily. On the downside, however, their permanent Internet connection often makes them an easy target for attackers, e.g., by exploiting software vulnerabilities via malware or phishing. Thus, it

*Email: poulami.das@crisp-da.de

†Email: sebastian.faust@cs.tu-darmstadt.de

‡Email: julian.loss@ruhr-uni-bochum.de. Work done while author was at Ruhr-University Bochum, Germany.

[1] To be more precise, in Bitcoin funds are assigned to the hash of a public key, and not to the public key itself.

is generally recommended to store only a small amount of cryptocurrency on a hot wallet, while larger amounts of money should be transferred to a *cold wallet.* A cold wallet stays disconnected from the network most of the time and may in practice be realized by a dedicated hardware device [Wik18b], or by a paper wallet where the secret key is printed on paper and stored in a secure place.

A simple way to construct a hot/cold wallet is to generate a key pair $(pk_{\mathsf{cold}}, sk_{\mathsf{cold}})$ and store the secret key sk_{cold} on the cold wallet, while the corresponding public key pk_{cold} is kept on the hot wallet (or published over the Internet). A user can then directly transfer money to the cold wallet by publishing a transaction on the blockchain that sends money to pk_{cold}. As long as the owner of the cold wallet does not want to spend its funds, the cold wallet never needs to come online. This naive approach has one important drawback. Since all transactions targeting the cold wallet send money to the *same* public key pk_{cold}, the cold wallet may accumulate, over time, a large amount of money. Moreover, all transactions are publicly recorded on the blockchain, and thus pk_{cold} becomes an attractive target for an attack the next time the wallet goes online (which will happen at the latest when the owner of the wallet wants to spend its coins).

To mitigate this attack, it is common practice in the cryptocurrency community to use each key pair only for a single transaction. Hence, we may generate a "large number" of fresh key pairs $(sk_1, pk_1), \ldots, (sk_\ell, pk_\ell)$. Then, the ℓ public keys are sent to the hot wallet, while the corresponding ℓ secret keys sk_i are kept on the cold wallet. While this approach keeps individual transactions unlinkable, it only works for an a-priori fixed number of transactions, and requires storage on the hot/cold wallet that grows linearly with ℓ.

Fortunately, in popular cryptocurrencies such as Bitcoin, these two shortcomings can be solved by exploiting the algebraic structure of the underlying signature scheme (e.g., the ECDSA signature scheme in Bitcoin). In the cryptocurrency literature, this approach is often called *deterministic wallets* [But13] and is standardized in the BIP32 improvement proposal [Wik18a].[2] At a high level, a deterministic wallet consists of a *master secret key msk* together with a matching *master public key mpk* and a deterministic *key derivation procedure.* At setup, the master public key is given to the hot wallet, whereas the master secret key is kept on the cold wallet. After setup, the hot and cold wallet can independently generate matching session keys using the key derivation procedure and their respective master keys. Using this approach, we only need to store a single (master) key on the hot/cold wallet in order to generate an arbitrary number of (one-time) session keys.

Informally, a deterministic wallet should offer two main security guarantees. First, an *unforgeability property*, which ensures that as long as the cold wallet is not compromised, signatures to authenticate new transactions can not be forged, and thus funds are safe. Second, an *unlinkability* property, which guarantees that public keys generated from the same master public key *mpk* are computationally indistinguishable from freshly generated public keys. Despite the widespread use of deterministic wallets (e.g., they are used in most hardware wallets such as *ledger* or TREZOR, and by common software wallets such as Jaxx), only limited formal security analysis of these schemes has been provided (we will discuss the related work in Section 1.3). The main contribution of our work is to close this gap.

1.1 Deterministic hot/cold wallets

Before we outline our contribution, we recall (a slightly simplified version of) the BIP32 wallet construction as used by popular cryptocurrencies. We emphasize that for ease of presentation, we abstract from some of the technical details of the BIP32 scheme. In particular, we focus in this work on the (conceptually cleaner) deterministic wallets ignoring the "hierarchical" component of BIP32 (see [Med18] for a full specification). We leave it as an important open problem to also develop a formal model for hierarchical wallets (see Section 7 for a more detailed discussion). In the following description we focus on ECDSA-based wallets as ECDSA is the underlying signature scheme used by most popular cryptocurrencies.

Let G denote the base point of an ECDSA elliptic curve. The deterministic ECDSA wallet uses an ECDSA key tuple as its master secret/public key pair, denoted by $(msk = x, mpk = x \cdot G)$. The master secret key *msk* is stored on the cold wallet, while the corresponding master public key *mpk* is kept on the corresponding hot wallet. In addition, the hot wallet and the cold wallet both keep a common secret string *ch* which is called the "chaincode". To derive a new session public key with identifier *ID*, the hot

[2] BIP32 stands for Bitcoin improvement proposal. The same approach is also used for other cryptocurrencies such as Ethereum or Dash.

wallet computes $w \leftarrow \mathsf{H}(ch, ID), pk_{ID} \leftarrow mpk + w \cdot G$ and the cold wallet computes the corresponding session secret key as $w \leftarrow \mathsf{H}(ch, ID), sk_{ID} \leftarrow msk + w$. As argued, e.g., in [MB18], this construction satisfies both unlinkability and unforgeability as long as the chaincode and all derived secret keys remain hidden from the adversary.

Unfortunately, hot wallet breaches happen frequently, and hence the assumption that the chaincode stays secret is rather unrealistic. When ch is revealed, however, the unlinkability property is trivially broken since the adversary can derive from mpk and ch the corresponding session public key pk_{ID} for any ID of its choice. Even worse, as we discuss in Section 4.1.2 (and as already suggested in [MB18]), a hot wallet security breach may in certain cases even break the unforgeability property of the wallet scheme.

1.2 Our contributions

At the conceptual level, our main contribution is to introduce a formal comprehensive security model to analyze hot/cold wallets. On the other hand, at the technical level, we design a new ECDSA-based wallet scheme and prove its security within our model. The latter is achieved using a modular approach, which shows that signature schemes exhibiting certain rerandomizability properties for the key suffice to securely instantiate wallets in our model. Further details are provided below.

Security model for wallets. As our first contribution we provide a formal security model that precisely captures the security properties that a hot/cold wallet should satisfy. In particular, we incorporate into our model hot wallet security breaches, access to derived public keys and corresponding signatures that may appear on the blockchain. More concretely, let $\mathsf{SWal} = (\mathsf{SWal.MGen}, \mathsf{SWal.SKDer}, \mathsf{SWal.PKDer}, \mathsf{SWal.Sign}, \mathsf{SWal.Verify})$ be a wallet scheme, where $\mathsf{SWal.MGen}$ denotes the master key generation algorithm, $(\mathsf{SWal.SKDer}, \mathsf{SWal.PKDer})$ are used for deriving session keys and $(\mathsf{SWal.Sign}, \mathsf{SWal.Verify})$ represent the signing and verification algorithms of the underlying signature scheme. The security of SWal is defined via two game-based security notions that we call *wallet unlinkability* and *wallet unforgeability*.

Our notion of unlinkability can informally be described as a form of forward security – similar in spirit to key exchange models for analyzing TLS. It guarantees that all money that was sent to session public keys $pk_{ID} \leftarrow \mathsf{SWal.PKDer}(mpk, ch, ID)$ derived *prior* to the hot wallet breach, can not be linked to mpk. Notably, our unlinkability property even holds against an adversary that sees a polynomial number of session public keys generated from mpk and signatures for adversarially chosen messages. On the other hand, our unforgeability notion considers a natural threat model where funds on the cold wallet remain secure even if the hot wallet is fully compromised. While at first sight it may seem that achieving unforgeability in such a setting is straightforward, it turns out that in particular for ECDSA-based wallets, we have to deal with several technical challenges. The main reason for this is that once the hot wallet is breached, the session public keys are not fresh anymore (i.e., all session public keys are now related to the master public key mpk). This hinders a straightforward reduction to the security of the underlying signature scheme used by the cryptocurrency. Even worse, we argue that for certain naive instantiations of wallet schemes, wallet unforgeability can be broken and an adversary may steal money from the cold wallet *without* ever breaking into it.

Stateful deterministic wallets. In order to achieve our security definition of forward unlinkability, we consider the natural notion of *stateful deterministic wallets*. In a stateful wallet, the hot and cold wallet share a common secret state St that is (deterministically) updated for every new session key pair. More concretely, the master key generation algorithm $\mathsf{SWal.MGen}$ outputs (together with the master key pair (mpk, msk)) an initial state St_0 that will be stored on both the hot and the cold wallet. Then, to derive new session keys, the secret/public key derivation algorithms $\mathsf{SWal.SKDer}$ and $\mathsf{SWal.PKDer}$ take as input additionally the current state St_{i-1} and output the new state St_i, while the old state St_{i-1} is erased from the hot/cold wallet. The update mechanism for deriving the new state has to guarantee that St_i looks random even if future states St_j (for $j > i$) are revealed. Together with a mechanism for deriving new session key pairs, our scheme achieves the strong aforementioned notion of forward unlinkability. We note that while state updates (together with secure erasures) are needed to achieve our new notion of forward unlinkability, our notion of unforgeability might also be achievable by some of the currently used (stateless) wallet schemes.

Modular approach for provably secure wallets. To securely instantiate our stateful deterministic wallets, we provide a modular approach that uses digital signature schemes with rerandomizable keys. This notion – originally due to Fleischhacker et al. [FKM$^+$16] – extends standard digital signature schemes with two additional algorithms: RandSK and RandPK. These algorithms take as input a secret key sk, respectively public key pk, and some randomness ρ and output fresh keys sk', respectively pk'. Besides the standard unforgeability property, signatures with rerandomizable keys guarantee that the key pair (sk', pk') is fresh and independent of the original keys (sk, pk) from which they were generated.

Given a secure signature scheme with rerandomizable keys, we show how to generically instantiate our wallet scheme as follows. Let St be the current state of the hot/cold wallet. The public key derivation algorithm $\mathsf{SWal.PKDer}(mpk, St, ID)$ first computes $(\omega_{ID}, St') = \mathsf{H}(St, ID)$. Then, it derives the new session public key pk_{ID} by running the public key rerandomizing algorithm RandPK via $pk_{ID} \leftarrow \mathsf{RandPK}(mpk, \omega_{ID})$, and erases the old state St. Analogously, the cold wallet can compute sk_{ID} by computing ω_{ID} as above and calling $sk_{ID} \leftarrow \mathsf{RandSK}(msk, \omega_{ID})$. If H is modeled as a random oracle that maps to the randomness space for rerandomizing keys, then the rerandomizability property mentioned above satisfies that our wallet construction achieves forward unlinkability. On the other hand, wallet unforgeability follows from the unforgeability of the underlying signature scheme. For the latter to go through, we rely on the special `RSign` oracle that is provided in the unforgebaility game of signatures with rerandomizable keys (see below). Besides its strong security guarantees, our generic wallet construction preserves the storage efficiency of the BIP32 standard and only requires one hash computation more per hot/cold wallet for every derived session key pair.

Of course, before we can use our wallet scheme in practice, we need to build signatures with rerandomizable keys from standard (practical) signature schemes ideally used by cryptocurrencies. As shown in [FKM$^+$16] the Schnorr signature scheme [Sch89] satisfies these properties. In addition, we show that also BLS signatures [BLS04] can be used to construct signatures with rerandomizable keys. Thus, these schemes are natural candidates for our wallet construction.

Provably secure ECDSA-based wallets. While many cryptocurrencies plan to use Schnorr and BLS signatures in the future, to date almost all legacy cryptocurrencies (e.g., Bitcoin or Ethereum) rely on the ECDSA signature scheme. The main technical contribution of our work is thus to propose the first provably secure construction of stateful deterministic wallets that work together with ECDSA-based cryptocurrencies such as Bitcoin. To achieve this, we make several subtle changes to the current way hot/cold wallets are built in BIP32 for Bitcoin. An important goal of our construction is that all these changes come with minimal overheads to guarantee efficiency and are compatible with Bitcoin and other state-of-the-art cryptocurrencies. The latter ensures that our wallet scheme can be readily deployed as a more secure alternative for existing hot/cold wallet systems. At the technical level, the main challenge of our work lies in proving that the ECDSA signatures can be used to construct a signature scheme with rerandomizable keys. Due to the rather “contrived” nature of ECDSA signatures our analysis is, however, more involved than for Schnorr and BLS signatures, and also requires us to slightly weaken the original notion of *unforgeability under rerandomized keys* due Fleischhacker et al. [FKM$^+$16]. We call this notion *unforgeability under honestly rerandomized keys (***uf-cma-hrk***).*

Formally, we prove **uf-cma-hrk** of a “salted version” of the ECDSA signature scheme assuming that the standard ECDSA signature scheme is existentially unforgeable under chosen message attacks (**uf-cma**). The main challenge for this reduction is that in the **uf-cma-hrk** game, the adversary may see signatures under related (i.e., rerandomized) keys, where the relation between these keys may be known to the adversary. This significantly complicates the reduction. More precisely, in the reduction we need to embed the target public key pk^* of the **uf-cma** game for the ECDSA signature scheme into the simulation of the adversary in the **uf-cma-hrk** game. Once pk^* has been embedded, the reduction may have to answer signing queries for *any* of the rerandomized keys that the adversary can ask via the oracle `RSign`. Unfortunately, for this simulation we neither know the corresponding secret keys nor can the reduction answer these queries by using the underlying ECDSA signing oracle from the **uf-cma** game.

To overcome this challenge, we develop an efficient method that transfers ECDSA signatures wrt. pk^* to signatures wrt. a related public key, and show how to apply it for proving the **uf-cma-hrk** security. The later is the main technical contribution of our work.

Practical considerations. As a final contribution, we explore the practical implications of our work. First, we argue that a careless implementation of hot/cold wallets using as underlying signature scheme, e.g., Schnorr or BLS, may result into a severe security vulnerability if the hot wallet is compromised. This may seem a bit surprising as the hot wallet does not contain any secret key material. At a high level, the vulnerability exploits a “related key attack” in these signature schemes, where an adversary that knows the “relation” between two related public keys pk_{ID} and $pk_{ID'}$ can transform a signature σ_{ID}

scheme under pk_{ID} to a signature $\sigma_{ID'}$ under $pk_{ID'}$. This may have severe consequences because once an adversary sees a signature σ_{ID} that transfers funds assigned to pk_{ID}, it can also transfer the funds held by $pk_{ID'}$.

As a second practical contribution, we describe how our ECDSA-based wallet scheme can be integrated into Bitcoin. One difficulty is that for the proof to go through, we need that signatures produced by the cold wallet are salted with fresh randomness and prefixed by the pulic key (or the hash of it). Fortunately, Bitcoin supports a simple scripting language such that these changes can be integrated at very low additional costs.

1.3 Related work

Research on wallet systems. Hot/cold wallets are widely used in cryptocurrencies and various implementations on standard computing and dedicated hardware devices are available. Most related to our work is the result of Gutoski and Stebila [GS15] who discuss a flaw in BIP32 and propose a (provably secure) countermeasure against it. Concretely, they study the well known attack against deterministic wallets [But13] that allows to recover the master secret key once a single session key has leaked from the cold wallet. They then propose a fix for this flaw which allows up to d session keys to leak, and show by a counting argument that under a one-more discrete-log assumption the master secret key can not be recovered. We emphasize that their model is rather restricted and does not consider an adversary learning public keys or signatures for keys which have not been compromised. More importantly, [GS15] prove only a very weak security guarantee. Namely, instead of aiming at the standard security notion of unforgeability where the adversary's goal is to forge a signature (as considered in our work), [GS15] consider the much weaker guarantee where the adversary's goal is to extract the entire master secret key. Hence, the security analysis in [GS15] does not consider adversaries that forge a signature with respect to some session public key, while in practice this clearly violates security.

Besides [GS15], various other works explore the security of hot/cold wallets. Similar to [GS15], Fan et al. [FTS+18] study the security against secret session key leakage (they call it "privilege escalation attacks"). Unfortunately, their proposed countermeasure is ad-hoc and no formal model nor security proof is provided. Another direction is taken by Turuani et al. [TVR16] who provide an automated verification of the Bitcoin Electrum wallet in the Dolev Yao model. Since the Dolev-Yao model assumes that ciphertexts, signatures etc. are all perfect, their analysis exclude potential vulnerabilities such as related key attacks, which turn out to be very relevant in the hot/cold wallet setting.

Another line of recent work focuses on the security analysis of hardware wallets [MPas19, AGKK19]. Both works target different goals. The work of Marcedone et al. [MPas19] aims at integrating two-factor authentication into wallet schemes, while Arapinis et al. [AGKK19] consider hardware attacks against hardware wallets and provide a formal modeling of such attacks in the UC framework. Similar to the latter, Curtoius et al. [CEV14] investigate how implementation flaws such as bad and correlated randomness may affect security. Other works that study the implications of weak randomness in wallets are [BR18, BH19].

Orthogonal to our work is a large body of work on threshold ECDSA [GGN16, LN18, DKLS18] and multisignatures [BDN18] to construct more secure wallets. Both approaches aim at distributing trust by requiring that multiple key holders authenticate transactions. These techniques can be combined with our hot/cold wallet to mitigate attacks against the cold wallet.

Other related work. One of the techniques that we use in this work is that certain signature schemes support the following efficient transformation: given a signature under some public key pk, one can produce a signature with respect to a related key pk'. While for certain signature schemes such as Schnorr [Sch89] this is a well-known trick that has been used in various works [FF13, KMP16, ZCC+15], we are not aware of any prior use of such an algorithm for the ECDSA signature scheme. In addition, as discussed above we make use of the abstraction of signature schemes with rerandomizable keys that was originally introduced by Fleischhacker et al. [FKM+16] in the context of sanitizable signatures.

2 Preliminaries

Notation. We denote as $s \xleftarrow{\$} \mathcal{H}$ the uniform sampling of the variable s from the set $\mathcal{H}$. If ℓ is an integer, then $[\ell]$ is the set $\{1, \ldots, \ell\}$. We use uppercase letters A, B to denote algorithms. Unless otherwise stated, all our algorithms are probabilistic and we write $y \xleftarrow{\$} \mathsf{A}(x)$ to denote that A returns output y when run on input x. We write $y \leftarrow \mathsf{A}(x, \rho)$ to denote that A returns output y when run on input x and randomness ρ. Note that in this way, A becomes a deterministic algorithm. We use the notation $\mathsf{A}(x)$ to denote the set of all possible outputs of (probabilistic) algorithm A on input x.

We write A^{B} to denote that A has oracle access to B during its execution. For ease of notation, we generally assume that boolean variables are initialized to false, integers are set initially to 0, lists are initialized to $\emptyset$, and undefined entries of lists are initialized to $\perp$. To further simplify our definitions and notation, we assume that public parameters par have been securely generated and define the scheme or algebraic structure in context. We denote throughout the paper κ as the security parameter. For bit strings $a, b \in \{0,1\}^*$ if we write "$a = (b, \cdot)$" we check if the prefix of a is equal to b; likewise with "$a \neq (b, \cdot)$" we check if the prefix of a is different from b.

Security Games. We use standard code-based security games [BR04]. A *game* $\mathbf{G}$ is an interactive probability experiment between an *adversary* A and an (implicit) *challenger* which provides answers to oracle queries posed by A. $\mathbf{G}$ has one *main procedure* and can have any number of additional *oracle procedures* that describe how oracle queries are answered. We distinguish such oracle procedures from algorithmic ones by using a distinct font $\mathtt{Oracle}$. The output of $\mathbf{G}$ when interacting with adversary A is denoted as $\mathbf{G}^{\mathsf{A}}$. Finally, the randomness in any probability term of the form $\Pr[\mathbf{G}^{\mathsf{A}} = 1]$ is assumed to be over all the random coins in game $\mathbf{G}$.

Random Oracle Model. We model hash functions as random oracles [BR93]. The code of hash function H is defined as follows. On input x from the domain of the hash function, H checks whether $\mathsf{H}(x)$ has been previously defined. If so, it returns $\mathsf{H}(x)$. Else, it sets $\mathsf{H}(x)$ to a uniformly random element from the range of H and then returns $\mathsf{H}(x)$.

Elliptic Curve Cryptography. We denote an elliptic curve group as $\mathbb{E} = \mathbb{E}(par)$ with order p. The base point of the group $\mathbb{E}$ is denoted as $G := (x_b, y_b)$. Any point $S := (x_s, y_s)$ in the group $\mathbb{E}$ can be written as $S = aG$, where $a\mathbb{Z}_p$ and we use additive notation.

2.1 Signature Schemes

In this section, we introduce the syntax and relevant security notions for signature schemes.

Definition 2.1 (Signature Scheme). A *signature scheme* Sig is a triple of algorithms $\mathsf{Sig} = (\mathsf{Sig.Gen}, \mathsf{Sig.Sign}, \mathsf{Sig.Verify})$. The randomized *key generation algorithm* $\mathsf{Sig.Gen}$ takes as input public parameters par and returns a pair (sk, pk), of secret and public keys. The randomized *signing algorithm* $\mathsf{Sig.Sign}$ takes as input a secret key sk and a message m and returns a signature σ. The deterministic *verification algorithm* $\mathsf{Sig.Verify}$ takes as input a public key pk, a signature σ, and a message m. It returns 1 (accept) or 0 (reject). We require *correctness*: For all $(sk, pk) \in \mathsf{Sig.Gen}(par)$, and all $m \in \{0,1\}^*$, we have that

$$\Pr_{\sigma \xleftarrow{\$} \mathsf{Sig.Sign}(sk, m)} [\mathsf{Sig.Verify}(pk, \sigma, m) = 1] = 1.$$

We also adopt the notion of signature schemes with rerandomizable keys from Fleischhacker et al. [FKM+16].

Definition 2.2 (Signature Scheme with Perfectly Rerandomizable Keys). A *signature scheme with perfectly rerandomizable keys* is a tuple of algorithms $\mathsf{RSig} = (\mathsf{RSig.Gen}, \mathsf{RSig.Sign}, \mathsf{RSig.Verify}, \mathsf{RSig.RandSK}, \mathsf{RSig.RandPK})$ where $(\mathsf{RSig.Gen}, \mathsf{RSig.Sign}, \mathsf{RSig.Verify})$ are the standard algorithms of a signature scheme as defined above. Moreover, we assume that the public parameters par define a randomness space $\chi := \chi(par)$. The probabilistic *secret key rerandomization algorithm* $\mathsf{RSig.RandSK}$ takes as input a secret key sk and randomness $\rho \in \chi$ and outputs a rerandomized secret key sk'. The probabilistic *public key rerandomization algorithm* $\mathsf{RSig.RandPK}$ takes as input a public key pk and randomness $\rho \in \chi$ and outputs a rerandomized public key pk'. We make the convention that for the empty string ϵ, we have that $\mathsf{RSig.RandPK}(pk, \epsilon) = pk$ and $\mathsf{RSig.RandSK}(sk, \epsilon) = sk$. We further require:

1. *(Perfect) rerandomizability of keys:* For all $(sk, pk) \in \mathsf{RSig.Gen}(par)$ and $\rho \xleftarrow{\$} \chi$, the distributions

```
main $\textbf{uf-cma}_{\mathsf{Sig}}$
$(sk, pk) \xleftarrow{\$} \mathsf{Sig.Gen}(par)$
$(m^*, \sigma^*) \xleftarrow{\$} \mathsf{C}^{\texttt{SignO}}(pk)$
If $m^* \in Sigs$: $\mathsf{bad} \leftarrow \mathsf{true}$
$b' \leftarrow \mathsf{Sig.Verify}(m^*, pk^*, \sigma^*)$
Return $b' \wedge \neg\mathsf{bad}$

Oracle $\texttt{SignO}(m)$
$\sigma \xleftarrow{\$} \mathsf{Sig.Sign}(sk, m)$
$Sigs \leftarrow Sigs \cup \{m\}$
Return $\sigma$
```

Figure 1: Security game $\textbf{uf-cma}_{\mathsf{Sig}}$ with adversary C.

```
main $\textbf{uf-cma-hrk}_{\mathsf{RSig}}$
$\mathsf{RList} \leftarrow \{\epsilon\}$
$(sk, pk) \xleftarrow{\$} \mathsf{RSig.Gen}(par)$
$(m^*, \sigma^*, \rho^*) \xleftarrow{\$} \mathsf{C}^{\texttt{Rand},\texttt{RSign}}(pk)$
If $m^* \in Sigs$: $\mathsf{bad} \leftarrow \mathsf{true}$
If $\rho^* \notin \mathsf{RList}$: $\mathsf{bad} \leftarrow \mathsf{true}$
$pk^* \leftarrow \mathsf{RSig.RandPK}(pk, \rho^*)$
$b \leftarrow \mathsf{RSig.Verify}(pk^*, \sigma^*, m^*)$
Return $b \wedge \neg\mathsf{bad}$

Oracle $\texttt{RSign}(m, \rho)$
If $\rho \notin \mathsf{RList}$: Return $\perp$
$sk' \leftarrow \mathsf{RSig.RandSK}(sk, \rho)$
$\sigma \xleftarrow{\$} \mathsf{RSig.Sign}(m, sk')$
$Sigs \leftarrow Sigs \cup \{m\}$
Return $\sigma$

Oracle $\texttt{Rand}$
$\rho \xleftarrow{\$} \chi$
$\mathsf{RList} \leftarrow \mathsf{RList} \cup \{\rho\}$
Return $\rho$
```

Figure 2: Security game $\textbf{uf-cma-hrk}_{\mathsf{RSig}}$ with adversary C.

of (sk', pk') and (sk'', pk'') are identical, where:

$$(sk', pk') \leftarrow (\mathsf{RSig.RandPK}(pk, \rho), \mathsf{RSig.RandSK}(sk, \rho)),$$
$$(sk'', pk'') \xleftarrow{\$} \mathsf{RSig.Gen}(par).$$

2. *Correctness under rerandomized keys:* For all $(sk, pk) \in \mathsf{RSig.Gen}(par)$, for all $\rho \in \chi$, and for all $m \in \{0,1\}^*$, the rerandomized keys $sk' \leftarrow \mathsf{RSig.RandSK}(sk, \rho)$ and $pk' \leftarrow \mathsf{RSig.RandSK}(pk, \rho)$ satisfy:
$$\Pr_{\sigma \xleftarrow{\$} \mathsf{RSig.Sign}(sk', m)} [\mathsf{RSig.Verify}(pk', \sigma, m) = 1] = 1.$$

Security of Signature Schemes. In this work we will use the standard security notion of *existential unfogeability under chosen message attacks (UFCMA)*. We formalize this notion for a signature scheme Sig via the game $\textbf{uf-cma}_{\mathsf{Sig}}$ (Figure 1). In this game, the challenger begins by sampling (sk, pk) as $(sk, pk) \xleftarrow{\$} \mathsf{Gen}(par)$. The adversary is then given the public key pk and can adaptively sign messages of its choice under the corresponding secret key via an oracle $\texttt{SignO}$. Its goal is to forge a signature on a *fresh* message m^*, i.e., one that was not previously queried to $\texttt{SignO}$. For an algorithm C, we define C's advantage in game $\textbf{uf-cma}_{\mathsf{Sig}}$ as $\mathsf{Adv}^{\mathsf{C}}_{\textbf{uf-cma},\mathsf{Sig}} = \Pr\left[\textbf{uf-cma}^{\mathsf{C}}_{\mathsf{Sig}} = 1\right]$.

For a signature scheme with rerandomizable keys RSig, we also introduce a new security notion called *unforgeability under honestly rerandomized keys* that is formalized via game $\textbf{uf-cma-hrk}_{\mathsf{RSig}}$ (Figure 2). This notion constitutes a weaker form of the notion of *existential unforgeability under rerandomized keys* proposed in [FKM+16]. In the latter notion, the adversary is able to query the signing oracle not only for signatures corresponding to the public key pk that it obtains in the unforgeability experiment, but also for signatures that correspond to *arbitrary rerandomizations of pk*. Similarly, the winning condition is also relaxed in this notion by allowing the adversary to return a forgery under an (arbitrarily) rerandomized key (but still on a fresh message m^*). The main difference between the security notion from [FKM+16] and our new one is that the adversary is restricted to *honest* rerandomizations of pk, i.e., randomizations where the randomness is chosen by the challenger uniformly at random from χ. We model this via an additional oracle in the security game. For an algorithm C, we define C's advantage in game $\textbf{uf-cma-hrk}_{\mathsf{RSig}}$ as $\mathsf{Adv}^{\mathsf{C}}_{\textbf{uf-cma-hrk},\mathsf{RSig}} = \Pr\left[\textbf{uf-cma-hrk}^{\mathsf{C}}_{\mathsf{RSig}} = 1\right]$.

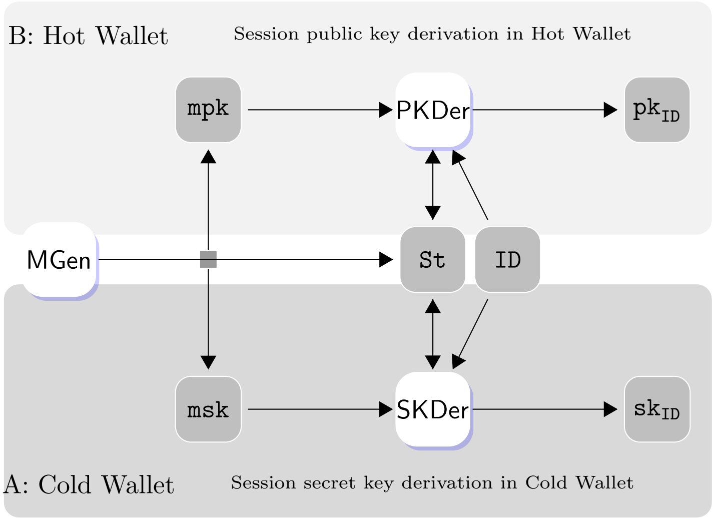

B: Hot Wallet
Session public key derivation in Hot Wallet
mpk
PKDer
pk_ID
MGen
St
ID
msk
SKDer
sk_ID
A: Cold Wallet
Session secret key derivation in Cold Wallet

Figure 3: Both Hot/ Cold wallet internally stores the common state St. The master keys are stored in the respective wallets. When a session secret key is generated within the cold wallet as $(sk_{ID}, St) \leftarrow \mathsf{SWal.SKDer}(msk, ID, St)$, the state St gets refreshed. The session public key pk_{ID} is generated within the hot wallet as $(pk_{ID}, St) \leftarrow \mathsf{SWal.PKDer}(mpk, ID, St)$, and the corresponding state St is refreshed in the same manner.

3 The Stateful Model for Wallets

In this section, we introduce our formal security model for stateful deterministic wallets. At a high level, a stateful deterministic wallet scheme allows two parties A (the cold wallet) and B (the hot wallet) to derive matching session key pairs (for signing/verification) from a pair of master keys. As presented in Figure 3, A keeps her master secret key msk and gives the master public key mpk to B. A and B can now use the key derivation procedures SKDer and PKDer, respectively, to derive an arbitrary number of session key pairs, locally, i.e., without further interaction. Intuitively, this is possible since every part of the key derivation procedure is deterministic and therefore, both A and B "automatically" carry out the same sequence of derivations.

In contrast to standard hot/cold wallets, we will make one conceptual change and add to our schemes a *state*, denoted St below. The state St is updated (deterministically) during each call to one of the key derivation procedures. As we will explain shortly, this allows to obtain a strong form of forward privacy, which we will refer to as *unlinkability*. For A to easily identify keys on the blockchain for which she can derive a corresponding secret key and to keep track of the order they where derived in by B, session keys also have an identifier $ID \in \{0,1\}^*$ which is used as an argument for the key derivation procedures. We now proceed to give the syntax of a stateful wallet scheme.

Definition 3.1 (Stateful Wallet). A *stateful wallet* is a tuple of algorithms

$$\mathsf{SWal} = (\mathsf{SWal.MGen}, \mathsf{SWal.SKDer}, \mathsf{SWal.PKDer}, \mathsf{SWal.Sign}, \mathsf{SWal.Verify}),$$

which are defined as follows. The randomized *master key generation algorithm* $\mathsf{SWal.MGen}(par)$ takes public parameters par as input and outputs a tuple $(St_0,\ mpk,\ msk)$ consisting of an *initial state* St_0, a *master secret key* msk and a *master public key* mpk. The deterministic *secret key derivation algorithm* SWal.SKDer takes as input a master secret key msk, an identity ID, and a state St. It outputs a session secret key sk_{ID} and an updated state St'. The deterministic *public key derivation algorithm* SWal.PKDer takes as input a master public key mpk, an identity ID, and a state St. It outputs a session public key pk_{ID} and an updated state St'. The randomized signing algorithm SWal.Sign takes as input a (session) secret key sk and a message m and returns a signature σ. The deterministic verification algorithm SWal.Verify takes as input a (session) public key pk, a signature σ, and a message m. It returns 1 (accept) or 0 (reject).

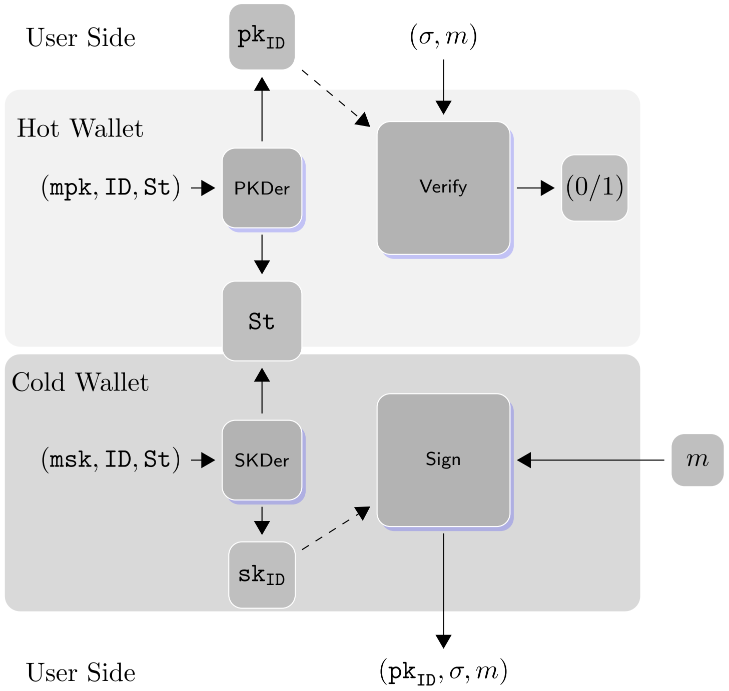


Figure 4: (1) The cold wallet signs a message m with its session secret key sk_{ID} as $\sigma \leftarrow \mathsf{SWal.Sign}(sk_{ID}, m)$. (2) Anyone can later verify the validity of a signature σ on message m as $(0/1) \leftarrow \mathsf{SWal.Verify}(pk_{ID}, \sigma, m)$.

Let $\mathsf{SWal} = (\mathsf{SWal.MGen}, \mathsf{SWal.SKDer}, \mathsf{SWal.PKDer}, \mathsf{SWal.Sign}, \mathsf{SWal.Verify})$ denote a stateful wallet according to Definition 3.1, for the remainder of this section. We now define correctness of stateful deterministic wallets. Roughly speaking, correctness should ensure that if the cold wallet A and the hot wallet B derive session key pairs on the same set of identities $ID_0, ..., ID_{n-1} \in \{0,1\}^*$ and in the same order, any signature created under one of the resulting signing keys of A should correctly verify under the corresponding verification key of B. In other words, all the derived session keys should "match".

Definition 3.2 (Correctness of Stateful Wallets). For all $(St_0, msk, mpk) \in \mathsf{SWal.MGen}(par)$, all $n \in \mathbb{N}$, all $\vec{ID} := (ID_0, ..., ID_{n-1}) \in \{0,1\}^*$, we set $St_0[\vec{ID}, St_0, msk] = St_0[\vec{ID}, St_0, mpk] := St_0$ and define the sequence $\left\{\left(sk_i[\vec{ID}, St_0, msk], St_i[\vec{ID}, St_0, msk]\right)\right\}_{1 \le i \le n}$ recursively as

$$\left(sk_i[\vec{ID}, St_0, msk], St_i[\vec{ID}, St_0, msk]\right) := \mathsf{SWal.SKDer}(msk, ID_{i-1}, St_{i-1}[\vec{ID}, St_0, msk]).$$

Analogously, we define $\left\{\left(pk_i[\vec{ID}, St_0, mpk], St_i[\vec{ID}, St_0, mpk]\right)\right\}_{1 \le i \le n}$ recursively as

$$\left(pk_i[\vec{ID}, St_0, mpk], St_i[\vec{ID}, St_0, mpk]\right) := \mathsf{SWal.PKDer}(mpk, ID_{i-1}, St_{i-1}[\vec{ID}, St_0, mpk]).$$

We say that SWal is *correct* if for all $n \in \mathbb{N}$, all $(ID_0, ..., ID_{n-1}) \in \{0,1\}^*$, all $(St_0, msk, mpk) \in \mathsf{SWal.MGen}(par)$, and all $m \in \{0,1\}^*$, we have for $pk := pk_n[\vec{ID}, St_0, mpk]$ and $sk := sk_n[\vec{ID}, St_0, msk]$:

$$\Pr_{\sigma \xleftarrow{\$} \mathsf{SWal.Sign}(sk, m)}[\mathsf{SWal.Verify}(pk, \sigma, m) = 1] = 1.$$

In the next subsection, we introduce the two basic security notions for stateful wallets, namely a) *unlinkability* of generated public session keys, and b) *unforgeability* of corresponding signatures.

3.1 Wallet Unlinkability

We begin by introducing the notion of *wallet unlinkability*. Intuitively, unlinkabililty guarantees that transactions sending money to different public session keys that were derived from the *same master key* should be unlinkable. Formally, we require that, given the master public key, the distribution of session public keys is computationally indistinguishable from session public keys that are generated from a fresh (i.e., independently and randomly chosen) master public key and state. Unfortunately, there is little hope to achieve this guarantee for keys to which the adversary knows the state St used to derive them. Therefore, our notion of unlinkability satisfies a weaker form of privacy called *forward unlinkability*. This means that keys generated prior to a hot wallet breach (i.e., when the adversary learns the state) cannot be linked to mpk.

The wallet unlinkability game $\mathbf{unl}_{\mathsf{SWal}}$ is presented in Figure 5. Initially, A receives as input a master public key mpk generated via $\mathsf{MGen}(\mathit{par})$ and subsequently interacts with oracles PK, `WalSign` and Chall that reflect A's capabilities. The game internally maintains a state St, which is updated when A calls the oracle PK to derive new keys. In addition, at any point in time A can read out the current state St by calling the oracle `getSt`. This models A's capability to break into the hot wallet on which the state is stored. Finally, the oracle Chall allows A to obtain a challenge public key $\mathit{pk}_{\mathit{ID}}$ for a user identity ID of its choice. This challenge public key is either "real" or "random", i.e., it depends on mpk or was sampled freshly and independently of mpk (see below for details). A's goal is to distinguish these two scenarios. However, A is only considered successful if it obtains St (via oracle `getSt`) *after* being given the challenge public key $\mathit{pk}_{\mathit{ID}}$.[3] We now proceed in explaining the oracles to which A has access in more detail.

$\mathsf{PK}(\mathit{ID})$**:** The oracle PK takes as input an ID and returns a corresponding session public key $\mathit{pk}_{\mathit{ID}}$. It models A's capability to observe transactions stored on the blockchain that transfer money to $\mathit{pk}_{\mathit{ID}}$. A typical setting where this may occur is when funds are sent via the blockchain to the cold wallet. For simplicity of bookkeeping, we make the convention that identifiers are unique and thus A can call PK only once per ID.

`WalSign`(m, ID)**:** The oracle `WalSign` takes as input an identity ID and a message m and returns the corresponding signature if $\mathit{pk}_{\mathit{ID}}$ has been previously returned as a result to a $\mathsf{PK}(\mathit{ID})$ query. As such, it allows A to sign, in an adaptive fashion, messages of its choice under public keys that it previously obtained via the oracle PK. `WalSign` models that an adversary A may obtain signatures that are produced by the cold wallet with $\mathit{sk}_{\mathit{ID}}$, when funds are spent from the cold wallet (e.g., when the owner of the cold wallet buys something with the collected coins).

`getSt`**:** The oracle `getSt` returns the current state St and records this event by setting $\mathit{StateQuery}$ to true. As mentioned above, this models hot wallet corruption.

$\mathsf{Chall}(\mathit{ID})$**:** The oracle Chall takes as input an ID and returns a public key $\mathit{pk}^b_{\mathit{ID}}$ that depends on the uniformly random bit b sampled internally by the game $\mathbf{unl}_{\mathsf{SWal}}$. Chall can be called only a single time. If $b = 0$, $\mathit{pk}^0_{\mathit{ID}}$ is derived from the current state St and mpk as $(\mathit{pk}^0_{\mathit{ID}}, \cdot) \leftarrow \mathsf{SWal.PKDer}(\mathit{mpk}, \mathit{ID}, \mathit{St})$. If $b = 1$, $\mathit{pk}^1_{\mathit{ID}}$ is derived from a freshly generated master public key and state for the same identity ID, i.e., via the sequence of steps:

- $(\hat{\mathit{St}}, \cdot, \hat{\mathit{mpk}}) \xleftarrow{\$} \mathsf{SWal.MGen}(\mathit{par})$
- $(\mathit{pk}^1_{\mathit{ID}}, \cdot) \leftarrow \mathsf{SWal.PKDer}(\hat{\mathit{mpk}}, \mathit{ID}, \hat{\mathit{St}})$

If A sets $\mathit{StateQuery}$ prior to calling Chall, or queries Chall on an identity ID that it previously queried PK on, Chall always returns $\perp$ in order to prevent a trivial attack on unlinkability. We define A's advantage in $\mathbf{unl}_{\mathsf{SWal}}$ as

$$\mathsf{Adv}^{\mathsf{A}}_{\mathbf{unl},\mathsf{SWal}} = \left| \Pr\left[\mathbf{unl}^{\mathsf{A}}_{\mathsf{SWal}} = 1\right] - \frac{1}{2} \right| . \tag{1}$$

[3] Recall that otherwise the adversary can trivially distinguish between "real" or "random".

```
main $\mathbf{unl}_{\mathsf{SWal}}$
$(St, msk, mpk) \xleftarrow{\$} \mathsf{SWal.MGen}(par)$
$b \xleftarrow{\$} \{0,1\}$
$\mathtt{Orc} \leftarrow \{\mathsf{PK}, \mathtt{WalSign}, \mathsf{Chall}, \mathtt{getSt}\}$
$b' \xleftarrow{\$} \mathsf{A}^{\mathtt{Orc}}(mpk)$
Return $b' = b$

Oracle $\mathtt{WalSign}(m, ID)$
If $SSNKeys[ID] = \bot$: Return $\bot$
$(pk_{ID}, sk_{ID}) \leftarrow SSNKeys[ID]$
$\sigma \xleftarrow{\$} \mathsf{SWal.Sign}(sk_{ID}, m)$
Return $\sigma$

Oracle $\mathtt{PK}\,(ID)$ // Once per ID
$tmp_1 \leftarrow (msk, ID, St)$
$tmp_2 \leftarrow (mpk, ID, St)$
$(sk_{ID}, St) \leftarrow \mathsf{SWal.SKDer}(tmp_1)$
$(pk_{ID}, St) \leftarrow \mathsf{SWal.PKDer}(tmp_2)$
$SSNKeys[ID] \leftarrow (pk_{ID}, sk_{ID})$
Return $pk_{ID}$

Oracle $\mathtt{getSt}$
$StateQuery \leftarrow \mathsf{true}$
Return $St$

Oracle $\mathsf{Chall}\,(ID)$ //One time
If $StateQuery$: Return $\bot$
If $SSNKeys[ID] \neq \bot$: Return $\bot$
// Generate real key
$tmp_1 \leftarrow (msk, ID, St)$
$tmp_2 \leftarrow (mpk, ID, St)$
$(sk^0_{ID}, St) \leftarrow \mathsf{SWal.SKDer}(tmp_1)$
$(pk^0_{ID}, St) \leftarrow \mathsf{SWal.PKDer}(tmp_2)$
// Generate random key
$(\hat{St}, \hat{msk}, \hat{mpk}) \xleftarrow{\$} \mathsf{SWal.MGen}(par)$
$tmp_1 \leftarrow (\hat{msk}, ID, \hat{St})$
$tmp_2 \leftarrow (\hat{mpk}, ID, \hat{St})$
$(sk^1_{ID}, \cdot) \leftarrow \mathsf{SWal.SKDer}(tmp_1)$
$(pk^1_{ID}, \cdot) \leftarrow \mathsf{SWal.PKDer}(tmp_2)$
$SSNKeys[ID] \leftarrow (pk^b_{ID}, sk^b_{ID})$
Return $pk^b_{ID}$
```

Figure 5: Adversary A playing in Game $\mathbf{unl}_{\mathsf{SWal}}$.

```
main $\mathbf{wunf}_{\mathsf{SWal}}$
$(St, msk, mpk) \xleftarrow{\$} \mathsf{SWal.MGen}(par)$
$(m^*, \sigma^*, ID^*) \xleftarrow{\$} \mathsf{A}^{\mathsf{PK},\mathtt{WalSign}}(mpk, St)$
If $SSNKeys[ID^*] = \bot$
    Return 0
$(pk_{ID^*}, sk_{ID^*}) \leftarrow SSNKeys[ID^*]$
If $m^* \in Sigs[ID^*]$
    Return 0
If $\mathsf{SWal.Verify}(pk_{ID^*}, \sigma^*, m^*) = 0$
    Return 0
Return 1

Oracle $\mathtt{WalSign}(m, ID)$
If $SSNKeys[ID] = \bot$: Return $\bot$
$(pk_{ID}, sk_{ID}) \leftarrow SSNKeys[ID]$
$\sigma \xleftarrow{\$} \mathsf{SWal.Sign}(sk_{ID}, m)$
$Sigs[ID] \leftarrow Sigs[ID] \cup \{m\}$
Return $\sigma$

Oracle $\mathtt{PK}\,(ID)$ // Once per ID
$tmp_1 \leftarrow (msk, ID, St)$
$tmp_2 \leftarrow (mpk, ID, St)$
$(sk_{ID}, St) \leftarrow \mathsf{SWal.SKDer}(tmp_1)$
$(pk_{ID}, St) \leftarrow \mathsf{SWal.PKDer}(tmp_2)$
$SSNKeys[ID] \leftarrow (pk_{ID}, sk_{ID})$
$Sigs[ID] \leftarrow \emptyset$
Return $pk_{ID}$
```

Figure 6: Adversary A playing in Game $\mathbf{wunf}$.

3.2 Wallet Unforgeability

In this subsection we describe the *wallet unforgeability* notion. In Game $\mathbf{wunf}^{\mathsf{A}}_{\mathsf{SWal}}$ (depicted in Figure 6) we consider again an adversary A that receives as input a master public key mpk and has subsequently access to oracles PK and $\mathtt{WalSign}$, which work as their corresponding oracles in the unlinkability game. In addition, A gets as input the *initial state St*. A wins if it can produce a triple (m^*, σ^*, ID^*) such that σ^* is a valid forgery on message m^* under a public key pk_{ID^*} previously obtained from a call to PK. Here, valid means that no signature on message m^* under pk_{ID^*} was previously obtained from a call to $\mathtt{WalSign}$. We denote A's advantage in $\mathbf{wunf}_{\mathsf{SWal}}$ as

$$\mathsf{Adv}^{\mathsf{A}}_{\mathbf{wunf},\mathsf{SWal}} = \Pr\Big[\mathbf{wunf}^{\mathsf{A}}_{\mathsf{SWal}} = 1\Big]. \tag{2}$$

Unforgeability for Keys with Compromised State. At a high-level, the $\mathbf{wunf}_{\mathsf{SWal}}$ game models that once funds are transferred to the cold wallet they remain secure even if (a) the hot wallet is compromised, and (b) the adversary can see transfers of coins sent from the cold wallet. We now explain the game in more detail. In contrast to the $\mathbf{unl}_{\mathsf{SWal}}$ game from the previous section, in the $\mathbf{wunf}_{\mathsf{SWal}}$ game the adversary is given the *state St* as part of its initial input. This models the "worst-case" adversary that breaks into the hot wallet right after the hot/cold wallet has been initialized. In addition, to giving A the initial state St and the master public key mpk, we also give it access to the PK and $\mathtt{WalSign}$ oracle. The first can be queried by the adversary on identity ID to derive a new key pair (pk_{ID}, sk_{ID}) from the master keys and the current state, and is used mainly for bookkeeping purposes.[4] The second oracle $\mathtt{WalSign}$ is

[4] Notice that in $\mathbf{wunf}_{\mathsf{SWal}}$ the adversary obtains mpk and the initial state, and hence can compute the output pk of PK himself.

as in the $\mathbf{unl}_{\mathsf{SWal}}$ game except that we also keep track of the messages that were already signed via the map $\mathit{Sigs}[ID]$.

As already mentioned above, since the adversary receives mpk and the initial state St in the $\mathbf{wunf}_{\mathsf{SWal}}$ game, it can derive all possible pk_{ID} (even without calling $\mathtt{PK}(ID)$). This subtle difference significantly complicates the security proof in the subsequent sections and is a crucial aspect of our unforgeability notion. More concretely, since A knows the state throughout the entire game $\mathbf{wunf}_{\mathsf{SWal}}$, it may be able to mount a related key attack (RKA) against the underlying signature scheme used in our wallet construction. At a high-level the RKA allows the adversary to "transfer" a signature σ_{ID} with respect to pk_{ID} to a valid signature σ_{ID^*} for pk_{ID^*}. Signature schemes that are susceptible to such an RKA are for instance the Schnorr or BLS signature scheme, and we will discuss how to attack a hot/cold wallet instantiated in a naive way with these schemes in the appendix. Let us briefly explain how an adversary in the $\mathbf{wunf}_{\mathsf{SWal}}$ game can exploit such an RKA to break the underlying wallet scheme.

To this end, consider an adversary A that breaks into the hot wallet and obtains mpk and St. This break-in is modeled in $\mathbf{wunf}_{\mathsf{SWal}}$ by giving the adversary mpk, St at the beginning of the game. Next, the adversary waits until some funds are transferred to the cold wallet, which we model by calls to the PK oracle. Finally, A queries the $\mathtt{WalSign}$ oracle to transfer some fraction of funds – say the funds stored under pk_{ID} – from the cold wallet to some new address. In practice, this may happen for example when some of the funds kept on the cold wallet are spent for a purchase. Once the adversary has received a single signature σ_{ID} produced by the cold wallet, it can apply the RKA to steal *all funds* that have ever been transferred to the cold wallet. More precisely, given the signature σ_{ID}, the master public key mpk, and the state St it can produce valid signatures σ_{ID^*} for pk_{ID^*} where pk_{ID^*} resulted from a previous call to PK on input ID^*.

This attack results into a severe security breach as the owner of the cold wallet can loose its entire funds stored on the cold wallet. Since the attack does not require to break into the cold wallet, it strongly violates the original purpose of the hot/cold wallet concept in cryptocurrencies. Indeed, a user that transfers its funds to the cold wallet would assume that once the funds are transferred to the cold wallet, they are safe except for a break-in to the cold wallet.

As demonstrated in the subsequent section, an easy way to thwart this attack is to use *public key prefixing*, i.e., to compute a signature on m as $\mathsf{Sign}(sk_,(pk, m))$. Interestingly, this technique was also used in [MSM+15], with the purpose of preventing an RKA. This further highlights the close relation between resistance to RKAs and unforgeability in our model.

Of course, exploiting an RKA is only one possibility of stealing funds from the cold wallet, and there may be other types of attacks allowing the adversary to forge signatures with respect to keys stored on the cold wallet, given that it knows the state. Nevertheless, it also clearly underlines the importance of a formal security analysis of hot/cold wallet schemes within a strong security model. In the next section, we show how to reduce the security of a wallet scheme in the above unforgeability game to the security of the signature scheme that underlies the wallet construction.

4 Generic Constructions

In this section, we show how to realize a stateful wallet from any signature scheme with uniquely rerandomizable keys. Such a signature scheme is defined as follows.

Definition 4.1 (Signature scheme with uniquely rerandomizable keys). A rerandomizable signature scheme RSig, is said to have *uniquely rerandomizable public keys* if for all $(\rho, \rho') \in \chi$, we have that $\mathsf{RandPK}(pk, \rho) = \mathsf{RandPK}(pk, \rho')$ implies $\rho = \rho'$.

We begin by explaining our generic construction. We then prove its security with respect to the security notions introduced in Section 3. We assume in the following a signature scheme with uniquely rerandomizable keys $\mathsf{RSig} = (\mathsf{RSig.Gen}, \mathsf{RSig.Sign}, \mathsf{RSig.Verify}, \mathsf{RSig.RandSK}, \mathsf{RSig.RandPK})$. Our construction $\mathsf{swal}[\mathsf{H}]$ of a stateful wallet which internally uses the hash function $\mathsf{H}\colon \{0,1\}^* \to \mathbb{Z}_p \times \{0,1\}^\kappa$ (for state updates) is depicted in Figure 7.

4.1 Security Analysis

We proceed to analyze the properties of *unlinkability* and *unforgeability* of our stateful wallet construction (c.f. Figure 7) below.

```
Algorithm $\mathsf{SWal[H].MGen}(par)$
$St \xleftarrow{\$} \{0,1\}^\kappa$
$(mpk, msk) \xleftarrow{\$} \mathsf{RSig.Gen}(par)$
Return $(St, msk, mpk)$

Algorithm $\mathsf{SWal[H].Sign}(m, sk, pk)$
$\hat{m} \leftarrow (pk, m)$
$\sigma \xleftarrow{\$} \mathsf{RSig.Sign}(sk, \hat{m})$
Return $\sigma$

Algorithm $\mathsf{SWal[H].Verify}(pk, \sigma, m)$
$\hat{m} \leftarrow (pk, m)$
Return $\mathsf{RSig.Verify}(pk, \sigma, \hat{m})$

Algorithm $\mathsf{SWal[H].SKDer}(msk, ID, St)$
$(\omega_{ID}, St) \leftarrow \mathsf{H}(St, ID)$
$sk_{ID} \xleftarrow{\$} \mathsf{RSig.RandSK}(msk, \omega_{ID})$
Return $(sk_{ID}, St)$

Algorithm $\mathsf{SWal[H].PKDer}(mpk, ID, St)$
$(\omega_{ID}, St) \leftarrow \mathsf{H}(St, ID)$
$pk_{ID} \leftarrow \mathsf{RSig.RandPK}(mpk, \omega_{ID})$
Return $(pk_{ID}, St)$
```

Figure 7: Construction of $\mathsf{swal[H]}$ from RSig and H.

4.1.1 Unlinkability

We begin by proving unlinkability of our generic construction. The proof is rather simple and follows from collision resistance of H and that H is modeled as a random oracle. It also relies on the rerandomizability property of the underlying signature scheme.

Theorem 4.2 *Let* $\mathsf{swal[H]}$ *be the construction defined in Figure 7. Then for any adversary* A *playing in game* $\mathbf{unl}_{\mathsf{swal[H]}}$*, we have*

$$\mathsf{Adv}^{\mathsf{A}}_{\mathbf{unl},\mathsf{swal[H]}} \leq \frac{q_H(q_P+2)}{2^\kappa},$$

where q_H *and* q_P *are the number of random oracle queries and queries to oracle* PK*, respectively, that* A *makes.*

Proof. Consider an adversary A playing in game $\mathbf{unl}_{\mathsf{swal[H]}}$. A interacts with oracles PK, $\mathtt{WalSign}$, $\mathtt{getSt}$, Chall, and the random oracle H. We can assume without loss of generality that A always calls $\mathsf{Chall}\,(ID)$ *before* calling $\mathtt{getSt}$ and exclusively on an identity ID that was never previously queried to PK; otherwise, $\mathsf{Adv}^{\mathsf{A}}_{\mathbf{unl},\mathsf{swal[H]}} = 0$ and the theorem holds trivially. In the following, let $\mathcal{S}$ denote the set of values taken by the variables $St, \hat{St}$ *before* A calls $\mathsf{Chall}\,(ID)$. Furthermore, let pk^0_{ID}, pk^1_{ID} denote the keys internally sampled in $\mathbf{unl}_{\mathsf{swal[H]}}$ upon A's call $\mathsf{Chall}\,(ID)$. Note that by definition of $\mathsf{swal[H].PKDer}$, unless A manages to make a query of the form $\mathsf{H}\left(St', ID\right)$ where $St' \in \mathcal{S}$, pk^0_{ID} and pk^1_{ID} are identically distributed from its point of view. The reason is that as long as such a query hasn't been made, the values of $St, \hat{St}$ used to derive pk^0_{ID} and pk^1_{ID}, respectively, are uniformly distributed from A's point of view. Now, the rerandomizability property of RSig ensures that both pk^0_{ID} and pk^1_{ID} are identically distributed to a freshly generated public key $pk \xleftarrow{\$} \mathsf{RSig}(par)$ (and therefore identically distributed to each other). In this case we again have that $\mathsf{Adv}^{\mathsf{A}}_{\mathbf{unl},\mathsf{swal[H]}} = 0$. It therefore remains to argue that A makes such a call to H with probability at most $\left(q_H(q_P+2)\right)/2^\kappa$. This can be seen as follows. Since A makes at most q_P queries to PK throughout $\mathbf{unl}_{\mathsf{swal[H]}}$, in particular $|\mathcal{S}| \leq q_P + 2$. Since we have assumed that A always calls $\mathtt{getSt}$ *after* calling Chall (which internally updates St), *all* values in $\mathcal{S}$ are uniformly distributed from A's point of view, until it learns any particular value $St' \in \mathcal{S}$ (note that after such St' becomes known to A, it is able to infer all values that were added to $\mathcal{S}$ after St'). Therefore, the probability that for any particular query of the form $\mathsf{H}\left(St', ID\right)$, $St' \in \mathcal{S}$, is at most $(q_P+2)/2^\kappa$. Since A makes at most q_H such queries of the form $\mathsf{H}\left(St', ID\right)$, the probability that for any of them, $St' \in \mathcal{S}$, is at most $\left(q_H(q_P+2)\right)/2^\kappa$, which proves the lemma. ■

4.1.2 Unforgeability

We now turn towards the unforgeability of our construction. Before giving the proof, we provide some intuition about our proof technique. At a high level, the idea is to reduce the security of the stateful wallet scheme $\mathsf{swal}[\mathsf{H}]$ (relative to $\mathbf{wunf}_{\mathsf{swal}[\mathsf{H}]}$) to the security of RSig (relative to $\mathbf{uf\text{-}cma\text{-}hrk}_{\mathsf{RSig}}$). As such, the proof consists mainly of the description of a reduction C trying to come up with a valid forgery in order to win the game $\mathbf{uf\text{-}cma\text{-}hrk}_{\mathsf{RSig}}$ by simulating $\mathbf{wunf}_{\mathsf{swal}[\mathsf{H}]}$ to an adversary A. Recall that C obtains a public key pk_{C} from its challenger in $\mathbf{uf\text{-}cma\text{-}hrk}_{\mathsf{RSig}}$ and has access to oracles $\mathtt{Rand}$, $\mathtt{RSign}$. It can call the oracle $\mathtt{Rand}$ to obtain a random value ρ. Later, C can use the signing oracle $\mathtt{RSign}$ on input (m, ρ), which provides signatures on a message m of C's choice under the rerandomized key $pk' := \mathsf{swal}[\mathsf{H}].\mathsf{RandPK}(pk_{\mathsf{C}}, \rho)$. Note that C can query $\mathtt{RSign}$ also to get signatures under pk_{C} by setting $\rho = \epsilon$. C's goal is to simulate the oracles in the $\mathbf{wunf}_{\mathsf{swal}[\mathsf{H}]}$ experiment and to suitably embed pk_{C} into the key pk_{ID^*} under which A eventually returns a forgery (σ^*, m^*). The hope is that it can use (σ^*, m^*) to win $\mathbf{uf\text{-}cma\text{-}hrk}_{\mathsf{RSig}}$.

A promising approach is therefore to embed pk_{C} into the master public key mpk within the simulation. This way, every answer to a query $\mathsf{PK}(ID)$ can easily be computed as $(\omega_{ID}, \cdot) \leftarrow \mathsf{H}(St, ID)$, $pk_{ID} \leftarrow \mathsf{swal}[\mathsf{H}].\mathsf{RandPK}(mpk, \omega_{ID})$. To simulate any signature under pk_{ID} to A, C can make a query of the form $\mathtt{RSign}(\hat{m}, \omega_{ID})$, where $\hat{m} = (pk_{ID}, m)$. When A returns the forgery (m^*, σ^*, ID^*), it is valid under the following conditions: (i) pk_{ID^*} is a valid session public key that was returned to A as the answer to a query $\mathsf{PK}(ID^*)$, (ii) A has not yet queried $\mathtt{WalSign}$ for a signature on m^* under pk_{ID^*}, (iii) the signature σ^* is valid, i.e., $\mathsf{swal}[\mathsf{H}].\mathsf{Verify}(pk_{ID^*}, \sigma^*, m^*) = 1$. As part of the proof, we show that C can win $\mathbf{uf\text{-}cma\text{-}hrk}_{\mathsf{RSig}}$ by returning the forgery (m^*, σ^*, ρ^*), where $\rho^* = \omega^*_{ID}$.

Theorem 4.3 *Let A be an algorithm that plays in the unforgeability game $\mathbf{wunf}_{\mathsf{swal}[\mathsf{H}]}$, where $\mathsf{swal}[\mathsf{H}]$ denotes the construction defined in Figure 7. Then if RSig is a signature scheme with uniquely rerandomizable keys, then there exists an algorithm C running in roughly the same time as A, such that*

$$\mathsf{Adv}^{\mathsf{A}}_{\mathbf{wunf},\mathsf{swal}[\mathsf{H}]} \leq \mathsf{Adv}^{\mathsf{C}}_{\mathbf{uf\text{-}cma\text{-}hrk},\mathsf{RSig}} + \frac{q^2}{p}$$

where q is the number of random oracle queries that A makes.

Proof. Consider an adversary A playing $\mathbf{wunf}_{\mathsf{swal}[\mathsf{H}]}$. As such, A is given the initial master public key mpk and the initial state St, and is granted access to the oracles $\mathsf{PK}, \mathtt{WalSign}$ and the random oracle H. We prove the Theorem via a sequence of games.

Game $\mathbf{G}_0$: This game behaves exactly as $\mathbf{wunf}_{\mathsf{swal}[\mathsf{H}]}$, i.e., $\mathbf{G}_0 := \mathbf{wunf}_{\mathsf{swal}[\mathsf{H}]}$. Internally however, $\mathbf{G}_0$ additionally sets $\mathtt{flag} \leftarrow \mathsf{true}$, whenever there is a call of the form $\mathsf{PK}(ID)$, such that the tuple (sk_{ID}, pk_{ID}) of session keys corresponding to this query, collides with a pair of session keys that was previously derived for another identity $ID' \neq ID$, i.e., $(pk_{ID}, sk_{ID}) = (pk_{ID'}, sk_{ID'}) = SSNKeys[ID']$.

Game $\mathbf{G}_1$: $\mathbf{G}_1$ behaves as $\mathbf{G}_0$, but aborts whenever $\mathtt{flag}$ is set to true. We let $E_{0,1}$ denote the event that $\mathtt{flag} = \mathsf{true}$ during the execution of $\mathbf{G}_1$.

Claim 4.4 $\Pr[E_{0,1}] \leq \frac{q^2}{p}$.

Proof. A collision of the form $(pk_{ID}, sk_{ID}) = (pk_{ID'}, sk_{ID'})$, where $ID \neq ID'$ implies that

$$\mathsf{RSig}.\mathsf{RandPK}(mpk, \omega_{ID}) = \mathsf{RSig}.\mathsf{RandPK}(mpk, \omega_{ID'}).$$

From the property of signature scheme with uniquely rerandomizable keys of RSig, this would mean $\omega_{ID} = \omega_{ID'}$, where $(\omega_{ID}, \cdot) = \mathsf{H}(\cdot, ID)$, $(\omega_{ID'}, \cdot) = \mathsf{H}(\cdot, ID')$. Since there are q queries to H the probability of event $E_{0,1}$ is bounded by $\frac{q^2}{p}$. ■

Thus, $\mathsf{Adv}^{\mathsf{A}}_{\mathbf{wunf},\mathsf{swal}[\mathsf{H}]} \leq \mathsf{Adv}^{\mathsf{A}}_{\mathbf{G}_1} + \frac{q^2}{p}$.

Next, we show how winning game $\mathbf{uf\text{-}cma\text{-}hrk}_{\mathsf{RSig}}$ reduces to winning game $\mathbf{G}_1$. To this end, we describe an algorithm $\mathsf{C}^{\mathtt{Rand},\mathtt{RSign}}$ (depicted in Figure 8) that plays in game $\mathbf{uf\text{-}cma\text{-}hrk}_{\mathsf{RSig}}$. C obtains as input a public key pk_{C} and is given access to the oracles $\mathtt{Rand}$ and $\mathtt{RSign}$. C simulates $\mathbf{G}_1$ to A as described in the following.

```
Algorithm $\mathsf{C}^{\mathtt{RSign},\mathtt{Rand}}(pk_{\mathsf{C}})$
$St \xleftarrow{\$} \{0,1\}^\kappa$
$(m^*, \sigma^*, ID^*) \xleftarrow{\$} \mathsf{A}^{\mathsf{PK},\mathsf{WSign},\mathsf{H}}(mpk, St)$
If $SSNKeys[ID^*] = \perp$: Abort
If $m^* \in Sigs[ID^*]$: Abort
$(pk_{ID^*}, \omega_{ID^*}) \leftarrow SSNKeys[ID^*]$
If $\mathsf{SWal}[\mathsf{H}].\mathsf{Verify}(pk_{ID^*}, \sigma^*, m^*) = 0$ :
    Abort
$\hat{m}^* \leftarrow (pk_{ID^*}, m^*)$
Return $(\hat{m}^*, \sigma^*, \omega_{ID^*})$

Procedure $\mathtt{PK}\,(ID)$ //Once per ID
$(\omega_{ID}, St) \leftarrow \mathsf{H}(St, ID)$
$pk_{ID} \leftarrow \mathsf{SWal}[\mathsf{H}].\mathsf{RandPK}(mpk, \omega_{ID})$
If $(pk_{ID}, \omega_{ID}) \in SSNKeys$: Abort
$SSNKeys[ID] \leftarrow (pk_{ID}, \omega_{ID})$
Return $pk_{ID}$

Procedure $\mathtt{WalSign}(m, ID)$
If $SSNKeys[ID] = \perp$: Return $\perp$
$(pk_{ID}, \omega_{ID}) \leftarrow SSNKeys[ID]$
$\hat{m} \leftarrow (pk_{ID}, m)$
$\sigma \leftarrow \mathtt{RSign}(\hat{m}, \omega_{ID})$
$Sigs[ID] \leftarrow Sigs[ID] \cup \{\hat{m}\}$
Return $\sigma$

Procedure $\mathsf{H}\,(s)$
If $H[s] \neq \perp$
    Return $H[s]$
$\rho \xleftarrow{\$} \mathtt{Rand}$
$\varphi \xleftarrow{\$} \{0,1\}^\kappa$
$H[s] \leftarrow (\rho, \varphi)$
Return $H[s]$
```

Figure 8: C's simulation of $\mathbf{wunf}_{\mathsf{swal}[\mathsf{H}]}$ to A.

Setup. C first samples an initial state $St \xleftarrow{\$} \{0,1\}^\kappa$ and uses the public key pk_{C} from the $\mathbf{uf\text{-}cma\text{-}hrk}_{\mathsf{RSig}}$ game as the master public key mpk in its simulation of $\mathbf{wunf}_{\mathsf{swal}[\mathsf{H}]}$, i.e., it runs A on input $mpk = pk_{\mathsf{C}}, St$ in $\mathbf{wunf}_{\mathsf{swal}[\mathsf{H}]}$. Throughout the game, C keeps updating St each time it answers a query to PK from A, as we describe below.

Simulation of Random Oracle Queries. C has to answer queries of the form $\mathsf{H}\,(s)$: Queries of this type are simulated in the programmable random oracle model as follows. When A makes a query of the form $\mathsf{H}(s)$, C returns $H[s]$ if it was already set. Otherwise, it proceeds as follows. Firstly, C fetches $\rho \xleftarrow{\$} \mathtt{Rand}$ by querying the oracle $\mathtt{Rand}$. Let us note that $\mathtt{Rand}$ internally updates $\mathsf{RList} \leftarrow \mathsf{RList} \cup \{\rho\}$. Secondly, C freshly samples $\varphi \xleftarrow{\$} \{0,1\}^\kappa$. Finally, C returns $H[s] = (\rho, \varphi)$.

Simulation of Public Key Queries. C answers a call of A to $\mathtt{PK}\,(ID)$ by computing pk_{ID} as $pk_{ID} \leftarrow \mathsf{RSig}.\mathsf{RandPK}(pk_{\mathsf{C}}, \omega_{ID})$ where $(\omega_{ID}, St) \leftarrow \mathsf{H}\,(St, ID)$. If C detects a collision among (pk_{ID}, ω_{ID}) and a value previously stored in $SSNKeys$, C aborts the simulation. Otherwise, it sets $SSNKeys[ID] \leftarrow (pk_{ID}, \omega_{ID})$ and returns pk_{ID}.

Simulation of Signing Queries. When A queries $\mathtt{WalSign}$ on input (m, ID), C first recovers the pair $(pk_{ID}, \omega_{ID}) \leftarrow SSNKeys[ID]$ (it returns $\perp$ if $SSNKeys[ID] = \perp$). Next, C sets $\hat{m} = (pk_{ID}, m)$ and obtains $\sigma \xleftarrow{\$} \mathtt{RSign}(\hat{m}, \omega_{ID})$ by querying its own challenge signing oracle. Since $\mathsf{H}\,(\cdot,\cdot)$ is programmed as explained above by making a call to $\mathtt{Rand}$, we know that $\omega_{ID} \in \mathsf{RList}$. Hence, the query $\mathtt{RSign}(\hat{m}, \omega_{ID})$ is indeed valid, i.e, does not return $\perp$. From the definition of signature schemes with rerandomizable keys, $\mathsf{SWal}[\mathsf{H}].\mathsf{Verify}(pk_{ID}, \sigma, m) = \mathsf{RSig}.\mathsf{Verify}(\mathsf{RSig}.\mathsf{RandPK}(pk_{\mathsf{C}}, \omega_{ID}), \sigma, \hat{m})) = 1$, and so the simulated signatures are also correctly distributed.

Extracting the forgery. When A returns the tuple (m^*, σ^*, ID^*), C aborts if it encounters any of the cases in which $\mathbf{G}_1$ would return 0 at this point (c.f. Figure 8). Otherwise it proceeds as follows. It first recovers the pair $(pk_{ID^*}, \omega_{ID^*}) \leftarrow SSNKeys[ID^*]$, and then returns $(\hat{m}^*, \sigma^*, \omega_{ID^*}) = ((pk_{ID^*}, m^*), \sigma^*, \omega_{ID^*})$. $(\hat{m}^*, \sigma^*, \omega_{ID^*})$ is a valid forgery in $\mathbf{uf\text{-}cma\text{-}hrk}$ game since:

1. From the simulation, we have that $pk_{ID^*} = pk_{\mathsf{C}} \cdot \omega_{ID^*}$ and $\omega_{ID^*} \in \mathsf{RList}$.
2. Since $\mathsf{swal}[\mathsf{H}].\mathsf{Verify}(pk_{ID^*}, \sigma^*, m^*) = 1$, it follows from the previous point that $\mathsf{RSig}.\mathsf{Verify}(pk_{ID^*}, \sigma^*, \hat{m}^*) = 1$.
3. $m^* \notin Sigs[ID^*]$ implies that C never simulated a signature on message m^* under public key pk_{ID^*} to A before. Since every identifier has a unique key in $\mathbf{G}_1$, it follows that C never made a query of the form $\mathtt{RSign}(\hat{m}^*, \cdot)$ throughout its simulation. Consequently, $\hat{m}^* \notin Sigs$.

It is clear that C provides a perfect simulation of $\mathbf{G}_1$ to A. Therefore, we obtain

$$\mathsf{Adv}^{\mathsf{A}}_{\mathbf{wunf},\mathsf{swal}[\mathsf{H}]} \le \mathsf{Adv}^{\mathsf{A}}_{\mathbf{G}_1,\mathsf{swal}[\mathsf{H}]} + \frac{q^2}{p} = \mathsf{Adv}^{\mathsf{C}}_{\mathbf{uf\text{-}cma\text{-}hrk},\mathsf{RSig}} + \frac{q^2}{p},$$

```
Algorithm $\mathsf{EC[H].Gen}\,(par)$
00 $x \xleftarrow{\$} \mathbb{Z}_p$
01 $X \leftarrow x \cdot G$
02 $sk \leftarrow x$
03 $pk \leftarrow X$
04 Return $(pk, sk)$

Algorithm $\mathsf{EC[H].Sign}\,(sk = x, m)$
05 $z \leftarrow \mathsf{H}\,(m)$
06 $t \xleftarrow{\$} \mathbb{Z}_p$
07 $(e_x, e_y) \leftarrow t \cdot G$
08 $r \leftarrow e_x \mod p$
09 If $r = 0 \mod p$
10     Goto Step 2
11 $s \leftarrow t^{-1}\,(z + rx) \mod p$
12 If $s = 0 \mod p$
13     Goto Step 2
14 Return $\sigma := (r, s)$

Algorithm $\mathsf{EC[H].Verify}\,(pk = X, \sigma, m)$
15 Parse $(r, s) \leftarrow \sigma$
16 If $(r, s) \notin \mathbb{Z}_p$
17     Return 0
18 $w \leftarrow s^{-1} \mod p$
19 $z \leftarrow \mathsf{H}\,(m)$
20 $u_1 \leftarrow zw \mod p$
21 $u_2 \leftarrow rw \mod p$
22 $(e_x, e_y) \leftarrow u_1 \cdot G + u_2 \cdot X$
23 If $(e_x, e_y) = (0, 0)$
24     Return 0
25 Return $r = e_x \mod p$
```

Figure 9: $\mathsf{EC}\,[\mathsf{H}] = (\mathsf{EC[H].Gen}, \mathsf{EC[H].Sign}, \mathsf{EC[H].Verify})$: ECDSA Signature scheme relative to elliptic curve $\mathbb{E}$ and hash function $\mathsf{H}\colon \{0,1\}^* \to \mathbb{Z}_p$.

```
Algorithm $\mathsf{REC[H_0].Sign}\,(sk, m)$
00 $\psi \xleftarrow{\$} \{0,1\}^\kappa$
01 $\hat{m} \leftarrow (pk, \psi, m)$
02 $\sigma' \leftarrow \mathsf{EC[H_0].Sign}\,(sk, \hat{m})$
03 Return $\sigma = (\psi, \sigma')$

Algorithm $\mathsf{REC[H_0].Verify}\,(pk, \sigma, m)$
04 $(\psi, \sigma') \leftarrow \sigma$
05 $\hat{m} \leftarrow (pk, \psi, m)$
06 Return $\mathsf{EC[H_0].Verify}\,(pk, \sigma', \hat{m})$

Algorithm $\mathsf{REC[H_0].RandSK}\,(sk, \rho)$
00 $sk' \leftarrow sk \cdot \rho \bmod p$
01 Return $sk'$

Algorithm $\mathsf{REC[H_0].RandPK}\,(pk, \rho)$
02 $pk' \leftarrow pk \cdot \rho$
03 Return $pk'$
```

Figure 10: Salted and key-prefixed version of the ECDSA signature scheme with perfectly rerandomizable keys $\mathsf{REC[H_0]} := (\mathsf{REC[H_0].Gen} = \mathsf{EC[H_0].Gen}, \mathsf{REC[H_0].Sign}, \mathsf{REC[H_0].Verify}, \mathsf{REC[H_0].RandSK}, \mathsf{REC[H_0].RandPK})$ from the ECDSA signature scheme $\mathsf{EC[H_0]}$. $\mathsf{H_0}\colon \{0,1\}^* \to \mathbb{Z}_p$ denotes a hash function.

which implies the theorem.

■

5 A Construction from ECDSA

In this section, we prove security of a construction based on the $\mathsf{EC}\,[\mathsf{H}]$ scheme (cf. Figure 11). For the following discussion, let $\mathbb{E}\,(par)$ denote an elliptic curve with base point G and prime order p. Furthermore, assume hash functions $\mathsf{G}\colon \{0,1\}^* \to \mathbb{Z}_p$, $\mathsf{H_0}\colon \{0,1\}^* \to \mathbb{Z}_p$(modeled as random oracles). We prove that a salted variant of the standard $\mathsf{EC}\,[\mathsf{H}]$ scheme, denoted as $\mathsf{REC[H]}$ and depicted in Figure 10, satisfies the notion of unforgeability under honestly rerandomized keys.

5.1 Security Analysis of Our Construction

We now proceed to the main technical contribution of this paper, where we analyze the notion of unforgeability under honestly rerandomized keys of the construction $\mathsf{REC[H_0]}$ presented in Figure 10. We prove the following theorem.

Theorem 5.1 *Let $\mathsf{G}, \mathsf{H_0}\colon \{0,1\}^* \to \mathbb{Z}_p$ be hash functions (modeled as random oracles). Let A be an algorithm that plays in game* $\textbf{uf-cma-hrk}_{\mathsf{REC[H_0]}}$. *Then there exists an algorithm C running in roughly*

```
$\mathsf{Trf}[\mathsf{H},\mathsf{G}]_{\mathsf{EC}}(m_0, m_1, \sigma_1, \omega, X_0, X_1)$
$z_0 \leftarrow \mathsf{H}(m_0)$
$z_1 \leftarrow \mathsf{G}(m_1)$
If $\left(\mathsf{Verify}^{\mathsf{G}}(\sigma_1, m_1, X_1) = 0\right) \vee \left(\omega \neq \frac{z_1}{z_0} \vee X_1 \neq X_0 \cdot \omega\right)$ :
    Return $\perp$
$(r, s_1) \leftarrow \sigma_1$
$s_0 \leftarrow \frac{s_1}{\omega} \mod p$
$\sigma_0 \leftarrow (r, s_0)$
Return $\sigma_0$
```

Figure 11: Figure shows the $\mathsf{Trf}_{\mathsf{ECDSA}}$ algorithm for hash functions $\mathsf{H}, \mathsf{G}\colon \{0,1\}^* \to \mathbb{Z}_p$.

the same time as A*, such that*

$$\mathsf{Adv}^{\mathsf{A}}_{\mathbf{uf\text{-}cma\text{-}hrk},\mathsf{REC}[\mathsf{H}_0]} \leq \mathsf{Adv}^{\mathsf{C}}_{\mathbf{uf\text{-}cma},\mathsf{EC}[\mathsf{G}]} + \frac{5q^2}{p},$$

where q is the number of random oracle queries that A *makes.*

Algorithm $\mathsf{Trf}[\mathsf{H},\mathsf{G}]_{\mathsf{EC}}$ The algorithm $\mathsf{Trf}[\mathsf{H},\mathsf{G}]_{\mathsf{EC}}$ which serves as an essential tool in our proof of Theorem 5.1 is presented in Figure 11. It takes as input two distinct messages m_0, m_1, two ECDSA public keys X_0, X_1 related via the offset ω and a signature σ_1 of m_1 wrt. public key X_1. The algorithm then carries out several consistency checks and if they pass outputs a valid signature σ_0 of m_0 under the related public key X_0. Notice that the two signatures σ_0 and σ_1 are valid with respect to different hash function, i.e., σ_1 is a signature with respect to G, while σ_0 is a signature with respect to H. This in particular implies that the transformation in $\mathsf{Trf}[\mathsf{H},\mathsf{G}]_{\mathsf{EC}}$ does not result into a practical related key attack as both signatures σ_0 and σ_1 are valid with respect to different hash functions and the consistency checks in $\mathsf{Trf}[\mathsf{H},\mathsf{G}]_{\mathsf{EC}}$ strongly restrict on what messages the related signature can be computed.[5] The following lemma formalizes the properties of $\mathsf{Trf}[\mathsf{H},\mathsf{G}]_{\mathsf{EC}}$. The proof can be found in App. C.

Lemma 5.2 *Consider the algorithm* $\mathsf{Trf}[\mathsf{H},\mathsf{G}]_{\mathsf{EC}}$ *in Figure 11. Suppose that:*

- $\omega = \mathsf{G}(m_1)/\mathsf{H}(m_0) \in \mathbb{Z}_p$,
- $X_0, X_1 \in \mathbb{E}$ *s.t.* $X_0 = x_0 \cdot G$ *and* $X_1 = \omega \cdot X_0$,
- $\mathsf{EC}[\mathsf{G}].\mathsf{Verify}(X_1, \sigma_1, m_1) = 1$,
- $\sigma_0 \leftarrow \mathsf{Trf}[\mathsf{H},\mathsf{G}]_{\mathsf{EC}}(m_0, m_1, \sigma_1, \omega, X_0, X_1)$.

Then $\mathsf{EC}[\mathsf{H}].\mathsf{Verify}(X_0, \sigma_0, m_0) = 1$.

Before giving the formal proof, we give some intuition about the main difficulties that we need to overcome. At a high level, the idea is to reduce the security of the salted ECDSA construction $\mathsf{REC}[\mathsf{H}_0]$(relative to $\mathbf{uf\text{-}cma\text{-}hrk}_{\mathsf{REC}[\mathsf{H}_0]}$) to the security of $\mathsf{EC}[\mathsf{G}]$ (relative to $\mathbf{uf\text{-}cma}_{\mathsf{EC}[\mathsf{G}]}$). As such, the proof consists mainly of the description of a reduction C trying to come up with a valid forgery in order to win the game $\mathbf{uf\text{-}cma}_{\mathsf{EC}[\mathsf{G}]}$ by simulating $\mathbf{uf\text{-}cma\text{-}hrk}_{\mathsf{REC}[\mathsf{H}_0]}$ to the adversary A. C obtains a public key pk_{C} from its challenger and can query a signing oracle $\mathtt{SignO}(\cdot)$ which provides signatures on messages of C's choice under pk_{C}. It also can query the random oracle G. C's goal is to simulate the oracles in the $\mathbf{uf\text{-}cma\text{-}hrk}_{\mathsf{REC}[\mathsf{H}_0]}$ experiment and to suitably embed pk_{C} into the key pk^* under which A eventually returns a forgery (σ^*, m^*, ρ^*). The hope is that it can use (σ^*, m^*, ρ^*) to win $\mathbf{uf\text{-}cma}_{\mathsf{EC}[\mathsf{G}]}$.

C embeds pk_{C} as A's input public key pk. This allows C to rerandomize pk into pk' which is a crucial requirement for answering oracle queries posed by A. However, there are several issues with this approach. Firstly, C is not aware of any of the secret keys for the public keys generated as $pk' \leftarrow pk \cdot \rho = pk_{\mathsf{C}} \cdot \rho$.

[5] The RKA against ECDSA can also be deployed when setting $\mathsf{H} = \mathsf{G}$. However, this attack is not particularly useful for our simulation argument. For the simulation argument we require to move signatures between different hash functions.

Secondly, the signatures obtained by making a query $\texttt{SignO}(\cdot)$ to C's challenger are only valid under pk_{C}, so cannot be directly used to simulate signing queries of the form $\texttt{RSign}(m, \rho)$ to A. To solve the latter problem, C can convert a signature of the form $\sigma \leftarrow \texttt{SignO}(m')$ under pk_{C} into a signature $\hat{\sigma}$ under pk', and on message $\hat{m}$ using algorithm $\mathsf{Trf}[\mathsf{H}_0, \mathsf{G}]_{\mathsf{EC}}$. Here, pk_{C} and pk' are related as $pk_{\mathsf{C}} = pk' \cdot \rho^{-1}$, and $\rho^{-1} = \frac{G(m')}{\mathsf{H}_0(\hat{m})}$. Similarly, it can convert a forgery (σ^*, m^*) under an arbitrary related key pk^* into a forgery that is valid under pk_{C}, using $\mathsf{Trf}[\mathsf{G}, \mathsf{H}_0]_{\mathsf{EC}}$ in the "reverse" direction (note the inverted order of H_0 and G). To satisfy the relationship between the (hash of) messages involved in the signatures, C needs to carefully program the random oracle H_0 to make everything consistent with what A expects to see. This gets even more complicated because A can make direct queries to the programmed oracle $\mathsf{H}_0(\cdot)$ where each of the queries should look random from A's point of view.

We now turn to the formal proof of Theorem 5.1.

Proof. Consider an adversary A playing in Game $\textbf{uf-cma-hrk}_{\mathsf{REC}[\mathsf{H}_0]}$. As such A is granted access to the oracles $\texttt{Rand}, \texttt{RSign}$, and the random oracle $\mathsf{H}_0 \colon \{0,1\}^* \to \mathbb{Z}_p$. In the following, we use that $2^\kappa \leq p$. We prove the statement via a sequence of games. Each game $\mathbf{G}_{i(i>0)}$ is presented in Figure 13 via the description of the oracles that are modified with respect to the previous game $\mathbf{G}_{i-1}$. The exact differences of game $\mathbf{G}_i$ to game $\mathbf{G}_{i-1}$ are highlighted in the form of boxed pseudocode. Moreover, we denote by $E_{i-1,i}$ a difference event, where the indices of the event correspond to games $\mathbf{G}_{i-1}, \mathbf{G}_i$ that are affected by the event.

Game $\mathbf{G}_0$: The initial game $\mathbf{G}_0$ (Figure 12) corresponds to $\textbf{uf-cma-hrk}_{\mathsf{REC}[\mathsf{H}_0]}$, i.e., $\mathbf{G}_0 := \textbf{uf-cma-hrk}_{\mathsf{REC}[\mathsf{H}_0]}$. Since we are in the random oracle model, we explicitly list the random oracle H_0 in $\mathbf{G}_0$.

Game $\mathbf{G}_1$: In $\mathbf{G}_1$, the way that random oracle queries to H_0 from A are answered, is internally modified as follows. To answer queries to H_0, $\mathbf{G}_1$ internally keeps two lists H_0 and H_0' which it programs throughout its interaction with A. Depending on whether a queried message m contains as part of its prefix a public key pk', it programs $H_0[m]$ and $H_0'[m]$ in two different possible ways. Note that pk' is the result of rerandomizing pk as $pk' = pk \cdot \rho$, where $\rho \leftarrow \texttt{Rand}(\rho \in \mathsf{RList})$ is a previous answer to a oracle query $\texttt{Rand}$. We now analyze the three types of queries to H_0 that can occur.

- $\mathsf{H}_0[m] \neq \perp$: In this case, $\mathbf{G}_1$ returns $H_0[m]$.
- $\mathsf{H}_0[m] = \perp$ and m is of the form $m = (\cdot, pk', \cdot)$, s.t. $pk' = pk \cdot \rho$ for some $\rho \in \mathsf{RList}$: In this case, $\mathbf{G}_1$ computes $h \leftarrow \mathsf{G}(ctr)$, where $ctr \xleftarrow{\$} \{0,1\}^\kappa$. Consequently, $\mathbf{G}_1$ sets $H_0[m] \leftarrow \rho \cdot h \mod p$ and $H_0'[m] \leftarrow ctr$. It returns $H_0[m]$.
- Otherwise, $\mathbf{G}_1$ samples $h \xleftarrow{\$} \mathbb{Z}_p$ and sets $H_0[m] \leftarrow h$, $H_0'[m] \leftarrow \epsilon$. It then returns $H_0[m]$.

It is easy to see that all answers for queries to H_0 that $\mathbf{G}_1$ returns are uniformly distributed from A's perspective. This follows from the uniformity of output h computed via random oracle G. Therefore, $\mathbf{G}_1$ behaves exactly as $\mathbf{G}_0$.

Game $\mathbf{G}_2$: In $\mathbf{G}_2$, the way in which queries to $\texttt{Rand}$ are answered, is internally modified as follows. When A asks a query of the form $\texttt{Rand}$, the game aborts if there exists a message of the form $m = (\cdot, pk', \cdot)$ for which $H_0'[m]$ evaluates to ϵ and where pk' is the (rerandomized) key that corresponds to the return value ρ of $\texttt{Rand}$, i.e., $pk' = pk \cdot \rho$. The following claim bounds the probability of such an abort scenario.

Claim 5.3 Let $E_{1,2}$ denote the event that $\mathbf{G}_2$ aborts during a $\texttt{Rand}$ query, for which $H_0'[m]$ evaluates to ϵ, where $m = (\cdot, pk', \cdot)$. Then $\Pr[E_{1,2}] \leq \frac{q^2}{p}$.

Proof. During any particular call to the oracle $\texttt{Rand}$, this event can only occur if A has already made a query of the form $\mathsf{H}_0(m)$, where $m = (\cdot, pk', \cdot)$ (prior to the oracle $\texttt{Rand}$ returning the value ρ for this query). Since RList contains at most q values at any point during the game, any of them coincide with the (uniformly chosen) value ρ with probability at most $\frac{q}{p}$. Since keys are uniquely rerandomizable, a query of the form $\mathsf{H}_0(m)$ thus also has probability at most $\frac{q}{p}$ of having been made prior to this particular call to $\texttt{Rand}$. Since there at most q queries to $\texttt{Rand}$, it follows that $\Pr[E_{1,2}] \leq \frac{q^2}{p}$. ■

Since the games $\mathbf{G}_1$, $\mathbf{G}_2$ are equivalent unless the event $\Pr[E_{1,2}]$ occurs, $\mathbf{Adv}^{\mathsf{A}}_{\mathbf{G_2},\mathsf{REC}[\mathsf{H_0}]} \leq \mathbf{Adv}^{\mathsf{A}}_{\mathbf{G_1},\mathsf{REC}[\mathsf{H_0}]} + \Pr[E_{1,2}] \leq \mathbf{Adv}^{\mathsf{A}}_{\mathbf{G_1},\mathsf{REC}[\mathsf{H_0}]} + \frac{q^2}{p}$.

```
Game $\mathbf{G}_0$
$\mathsf{RList} \leftarrow \{\epsilon\}$
$\mathsf{bad} \leftarrow \mathsf{false}$
$(sk, pk) \xleftarrow{\$} \mathsf{REC}[\mathsf{H}_0].\mathsf{Gen}(par)$
$(m^*, \sigma^*, \rho^*) \xleftarrow{\$} \mathsf{C}^{\mathsf{H}_0,\mathtt{Rand},\mathtt{RSign}}(pk)$
$pk^* \leftarrow pk \cdot \rho^*$
If $m^* \in Sigs$: $\mathsf{bad} \leftarrow \mathsf{true}$
If $\rho^* \notin \mathsf{RList}$: $\mathsf{bad} \leftarrow \mathsf{true}$
$b \leftarrow \mathsf{REC}[\mathsf{H}_0].\mathsf{Verify}(pk^*, \sigma^*, m^*)$
Return $b \wedge \neg\mathsf{bad}$

Oracle $\mathtt{Rand}$
$\rho \xleftarrow{\$} \chi$
$\mathsf{RList} \leftarrow \mathsf{RList} \cup \{\rho\}$
Return $\rho$

Oracle $\mathtt{RSign}(m, \rho)$
If $\rho \notin \mathsf{RList}$: Return $\perp$
$\psi \xleftarrow{\$} \{0,1\}^\kappa$
$pk' \leftarrow pk \cdot \rho \mod p$
$sk' \leftarrow sk \cdot \rho \mod p$
$\hat{m} \leftarrow (\psi, pk', m)$
$\sigma \leftarrow \mathsf{REC}[\mathsf{H}_0].\mathsf{Sign}(\hat{m}, sk')$
$Sigs \leftarrow Sigs \cup \{m\}$
Return $(\psi, \sigma)$

Oracle $\mathsf{H}_0(m)$
If $H_0[m] \neq \perp$
    Return $H_0[m]$
$H_0[m] \xleftarrow{\$} \mathbb{Z}_p$
Return $H_0[m]$
```

Figure 12: Game $\mathbf{G}_0 = \mathbf{uf\text{-}cma\text{-}hrk}_{\mathsf{REC}[\mathsf{H}_0]}$ with adversary C.

```
Oracle $\mathsf{H}_0(m)$ in $\mathbf{G}_1$
If $H_0[m] \neq \perp$
    Return $H_0[m]$
Parse $m$ as $(\cdot, pk', \cdot)$
If $\exists \rho \in \mathsf{RList} : pk' = pk \cdot \rho$
    $ctr \leftarrow \{0,1\}^\kappa$
    $h \leftarrow \mathsf{G}(ctr)$
    $H_0[m] \leftarrow \rho \cdot h \mod p$
    $H_0'[m] \leftarrow ctr$
Else
    $h \xleftarrow{\$} \mathbb{Z}_p$
    $H_0[m] \leftarrow h$
    $H_0'[m] \leftarrow \epsilon$
Return $H_0[m]$

Oracle $\mathtt{Rand}$ in $\mathbf{G}_2$
$\rho \xleftarrow{\$} \chi$
$\boxed{pk' \leftarrow pk \cdot \rho}$
$\boxed{\forall m = (\cdot, pk', \cdot) :}$
    $\boxed{\text{If } H_0'[m] = \epsilon\text{: Abort}}$
$\mathsf{RList} \leftarrow \mathsf{RList} \cup \{\rho\}$
Return $\rho$

Oracle $\mathtt{RSign}(m, \rho)$ in $\mathbf{G}_3$
If $\rho \notin \mathsf{RList}$: Return $\perp$
$\psi \xleftarrow{\$} \{0,1\}^\kappa$
$pk' \leftarrow pk \cdot \rho \mod p$
$sk' \leftarrow sk \cdot \rho \mod p$
$\hat{m} \leftarrow (\psi, pk', m)$
$\boxed{\text{If } H_0'[\hat{m}] \neq \perp\text{: Abort}}$
$\hat{\sigma} \leftarrow \mathsf{EC}[\mathsf{H}_0].\mathsf{Sign}(sk', \hat{m})$
$Sigs \leftarrow Sigs \cup \{m\}$
Return $(\psi, \hat{\sigma})$

Oracle $\mathtt{RSign}(m, \rho)$ in $\mathbf{G}_4$
If $\rho \notin \mathsf{RList}$: Return $\perp$
$\psi \xleftarrow{\$} \{0,1\}^\kappa$
$pk' \leftarrow pk \cdot \rho$
$\hat{m} \leftarrow (\psi, pk', m)$
If $H_0'[\hat{m}] \neq \perp$: Abort
$\boxed{\text{Query } \mathsf{H}_0(\hat{m})}$
$\boxed{m' \leftarrow H_0'[\hat{m}]}$
$\boxed{\sigma' \leftarrow \mathsf{EC}[\mathsf{G}].\mathsf{Sign}(sk, m')}$
$\boxed{\hat{\sigma} \leftarrow \mathsf{Trf}[\mathsf{H}_0, \mathsf{G}]_{\mathsf{EC}}(\hat{m}, m', \sigma', \rho^{-1}, pk', pk)}$
$Sigs \leftarrow Sigs \cup \{m\}$
Return $(\psi, \hat{\sigma})$

main in $\mathbf{G}_5$
$(pk, sk) \leftarrow \mathsf{EC}.\mathsf{Gen}(par)$
$(m^*, \sigma^*, \rho^*) \xleftarrow{\$} \mathsf{A}^{\mathsf{H}_0,\mathtt{Rand},\mathtt{RSign}}(pk)$
$pk^* \leftarrow pk \cdot \rho^*$
$\boxed{\hat{m}^* \leftarrow (\psi, pk^*, m^*)}$
$\boxed{\text{If } H_0'[\hat{m}^*] = \epsilon\text{: Abort}}$
If $m^* \in Sigs$: $\mathsf{bad} \leftarrow \mathsf{true}$
If $\rho^* \notin \mathsf{RList}$: $\mathsf{bad} \leftarrow \mathsf{true}$
$b \leftarrow \mathsf{REC}[\mathsf{H}_0].\mathsf{Verify}(pk^*, \sigma^*, m^*)$
Return $b \wedge \neg\mathsf{bad}$
```

Figure 13: Games $\mathbf{G}_1$-$\mathbf{G}_5$

Game $\mathbf{G}_3$: In $\mathbf{G}_3$, the way in which signing queries from A are answered, is internally modified as follows. When A makes a query of the form $\mathtt{RSign}\,(m, \rho)$, $\mathbf{G}_3$ first checks whether $\rho \in \mathsf{RList}$ and if not, returns $\perp$. Otherwise, it samples $\psi \xleftarrow{\$} \{0,1\}^\kappa$, computes $pk' \leftarrow pk \cdot \rho, sk' := sk \cdot \rho \mod p$, and sets $\hat{m} \leftarrow \left(\psi, pk', m\right)$. If the list H_0' already contains an element for $H_0'\,[\hat{m}]$, i.e. $H_0'\,[\hat{m}] \neq \perp$, then the game aborts at this point. Otherwise, a signature $\hat{\sigma}$ is computed as $\hat{\sigma} \xleftarrow{\$} \mathsf{EC}[\mathsf{H}_0].\mathsf{Sign}\left(sk', \hat{m}\right)$. $\mathbf{G}_3$ subsequently returns $(\psi, \hat{\sigma})$. The only difference of game $\mathbf{G}_3$ to $\mathbf{G}_2$, is that game $\mathbf{G}_3$ potentially aborts at line 24 if $H_0'\,[\hat{m}] \neq \perp$. Hence, we obtain the following claim.

Claim 5.4 Let $E_{2,3}$ denote the event that $\mathbf{G}_3$ aborts during a signing query, when $H_0[\hat{m}] \neq \perp$, where $\hat{m} = (\psi, pk', m)$. Then $\Pr[E_{2,3}] \leq \frac{q^2}{p}$.

Proof. This event can only happen when A makes a correct guess of the message $\hat{m}$ and makes a query of the form $\mathsf{H}_0(\hat{m})$ prior to a $\mathtt{RSign}(m, \rho)$ query. $\hat{m}$ is constructed as $\hat{m} = (\psi, pk', m)$ where ψ is uniformly sampled as $\psi \xleftarrow{\$} \{0,1\}^\kappa$. Since A makes atmost q queries to $\mathsf{H}_0(\cdot)$, A can correctly guess a particular $\hat{m} = (\psi, pk', m)$ for a fixed m, with probability $\frac{q}{p}$. Since A makes at most q signing queries to $\mathtt{RSign}\,(m, \rho)$, A can correctly guess any $\hat{m}$ with a probability bounded by $\sum_{i=1}^{q} \frac{q}{p} \leq \frac{q^2}{p}$. ■

Since the games $\mathbf{G}_2$, $\mathbf{G}_3$ are equivalent unless the event $\Pr[E_{2,3}]$ occurs, $\mathbf{Adv}^{\mathsf{A}}_{\mathbf{G_2},\mathsf{REC}[\mathsf{H}_0]} \leq \mathbf{Adv}^{\mathsf{A}}_{\mathbf{G_3},\mathsf{REC}[\mathsf{H}_0]} + \Pr\left[E_{2,3}\right] \leq \mathbf{Adv}^{\mathsf{A}}_{\mathbf{G_3},\mathsf{REC}[\mathsf{H}_0]} + \frac{q^2}{p}$.

Game $\mathbf{G}_4$: In $\mathbf{G}_4$, the way that signing queries from A are answered, is again internally modified as follows. When A makes a query of the form $\mathtt{RSign}(m, \rho)$, $\mathbf{G}_4$ first checks whether $\rho \in \mathsf{RList}$ and if not, returns $\perp$. Otherwise, it samples $\psi \xleftarrow{\$} \{0,1\}^\kappa$ computes $pk' \leftarrow pk \cdot \rho$, and sets $\hat{m} \leftarrow \left(\psi, pk', m\right)$. The game aborts at this point if $H_0[\hat{m}] \neq \perp$. If it does not abort, it internally queries H_0 on input message $\hat{m}$. This means it queries $h \leftarrow \mathsf{G}\,(ctr)$, where $ctr \xleftarrow{\$} \{0,1\}^\kappa$. $\mathbf{G}_4$ internally sets $H_0[\hat{m}] \leftarrow \rho \cdot h \mod p$ and stores $H_0'[\hat{m}] \leftarrow ctr$. After making the query to H_0, $\mathbf{G}_4$ fetches $m' \leftarrow H_0'[\hat{m}]$, where m' was set to ctr during H_0 query. Since sk is known to the game, it can now compute the signature σ' as $\sigma' \xleftarrow{\$} \mathsf{EC}[\mathsf{G}].\mathsf{Sign}(sk, m')$. Finally, it computes and returns the signature $\hat{\sigma}$ as $\hat{\sigma} \leftarrow \mathsf{Trf}[\mathsf{H}_0, \mathsf{G}]_{\mathsf{EC}}(\hat{m}, m', \sigma', \rho^{-1}, pk', pk)$, where $pk = pk' \cdot \rho^{-1}$.

Claim 5.5 $\mathbf{Adv}^{\mathsf{A}}_{\mathbf{G_3},\mathsf{REC}[\mathsf{H}_0]} = \mathbf{Adv}^{\mathsf{A}}_{\mathbf{G_4},\mathsf{REC}[\mathsf{H}_0]}$

Proof. We argue that in both games, the answers to signing queries are identically distributed. To this end, we analyze how $\mathbf{G}_4$ replies to a query of the form $\mathtt{RSign}\,(m, \rho)$. First note that the explicit query to H_0 at line 33 is implicitly also made in $\mathbf{G}_3$ at line 25 and therefore does not change the behaviour of $\mathbf{G}_4$ (compared to $\mathbf{G}_3$). Next, $\mathbf{G}_4$ derives signature $(\psi, \hat{\sigma})$ on input (m, ρ) as $\hat{\sigma} \leftarrow \mathsf{Trf}[\mathsf{H}_0, \mathsf{G}]_{\mathsf{EC}}(\hat{m}, m', \sigma', \rho^{-1}, pk', pk)$, where $m' = H_0'[\hat{m}], pk = pk' \cdot \rho^{-1}, \mathsf{EC}[\mathsf{G}].\mathsf{Verify}(pk, \sigma', m') = 1$, and $\frac{\mathsf{G}(m')}{H_0[\hat{m}]} = \frac{h'}{H_0[\hat{m}]} = \frac{h'}{\rho \cdot h'} = \rho^{-1} \mod p$. It follows from Lemma 5.2 that $\hat{\sigma}$ constitutes a correct signature on message $\hat{m}$ and under public key pk' relative to $\mathsf{EC}[\mathsf{H}_0].\mathsf{Verify}$. It follows immediately that the signature $(\psi, \hat{\sigma})$ constitutes a valid signature relative to $\mathsf{REC}[\mathsf{H}_0].\mathsf{Verify}$. Moreover, the value of ψ is identically distributed in games $\mathbf{G}_3, \mathbf{G}_4$, which concludes the proof. ■

Game $\mathbf{G}_5$: $\mathbf{G}_5$ behaves identically to $\mathbf{G}_4$ except for the following modification in the main procedure: Upon receiving a forgery of the form $(m^*, \sigma^* = (\psi, \hat{\sigma}), \rho^*)$ from A, it sets $\hat{m}^* \leftarrow (\psi, pk^*, m^*)$ and aborts if $H_0'[\hat{m}^*] = \epsilon$.

Claim 5.6 Let $E_{4,5}$ be the event that $\mathbf{G}_5$ aborts if $H_0'[\hat{m}^*] = \epsilon$, where $\hat{m}^* = (\psi, pk^*, m^*)$. Then $\Pr[E_{4,5}] \leq \frac{q^2}{p}$.

Proof. The only way this event can happen, is if A manages to make a query of the form $\mathsf{H}_0(\hat{m}^*)$ before querying $\mathtt{Rand}$ to obtain the corresponding value of ρ^*. The proof of this claim follows in a similar way as the corresponding proof in claim 5.3. ■

Since the games $\mathbf{G}_4$, $\mathbf{G}_5$ are equivalent unless event $E_{4,5}$ occurs, $\mathbf{Adv}^{\mathsf{A}}_{\mathbf{G_4},\mathsf{REC}[\mathsf{H}_0]} \leq \mathbf{Adv}^{\mathsf{A}}_{\mathbf{G_5},\mathsf{REC}[\mathsf{H}_0]} + \frac{q^2}{p}$.

Reduction to UF-CMA security. We describe an algorithm $\mathsf{C}^{\mathtt{Sign0},\mathsf{G}}$ (depicted in Figure 14) that plays in the $\mathbf{uf\text{-}cma}_{\mathsf{EC}[\mathsf{G}]}$ game. C obtains as input a public key pk_{C} and is given access to the signing

```
main $\mathsf{C}^{\texttt{SignO},\mathsf{G}}(pk_{\mathsf{C}})$
$(m^*,\sigma^*,\rho^*) \xleftarrow{\$} \mathsf{A}^{\mathsf{H}_0,\texttt{Rand},\texttt{RSign}}(pk_{\mathsf{C}})$
$(\psi,\hat{\sigma}) \leftarrow \sigma^*$
$pk^* \leftarrow pk \cdot \rho^*$
$\hat{m}^* \leftarrow (\psi, pk^*, m^*)$
If $H_0'[\hat{m}^*] = \epsilon$: Abort
If $m^* \in Sigs$: $\mathsf{bad} \leftarrow \mathsf{true}$
If $\rho^* \notin \mathsf{RList}$:
    $\mathsf{bad} \leftarrow \mathsf{true}$
$b \leftarrow \mathsf{REC}[\mathsf{H}_0].\mathsf{Verify}\,(pk^*,\sigma^*,m^*)$
If $\neg b \vee \mathsf{bad}$: Abort
$m' \leftarrow H_0'[\hat{m}^*]$
$\texttt{tmp} \leftarrow (m',\hat{m}^*,\hat{\sigma}^*,\rho^*,pk_{\mathsf{C}},pk^*)$
$\sigma' \leftarrow \mathsf{Trf}[\mathsf{G},\mathsf{H}_0]_{\mathsf{EC}}(\texttt{tmp})$
Return $(m',\sigma')$

Procedure $\texttt{Rand}$
$\rho \xleftarrow{\$} \chi$
$pk' \leftarrow pk \cdot \rho$
$\forall m = (\cdot, pk', \cdot)$ :
    If $H_0'[m] = \epsilon$: Abort
$\mathsf{RList} \leftarrow \mathsf{RList} \cup \{\rho\}$
Return $\rho$

Procedure $\texttt{RSign}\,(m,\rho)$
If $\rho \notin \mathsf{RList}$: Return $\perp$
$\psi \xleftarrow{\$} \{0,1\}^\kappa$
$pk' \leftarrow pk \cdot \rho$
$\hat{m} \leftarrow (\psi, pk', m)$
If $H_0'[\hat{m}] \neq \perp$: Abort
Query $\mathsf{H}_0(\hat{m})$
$m' \leftarrow H_0'[\hat{m}]$
$\sigma' \leftarrow \texttt{SignO}(m')$
$\texttt{tmp} \leftarrow (\hat{m}, m', \sigma', \rho^{-1}, pk', pk_{\mathsf{C}})$
$\hat{\sigma} \leftarrow \mathsf{Trf}[\mathsf{H}_0,\mathsf{G}]_{\mathsf{EC}}(\texttt{tmp})$
$Sigs \leftarrow Sigs \cup \{m\}$
Return $(\psi,\hat{\sigma})$

Procedure $\mathsf{H}_0\,(m)$
If $H_0\,[m] \neq \perp$
    Return $H_0\,[m]$
Parse $m$ as $(\cdot, pk', \cdot)$
If $\exists \rho \in \mathsf{RList} : pk' = pk \cdot \rho$
    $ctr \leftarrow \{0,1\}^\kappa$
    $h \leftarrow \mathsf{G}\,(ctr)$
    $H_0\,[m] \leftarrow \rho \cdot h \mod p$
    $H_0'\,[m] \leftarrow ctr$
Else
    $h \xleftarrow{\$} \mathbb{Z}_p$
    $H_0\,[m] \leftarrow h$
    $H_0'\,[m] \leftarrow \epsilon$
Return $H_0\,[m]$
```

Figure 14: Reduction to UF-CMA game.

oracle $\texttt{SignO}$ to obtain signatures under pk_{C} under messages of its choice. Furthermore, C has access to the random oracle G. C runs A on input pk_{C} and simulates $\mathbf{G}_5$ to A as described in Figure 14.

Simulation of Randomness Queries. Queries to $\texttt{Rand}$ from A do not require knowledge of the secret key corresponding to pk_{C} and hence are straight forward to simulate.

Simulation of Random Oracle Queries. C's simulation of random oracle queries coincides with the above programming strategy that is already internally present in $\mathbf{G}_5$.

Simulation of Signing Queries. Recall that in $\mathbf{G}_5$, queries of the form $\texttt{RSign}\,(m,\rho)$ internally prompt the computation of signature $\sigma' = \mathsf{EC}[\mathsf{G}].\mathsf{Sign}(sk_{\mathsf{C}}, m')$, where $m' \leftarrow ctr$. Since C does not know sk_{C}, it needs to compute σ' via a call to its signing oracle, i.e., as $\sigma' \leftarrow \texttt{SignO}(m')$. Other than that C simulates such a query exactly as internally done for $\mathbf{G}_5$.

Extracting the forgery. When the tuple (m^*,σ^*,ρ^*) is returned as an answer from A, C first parses it as $(m^*,\sigma^*,\rho^*) = (m^*,(\psi^*,\hat{\sigma}^*),\rho^*)$, checks whether it constitutes a valid forgery, and aborts otherwise (note that in this case, $\mathbf{G}_5$ would return 0, so C can safely abort). In case C does not abort, it computes $pk^* = pk_{\mathsf{C}} \cdot \rho^*$, where pk^* is the public key under which A's forgery is valid. C computes $\hat{m}^* \leftarrow (\psi^*, pk^*, m^*)$ and if $H_0'\,[\hat{m}^*] = \epsilon$, it aborts. Otherwise, C fetches $m' \leftarrow H_0'\,[\hat{m}^*]$ and computes

$$\sigma' \leftarrow \mathsf{Trf}[\mathsf{G},\mathsf{H}_0]_{\mathsf{ECDSA}}\,(m',\hat{m}^*,\hat{\sigma}^*,\rho^*,pk_{\mathsf{C}},pk^*)\,.$$

Since $H_0\,[\hat{m}^*] = \mathsf{G}\,(H_0'\,[\hat{m}^*]) \cdot \rho^* = \mathsf{G}\,(m') \cdot \rho^*$, we have that $\frac{H_0[\tilde{m}^*]}{\mathsf{G}(m')} = \frac{\mathsf{G}(m')\cdot\rho^*}{\mathsf{G}(m')} = \rho^*$. Together with $pk^* = pk_{\mathsf{C}} \cdot \rho^*$ and $\mathsf{EC}[\mathsf{H}_0].\mathsf{Verify}(pk^*,\hat{\sigma}^*,\hat{m}^*) = 1$, Lemma 5.2 implies that

$$\mathsf{EC}[\mathsf{G}].\mathsf{Verify}\,(pk_{\mathsf{C}},\sigma',m') = 1.$$

Claim 5.7 (m', σ') constitutes a valid forgery in $\mathbf{uf\text{-}cma}_{\mathsf{EC[G]}}$ with probability $1 - q^2/p$.

Proof. We have to show that the query $\mathtt{SignO}(m')$ was not made by C during its simulation and hence (m', σ') is a valid forgery in $\mathbf{uf\text{-}cma}_{\mathsf{EC[G]}}$. Note that A has not made a query of the form $\mathtt{RSign}\,(m^*, \rho^*)$ throughout the simulation. Namely, if it had, (m^*, σ^*, ρ^*) would not constitute a valid forgery in $\mathbf{G}_5$ and the simulation would have aborted at this point. This implies that C never had to simulate a query $\mathtt{RSign}(m^*, \rho^*)$ to A which entailed a H_0 query on message $\hat{m}^* \leftarrow (\psi^*, pk^*, m^*)$. Hence, m' associated with query $\mathsf{H}_0(\hat{m}^*)$ was not queried by C to the oracle $\mathtt{SignO}$ in any query of the form $\mathtt{RSign}\,(m, \rho)$ with $m \neq m^*$ unless there exist (any) two values m_1, m_2 s.t. $H_0'[m_1] = H_0'[m_2] \neq \perp$. It is easy to see that this happens with probability at most q^2/p during C's simulation, since all values that C queries to the oracle $\mathtt{SignO}$ are sampled independently and uniformly at random from $\{0,1\}^\kappa$. ■

From claims 5.3-5.6, we have $\mathbf{Adv}^{\mathsf{A}}_{\mathbf{G_0},\mathsf{REC[H_0]}} \leq \mathbf{Adv}^{\mathsf{A}}_{\mathbf{G_5},\mathsf{REC[H_0]}} + \frac{4q^2}{p}$. Since C provides a perfect simulation of $\mathbf{G}_5$ to A up to an error of q^2/p, as shown in the previous claim, we obtain

$$\mathsf{Adv}^{\mathsf{A}}_{\mathbf{uf\text{-}cma\text{-}hrk},\mathsf{REC[H_0]}} \leq \mathsf{Adv}^{\mathsf{A}}_{\mathbf{G}_5} + \frac{4q^2}{p} \leq \mathsf{Adv}^{\mathsf{C}}_{\mathbf{uf\text{-}cma},\mathsf{EC[G]}} + \frac{4q^2}{p},$$

which implies the theorem.

■

6 Practical Considerations

Synchronizing hot/cold wallet. To achieve correctness according to Definition 3.2, the cold wallet and hot wallet (party A and party B in Fig. 3) respectively, need to derive their keys in the same (ordered) sequence. Fortunately, this can be realized easily in practice. A simple solution is to use an increasing counter for every freshly derived pair of session keys in place of the *ID* argument. In this case, no additional synchronization between the hot and cold wallet is necessary. However, it is also possible to include a more complicated *ID* structure, where the *ID* is provided by the wallet user as an input parameter. Consider a scenario, where a wallet user Bob wants to receive some payment for some *ID*. To this end, the hot wallet generates a fresh session public key pk_{ID} for *ID* via $\mathsf{SWal.PKDer}$. Then, *ID* is added to the transaction $\mathtt{tx}$ that is published on the blockchain. Later, when Bob wants to spend the transaction via the cold wallet, he can extract the *ID* from $\mathtt{tx}$ to generate the corresponding secret key sk_{ID} on the cold wallet. Notice, of course, that the values for *ID* have to be chosen "somewhat randomly" as otherwise the unlinkability property of the wallet scheme is broken. One simple way to achieve this is to let the hot wallet encrypt the *ID* and add the ciphertext to the transaction that sends money to the address pk_{ID}.

Statefulness of our scheme. We point out that the state in our stateful wallet scheme SWal may make our scheme more complex to use in practice (as evidented from the previous discuss on synchronization). However, the state is *only* needed in order to achieve forward unlinkability after compromise of the hot wallet. The unforgeability property proven in our work also works for the simpler stateless wallets. Hence, if forward unlinkability is not needed, one can use a stateless version of our constructions and benefit from our security analysis (i.e., unlinkability *without* state compromise and unforgeability).

The winning condition of wallet unforgeability. In Figure 6 the adversary wins the game if she manages to output a valid forgery $(pk_{ID^*}, \sigma^*, m^*)$ such that $\mathsf{SWal.Verify}(pk_{ID^*}, \sigma^*, m^*) = 1$. We emphasize that in practice for breaking a wallet in, e.g., Bitcoin, it suffices that the adversary creates a transaction spending money from address pk_{ID^*} *and* is accepted by the miners. The latter is quite important because there is no reason why in legacy cryptocurrencies, miners should execute the $\mathsf{SWal.Verify}$ algorithm of our SWal construction. Fortunately, however, in Bitcoin miners implicitly execute $\mathsf{REC[H].Verify}$ when verifying transactions, and hence our scheme and its security analysis is compatible with Bitcoin.[6]

[6] At a more technical level, in Bitcoin if we want to spend money from an address pk_{ID}, then the spending transaction (that is signed with sk_{ID}) contains pk_{ID}. Hence, it has a form that is compatible with the verification done by $\mathsf{REC[H].Verify}$. In fact, our security proof can also be adjusted to match *exactly* with the verification that is carried out by the miners.

TRANSACTION COST ANALYSIS. To integrate our scheme into Bitcoin, we have to make sure that (a) transactions are salted, (b) they are pre-fixed[7] by the public key pk from which the money is sent, and (c) such transactions are accepted by the miners. Fortunately, in Bitcoin this can be achieved using the simple scripting language, and we explain it in detail in App. B.2. While the pre-fixing of the public key (b) is naturally happening in Bitcoin, the random salting (a) is non-standard and results into additional costs. We discuss them briefly below and compare them with the standard costs of creating transactions in Bitcoin (i.e., without salting). Consider a transaction $\mathtt{tx}_0$ that transfers money from the cold wallet to a new address, and hence in our scheme has to be randomized. Due to the mechanics of Bitcoin also the transaction $\mathtt{tx}_1$ that spends $\mathtt{tx}_0$ will include this random salt. Thus, our cost analysis includes these two transactions. We summarize the costs in `Satoshi` and USD, depending on whether the transaction gets included in the next block, or within the next 6 subsequent blocks. Note that confirmation of a transaction in an earlier block results into higher costs[8]

Table 1: Standard vs Randomized Transactions Costs

Transaction Type	Confirmation in next block Fees (`Satoshi`/ USD)	Confirmation in next 6 blocks Fees (`Satoshi`/ USD)
$\mathtt{tx}_0$ (Standard)	7665/0.54	2190/0.17
$\mathtt{tx}_1$ (Standard)	8505/0.60	2430/0.19
$\mathtt{tx}_0$ (*Randomized*)	7875/0.56	2250/0.18
$\mathtt{tx}_1$ (*Randomized*)	8610/0.61	2460/0.19

7 Conclusion

In this work, we focused on analyzing the security of deterministic wallets. We developed two new security guarantees that we call wallet unlinkability and wallet unforgeability, and showed a modular approach for constructing such wallets from certain signature schemes. At the technical level, we proved that a simple extension of the ECDSA-based hot/cold wallet as used in Bitcoin can be proven secure in our model. A natural extension of our work will be to consider the case of hierarchical wallets. However, the hierarchical setting will require a significantly more complex model (additional oracles, more complex bookkeeping). The security analysis in this setting is also believed to be more involved. Hence, it is certainly an excellent direction for future research to extend our model to the hierarchical case.

8 Acknowledgments

This work was partly supported by the DFG CRC 1119 CROSSING (project S7), the German Federal Ministry of Education and Research (BMBF) iBlockchain project, and the ERC Project ERCC (FP7/615074). Additionally, the first two authors of this work have received funding from the German Federal Ministery of Education and Research and the Hessen State Ministry for Higher Education, Research and the Arts within their joint support of the National Research Center for Applied Cybersecurity (CRISP).

We would like to thank Eike Kiltz for preliminary discussions on our model and Hendrik Amler for helping us in the practical evaluations. We are grateful to our anonymous reviewer from CCS for important comments on the modularization of our proofs using the notion of unforgeability under rerandomizable keys.

[7] Notice that in our generic wallet construction (c.f. Figure 7), messages are key pre-fixed to prevent from the related key attack. For the salted ECDSA construction to satisfy the property of signatures with uniquely rerandomizable keys (c.f. Figure 10), messages are again key-prefixed. The key prefixing in the latter case is necessary as an essential technique for the proof of Th 5.1. Although theoretically our ECDSA based wallet construction is key pre-fixed twice, in practice key pre-fixing the message once will be enough.

[8] We have used the currency value from [Cur19] timestamped on 14th May, 2019. Notice that the increase in costs are around 3% compared to standard Bitcoin transactions. However the cost increase also depends on the application, and we leave it as an interesting question for future work to provide an application-dependent optimization of costs.

References

[AGKK19] Myrto Arapinis, Andriana Gkaniatsou, Dimitris Karakostas, and Aggelos Kiayias. A formal treatment of hardware wallets. Cryptology ePrint Archive, Report 2019/034, 2019. https://eprint.iacr.org/2019/034. (Cited on page 5.)

[BDN18] Dan Boneh, Manu Drijvers, and Gregory Neven. Compact multi-signatures for smaller blockchains. Cryptology ePrint Archive, Report 2018/483, 2018. https://eprint.iacr.org/2018/483. (Cited on page 5.)

[BH19] Joachim Breitner and Nadia Heninger. Biased nonce sense: Lattice attacks against weak ECDSA signatures in cryptocurrencies. *IACR Cryptology ePrint Archive*, 2019:23, 2019. (Cited on page 5.)

[Bit18] BitcoinExchangeGuide. CipherTrace Releases Report Exposing Close to $1 Billion Stolen in Crypto Hacks During 2018. https://bitcoinexchangeguide.com/ciphertrace-releases-report-exposing-close-to-1-billion-stolen-in_-crypto-hacks-during-2018/, 2018. (Cited on page 1.)

[Blo18] Bloomberg. How to Steal $500 Million in Cryptocurrency. http://fortune.com/2018/01/31/coincheck-hack-how/, 2018. (Cited on page 1.)

[BLS04] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. *J. Cryptology*, 17(4):297–319, 2004. (Cited on page 4.)

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In V. Ashby, editor, *ACM CCS 93*, pages 62–73. ACM Press, November 1993. (Cited on page 6.)

[BR04] Mihir Bellare and Phillip Rogaway. Code-based game-playing proofs and the security of triple encryption. Cryptology ePrint Archive, Report 2004/331, 2004. http://eprint.iacr.org/2004/331. (Cited on page 6.)

[BR18] Michael Brengel and Christian Rossow. Identifying key leakage of bitcoin users. In *Research in Attacks, Intrusions, and Defenses - 21st International Symposium, RAID 2018, Heraklion, Crete, Greece, September 10-12, 2018, Proceedings*, pages 623–643, 2018. (Cited on page 5.)

[But13] Vitalik Buterin. Deterministic Wallets, Their Advantages and their Understated Flaws. https://bitcoinmagazine.com/articles/deterministic-wallets-advantages-flaw-1385450276/, 2013. (Cited on page 2, 5.)

[CEV14] Nicolas T. Courtois, Pinar Emirdag, and Filippo Valsorda. Private key recovery combination attacks: On extreme fragility of popular bitcoin key management, wallet and cold storage solutions in presence of poor RNG events. *IACR Cryptology ePrint Archive*, 2014:848, 2014. (Cited on page 5.)

[Cur19] Bitcoin Fees for Transactions. https://bitcoinfees.earn.com/, 2019. (Cited on page 23.)

[DKLS18] Jack Doerner, Yashvanth Kondi, Eysa Lee, and Abhi Shelat. Secure two-party threshold ECDSA from ECDSA assumptions. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 980–997, 2018. (Cited on page 5.)

[FF13] Marc Fischlin and Nils Fleischhacker. Limitations of the meta-reduction technique: The case of schnorr signatures. In *Advances in Cryptology - EUROCRYPT 2013*, pages 444–460, 2013. (Cited on page 5.)

[FKM+16] Nils Fleischhacker, Johannes Krupp, Giulio Malavolta, Jonas Schneider, Dominique Schröder, and Mark Simkin. Efficient unlinkable sanitizable signatures from signatures with re-randomizable keys. In Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang, editors, *PKC 2016, Part I*, volume 9614 of *LNCS*, pages 301–330. Springer, Heidelberg, March 2016. (Cited on page 3, 4, 5, 6, 7, 26.)

[FTS+18] Chun-I Fan, Yi-Fan Tseng, Hui-Po Su, Ruei-Hau Hsu, and Hiroaki Kikuchi. Secure hierarchical bitcoin wallet scheme against privilege escalation attacks. In *IEEE Conference on Dependable and Secure Computing, DSC 2018*, pages 1–8, 2018. (Cited on page 5.)

[GGN16] Rosario Gennaro, Steven Goldfeder, and Arvind Narayanan. Threshold-optimal DSA/ECDSA signatures and an application to bitcoin wallet security. In *Applied Cryptography and Network Security - ACNS 2016*, pages 156–174, 2016. (Cited on page 5.)

[GS15] Gus Gutoski and Douglas Stebila. Hierarchical deterministic bitcoin wallets that tolerate key leakage. In *Financial Cryptography and Data Security - 19th International Conference, FC 2015*, pages 497–504, 2015. (Cited on page 5.)

[KMP16] Eike Kiltz, Daniel Masny, and Jiaxin Pan. Optimal security proofs for signatures from identification schemes. In *Advances in Cryptology - CRYPTO 2016, Part II*, pages 33–61, 2016. (Cited on page 5.)

[Lig18a] Lightning Bitcoin mainnet. `https://graph.lndexplorer.com/`, 2018. (Cited on page 30.)

[Lig18b] Lightning RFC BOLT 3. `https://github.com/lightningnetwork/lightning-rfc/blob/master/03-transactions.md`, 2018. (Cited on page 30.)

[LN18] Yehuda Lindell and Ariel Nof. Fast secure multiparty ECDSA with practical distributed key generation and applications to cryptocurrency custody. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 1837–1854, 2018. (Cited on page 5.)

[MB18] Gregory Maxwell and Iddo Bentov. Deterministic Wallets. `https://www.cs.cornell.edu/~iddo/detwal.pdf`, 2018. (Cited on page 3.)

[Med18] Mediawiki. BIP32 Specification. `https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki`, 2018. (Cited on page 2.)

[MPas19] Antonio Marcedone, Rafael Pass, and abhi shelat. Minimizing trust in hardware wallets with two factor signatures. Cryptology ePrint Archive, Report 2019/006, 2019. `https://eprint.iacr.org/2019/006`. (Cited on page 5.)

[MSM+15] Hiraku Morita, Jacob C. N. Schuldt, Takahiro Matsuda, Goichiro Hanaoka, and Tetsu Iwata. On the security of the schnorr signature scheme and DSA against related-key attacks. In *ICISC 2015 - 18th International Conference, Seoul, South Korea, November 25-27, 2015, Revised Selected Papers*, pages 20–35, 2015. (Cited on page 12.)

[Sch89] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, pages 239–252, 1989. (Cited on page 4, 5.)

[Seg18] Bitcoin Improvement Proposals for Segwit. `https://github.com/bitcoin/bips`, 2018. (Cited on page 29.)

[Seg19] Segregated Witness Wallet Development Guide. `https://bitcoincore.org/en/segwit_wallet_dev/`, 2019. (Cited on page 31.)

[Ske18] Rhys Skellern. Cryptocurrency Hacks: More Than $2b USD lost between 2011-2018. `https://medium.com/ecomi/cryptocurrency-hacks-more-than-2b-usd-lost-between-2011-2018_-67054b342219`, 2018. (Cited on page 1.)

```
main $\mathbf{uf\text{-}cma\text{-}rk}_{\mathsf{RSig}}$
$(sk, pk) \xleftarrow{\$} \mathsf{RSig.Gen}\,(par)$
$(m^*, \sigma^*, \rho^*) \xleftarrow{\$} \mathsf{C}^{\mathtt{RSign}}\,(pk)$
If $m^* \in Sigs$: $\mathsf{bad} \leftarrow \mathsf{true}$
$pk^* \leftarrow \mathsf{RSig.RandPK}(pk, \rho^*)$
$b \leftarrow \mathsf{RSig.Verify}\,(pk^*, \sigma^*, m^*)$
Return $b \wedge \neg\mathsf{bad}$

Oracle $\mathtt{RSign}\,(m, \rho)$
$sk' \leftarrow \mathsf{RSig.RandSK}(sk, \rho)$
$\sigma \xleftarrow{\$} \mathsf{RSig.Sign}\,(m, sk')$
$Sigs \leftarrow Sigs \cup \{m\}$
Return $\sigma$
```

Figure 15: Security game $\mathbf{uf\text{-}cma\text{-}rk}_{\mathsf{RSig}}$ with adversary A.

[TVR16] Mathieu Turuani, Thomas Voegtlin, and Michael Rusinowitch. Automated verification of electrum wallet. In *Financial Cryptography and Data Security - FC 2016 International Workshops, BITCOIN, VOTING, and WAHC*, pages 27–42, 2016. (Cited on page 5.)

[Wik18a] Bitcoin Wiki. BIP32 proposal. https://en.bitcoin.it/wiki/BIP_0032, 2018. (Cited on page 2.)

[Wik18b] Wikipedia. Hardware Wallet. https://en.bitcoin.it/wiki/Hardware_wallet, 2018. (Cited on page 2.)

[Wik19] Wikipedia. ECDSA Signature Scheme. https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm, 2019. (Cited on page 28.)

[Wui17] Pieter Wuille. Bitcoin Improvement Proposal 62. https://github.com/bitcoin/bips/blob/master/bip-0062.mediawiki, 2017. (Cited on page 29.)

[ZCC+15] Zongyang Zhang, Yu Chen, Sherman S. M. Chow, Goichiro Hanaoka, Zhenfu Cao, and Yunlei Zhao. Black-box separations of hash-and-sign signatures in the non-programmable random oracle model. In *Provable Security - 9th International Conference, ProvSec 2015*, pages 435–454, 2015. (Cited on page 5.)

A A Construction from BLS

Recall that the notion of unforgeability under honestly rerandomizable keys introduced in Section 2 is a weaker form of the notion of unforgeability under rerandomizedkeys proposed in [FKM+16]. For a signature scheme with rerandomizable keys RSig, we present a formalization of later via game $\mathbf{uf\text{-}cma\text{-}rk}_{\mathsf{RSig}}$ in Figure 15. In this section we show that the BLS signature scheme achieves the notion of unforgeability under rerandomized keys. We give a formal proof via Theorem A.2 in the following subsection. For the Schnorr signature scheme, we note that the corresponding result follows from Theorem 1 in [FKM+16]. We begin by recalling the BLS signature scheme BLS presented in Figure 16. Here, we assume that $par = \mathcal{G}$ defines a group $\mathbb{G}$ of prime order p with generator g. To prove that $\mathsf{RBLS[H]}$ satisfies the notion of $\mathbf{uf\text{-}cma\text{-}rk}$, we again use a transformation algorithm (similar to the one used in the ECDSA based construction) that converts signatures under a public key pk into signatures under a related public key pk'. The BLS transformation algorithm is depicted in Figure 17 and its properties are summarized in the following lemma:

Lemma A.1 *Consider the algorithm* $\mathsf{Trf[H]}_{\mathsf{BLS}}$ *depicted in Figure 16. Suppose that:*

- $X_0 = g^{x_0}, X_1 = g^{x_1} \in \mathbb{G}$ *and* $\omega \in \mathbb{Z}_p$ *s.t.* $X_1 = X_0 g^{\omega} = g^{x_0+\omega}$,
- $\mathsf{BLS[H].Verify}(X_1, \sigma_1, m) = 1$,
- $\sigma_0 \leftarrow \mathsf{Trf[H]}_{\mathsf{BLS}}(m, \sigma_1, \omega, X_0, X_1)$.

Then $\mathsf{BLS[H].Verify}(\sigma_0, X_0, m) = 1$.

```
Algorithm $\mathsf{BLS[H].Gen}\,(par = \mathcal{G})$
$x \xleftarrow{\$} \mathbb{Z}_p$
$X \leftarrow g^x$
$sk \leftarrow x$
$pk \leftarrow X$
Return $(pk, sk)$

Algorithm $\mathsf{BLS[H].Sign}\,(sk = x, m)$
Return $\sigma := \mathsf{H}\,(m)^x$

Algorithm $\mathsf{BLS[H].Verify}(pk = X, \sigma, m)$
Return $(e\,(\sigma, g) = e\,(\mathsf{H}\,(m)\,, X))$

Algorithm $\mathsf{RBLS[H].RandPK}(pk = X, \rho)$
$pk' = X \cdot g^{\rho}$
Return $pk'$

Algorithm $\mathsf{RBLS[H].RandSK}(sk = x, \rho)$
$sk' = x + \rho$
Return $sk'$
```

Figure 16: $\mathsf{BLS}\,[\mathsf{H}] = (\mathsf{BLS[H].Gen}, \mathsf{BLS[H].Sign}, \mathsf{BLS[H].Verify})$: BLS Signature scheme relative to groups $\mathbb{G}, \mathbb{G}_T$ with bilinear mapping $e\colon \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$, where g is the generator of the group $\mathbb{G}$ and hash function $\mathsf{H}\colon \{0,1\}^* \to \mathbb{G}$. $\mathsf{RBLS[H]} = (\mathsf{BLS}\,[\mathsf{H}]\,, \mathsf{RBLS[H].RandSK}, \mathsf{RBLS[H].RandPK})$: BLS signature scheme with rerandomization routines for secret and public keys.

```
$\mathsf{Trf[H]_{BLS}}\,(m, \sigma_1, \omega, X_0, X_1)$
If $(\mathsf{BLS[H].Verify}\,(\sigma_1, X_1, m) = 0) \vee (X_1 \neq X_0 \cdot g^{\omega})$ :
    Return $\perp$
$h \leftarrow \mathsf{H}(m)$
$\sigma_0 \leftarrow \sigma_1 \cdot h^{-\omega}$
Return $\sigma_0$
```

Figure 17: Transformation algorithm $\mathsf{Trf[H]_{BLS}}$ with hash function $\mathsf{H}\colon \{0,1\}^* \to \mathbb{G}$.

Proof. From the prerequisite of the lemma, we have that $X_1 = X_0 g^{\omega} = g^{x_0+\omega}$ and $\mathsf{BLS[H].Verify}(X_1, \sigma_1, , m) = 1$, which implies that both $\sigma_1 = \mathsf{H}(m)^{x_1} = \mathsf{H}(m)^{x_0+\omega}$ and $\mathsf{Trf[H]_{BLS}}(m, \sigma_1, \omega, X_0, X_1) \neq \perp$. $\mathsf{Trf[H]_{BLS}}$ now computes and returns $\sigma_0 = \sigma_1 \cdot \mathsf{H}(m)^{-\omega}$. Since $\sigma_0 = \mathsf{H}(m)^{x_0+\omega} \cdot \mathsf{H}(m)^{-\omega} = \mathsf{H}(m)^{x_0}$ is the unique signature on message m under public key X_0, it follows that $\mathsf{BLS[H].Verify}(X_0, \sigma_0, m) = 1$. ■

Theorem A.2 *Let $\mathsf{H}\colon \{0,1\}^* \to \mathbb{Z}_p$ be a hash function (modeled as a random oracle). Let A be an algorithm that plays in game $\mathbf{uf\text{-}cma\text{-}rk}_{\mathsf{RBLS[H]}}$. Then there exists an algorithm C running in roughly the same time as A, such that*

$$\mathsf{Adv}^{\mathsf{A}}_{\mathbf{uf\text{-}cma\text{-}rk},\mathsf{RBLS[H]}} \leq \mathsf{Adv}^{\mathsf{C}}_{\mathbf{uf\text{-}cma},\mathsf{BLS[H]}},$$

where q is the number of random oracle queries that A makes.

Proof. Consider an adversary A playing in game $\mathbf{uf\text{-}cma\text{-}rk}_{\mathsf{RBLS[H]}}$. As such, A is given an input public key $pk = X$, and is granted access to the oracle `RSign` and the random oracle H. We prove Theorem A.2 via the following reduction.

Reduction to UF-CMA Security. We describe an algorithm C (depicted in Figure 18) that plays in the $\mathbf{uf\text{-}cma}_{\mathsf{BLS[H]}}$ game. C obtains as input a public key pk_{C} and is given access to the signing oracle `SignO` to obtain signatures under pk_{C} under messages of its choice. Furthermore, C has access to the random oracle H. C runs A on input pk_{C} and simulates $\mathbf{uf\text{-}cma\text{-}rk}_{\mathsf{RBLS[H]}}$ to A as described in Figure 18.

Simulation of Signing Queries. Note that C does not know $sk' = \mathsf{RBLS[H].RandSK}(sk, \rho)$. However, C can use the signing oracle `SignO` in the $\mathbf{uf\text{-}cma}_{\mathsf{BLS[H]}}$ game to obtain signatures on a message m of C's choice under pk_{C}. Subsequently, C can use $\mathsf{Trf[H]_{BLS}}$ to convert the so-obtained signature σ into a signature σ' (also on m) under $pk' = pk \cdot g^{\rho}$, i.e, $\sigma' \leftarrow \mathsf{Trf[H]_{BLS}}(m, \sigma, \rho, pk', pk)$. By lemma A.1, σ' is a valid signature under pk'.

Extracting the forgery. When A returns tuple (m^*, σ^*, ρ^*), C derives the rerandomized key $pk^* = g^{pk+\rho^*}$ and checks whether (m^*, σ^*, ρ^*) is a valid forgery under pk^*. If yes, then C derives σ under pk_{C} as $\sigma \leftarrow \mathsf{Trf[H]_{BLS}}(m^*, \sigma^*, \rho^*, pk_{\mathsf{C}}, pk^*)$. σ is a valid forgery under pk_{C} in game $\mathbf{uf\text{-}cma\text{-}rk}_{\mathsf{BLS[H]}}$ since by

Lemma A.1, $\mathsf{BLS}[\mathsf{H}].\mathsf{Verify}(pk^*, m^*) = \mathsf{RBLS}[\mathsf{H}].\mathsf{Verify}(pk^*, m^*) = 1$ implies that $\mathsf{BLS}[\mathsf{H}].\mathsf{Verify}(pk_{\mathsf{C}}, m^*) = 1$ and moreover, $m^* \notin \mathit{Sigs}$ (because m^* is a valid forgery in $\textbf{uf-cma-rk}_{\mathsf{RBLS[H]}}$).

```
$\mathsf{C}^{\mathsf{Sign},\mathsf{H}}(pk_{\mathsf{C}})$
$\mathsf{bad} \leftarrow \mathsf{false}$
$(m^*, \sigma^*, \rho^*) \xleftarrow{\$} \mathsf{C}^{\mathsf{H},\mathtt{RSign}}(pk_{\mathsf{C}})$
$pk^* \leftarrow pk_{\mathsf{C}} \cdot g^{\rho^*}$
If $m^* \in \mathit{Sigs}$: $\mathsf{bad} \leftarrow \mathsf{true}$
$b \leftarrow \mathsf{RBLS}[\mathsf{H}].\mathsf{Verify}(pk^*, \sigma^*, m^*)$
If $\neg b \vee \mathsf{bad}$ : Abort
$\sigma \leftarrow \mathsf{Trf}[\mathsf{H}]_{\mathsf{BLS}}(m^*, \sigma^*, \rho^*, pk_{\mathsf{C}}, pk^*)$
Return $(m^*, \sigma)$

Oracle $\mathtt{RSign}(m, \rho)$
$pk' \leftarrow pk_{\mathsf{C}} \cdot g^{\rho}$
$\sigma \leftarrow \mathtt{SignO}(m)$
$\sigma' \leftarrow \mathsf{Trf}[\mathsf{H}]_{\mathsf{BLS}}(m, \sigma, \rho, pk', pk_{\mathsf{C}})$
$\mathit{Sigs} \leftarrow \mathit{Sigs} \cup \{m\}$
Return $\sigma'$
```

Figure 18: Reduction to **uf-cma** Game

■

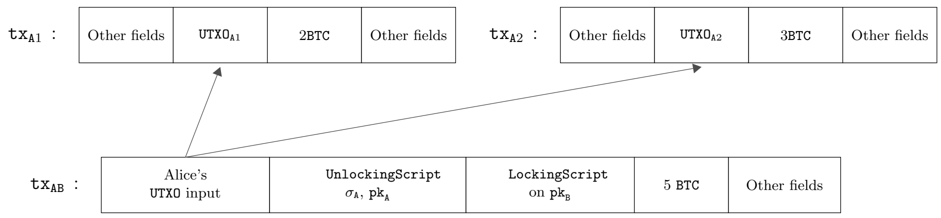
B The mechanics of Bitcoin

In this section, we discuss the underpinnings of the money mechanism in Bitcoin. The currency unit in Bitcoin is denoted as BTC. When a user Alice with key pair ($pk_{\mathtt{A}}$, $sk_{\mathtt{A}}$) wants to pay x amount of BTC to Bob having key pair ($pk_{\mathtt{B}}$, $sk_{\mathtt{B}}$), then it first needs to create a Bitcoin transaction. Let us denote this transaction as $\mathtt{tx}_{\mathtt{AB}}$. This transaction firstly includes information about Alice's payment in the input, secondly the destination address of Bob in the output, which essentially represents Bob's public key – $pk_{\mathtt{B}}$. After the transaction $\mathtt{tx}_{\mathtt{AB}}$ has been created, it is signed by Alice's secret key $sk_{\mathtt{A}}$ – as a result, a signature $\sigma_{\mathtt{A}}$ is generated. Once $\mathtt{tx}_{\mathtt{AB}}$ is propagated to the Bitcoin network, it will be validated by one of the mining nodes. The validation process essentially involves checking whether the signature $\sigma_{\mathtt{A}}$ provided by Alice is valid with respect to it's public key $pk_{\mathtt{A}}$. This signature generation, verification process in Bitcoin relies on the ECDSA signature scheme [Wik19]. Once $\mathtt{tx}_{\mathtt{AB}}$ qualifies as a valid transaction , it is included within a block. After a subsequent number of blocks, transaction $\mathtt{tx}_{\mathtt{AB}}$ gets confirmed in the Bitcoin network.

B.1 Payments over Bitcoin

In this subsection we want to take a closer look at how payments are done in Bitcoin via transactions. The majority of transactions in Bitcoin currently behaves as follows. The output of a Bitcoin transaction in its unspent form, is referred to as a UTXO – Unspent Transaction Output. A UTXO is analogous to the unspent money a user carries in its wallet. So a user can have \$46 cash in its wallet in the form of a combination of notes and coins - for example: two \$20 notes, one \$5 coin, and one \$1 coin. Similarly, in the cryptocurrency world, this user might possess 46 BTC in its Bitcoin wallet, in the form of a number of UTXO-s (for ex: $\mathtt{UTXO}_1 = 10\mathtt{BTC}$, $\mathtt{UTXO}_2 = 15\mathtt{BTC}$, $\mathtt{UTXO}_3 = 21\mathtt{BTC}$, so that $\mathtt{UTXO}_1 + \mathtt{UTXO}_2 + \mathtt{UTXO}_3 = 46\mathtt{BTC}$.). Likewise, when a user wants to pay via a Bitcoin transaction, then it is analogous to a regular cash payment in a shop. Suppose a user wants to buy bread worth \$4.65. the user may not have exactly \$4.65 in her wallet. Instead he gives a note of \$5 to the shop, out of which \$4.65 is spent for the purchase, while \$0.35 is returned to the user. In a similar way, a bitcoin transaction consists of an input part - specifying the UTXOs, the user wants to spend (analogous with the \$5 in the previous example) and a output part – specifying the newly created UTXOs to be paid to the recipient (analogous with the \$4.65 and \$.35 in the previous example). As is evident from the example above, both the input and output parts may contain more than one UTXO. In fact, the output field can be modified to create a more complicated transaction, or better include some important functionality. Before going into this direction of modifying an output field and its benefits, we first give details on the format of a transaction next.

Format of transactions. Here, we give a detailed overview on the important fields of a Bitcoin transaction with an illustration. Suppose Alice wants to use 5 BTC from her Bitcoin wallet to pay Bob. Henceforth Alice uses two UTXO-s from her wallet, where $\mathtt{UTXO}_{\mathtt{A1}} = 2\mathtt{BTC}$, $\mathtt{UTXO}_{\mathtt{A2}} = 3\mathtt{BTC}$. To pay Bob,



$\mathtt{tx}_{\mathtt{A1}}$:
Other fields
$\mathtt{UTXO}_{\mathtt{A1}}$
2BTC
Other fields
$\mathtt{tx}_{\mathtt{A2}}$:
Other fields
$\mathtt{UTXO}_{\mathtt{A2}}$
3BTC
Other fields
$\mathtt{tx}_{\mathtt{AB}}$:
Alice's UTXO input
UnlockingScript $\sigma_{\mathtt{A}}$, $\mathtt{pk}_{\mathtt{A}}$
LockingScript on $\mathtt{pk}_{\mathtt{B}}$
5 BTC
Other fields

Figure 19: A payment of 5 BTC from Alice ($pk_{\mathtt{A}}$, $sk_{\mathtt{A}}$) to Bob ($pk_{\mathtt{B}}$, $sk_{\mathtt{B}}$) via $\mathtt{tx}_{\mathtt{AB}}$.

Alice creates a new transaction. Let us name this transaction as $\mathtt{tx}_{\mathtt{AB}}$ (details in Figure 19). The newly created transaction $\mathtt{tx}_{\mathtt{AB}}$ contains the following fields:

1. Input: The input field contains

   - The details of the UTXO*s* which will be spent to create the current transaction. In this example - $\mathtt{UTXO}_{\mathtt{A1}}$, $\mathtt{UTXO}_{\mathtt{A2}}$.
   - The Unlocking Script contains a) the signature of the owner of $\mathtt{tx}_{\mathtt{AB}}$, i.e. the signature of Alice, computed as $\sigma_{\mathtt{A}} := \mathsf{Sign}(m = \mathsf{H}(\mathtt{tx}_{\mathtt{AB}}), sk_{\mathtt{A}})$. b) The public key of Alice – $pk_{\mathtt{A}}$, later required for signature verification.

2. Output: The output field may contain a number of so-called Locking Scripts, each one corresponding to one of the output UTXO*s*. The role of each Locking Script is to contain the destination address along with some conditions which are later relevant during transaction validation process. When a Locking Script is run with its matching Unlocking Script, if the result evaluates to true, it implies the transaction is valid. In the above example, the output contains one Locking Script corresponding to Bob's public key $pk_{\mathtt{B}}$.

Locking Script. Depending on the format of a Bitcoin transaction, the Locking Script specification varies. We describe the Locking Script in two of the most popular Bitcoin transaction formats. PAY-TO-PUBKEY-HASH FORMAT (P2PKH): As the name hints, Pay-to-PubKey-Hash Script represents payment to the destination address, which is essentially the hash of public key of the recipient. In this particular format, the Locking Script denoted as ScriptPubKey is of the following form

$$\mathtt{OP\_DUP\ OP\_HASH160}\ \ < \mathtt{hash160(pubKey)} >\ \ \mathtt{OP\_EQUAL}$$
$$\mathtt{OP\_CHECKSIG}$$

where, pubKey = public key of the recipient, the rest are the operators in the underlying scripting language. The corresponding Unlocking Script is

$$\mathtt{ScriptSig} := < \mathtt{Sig\ \ pubKey} >$$

where, Sig denotes the signature with respect to pubKey. The two scripts – ScriptSig, ScriptPubKey are run back to back within a forth-like stack based programming language. The script executes from left to right, where any non-operator is pushed into the stack. When the cursor reaches an operator, then necessary inputs are popped from the stack, and evaluated to produce an output.

The execution of following Unlocking-Locking Script has been illustrated in Table 2. The cursor will scan the Script from left to right.

$$\begin{aligned}\mathtt{Script} = \mathtt{Sig\ \ pubKey\ \ OP\_DUP\ \ OP\_HASH160}\\ < \mathtt{hash160(pubKey)} > \ \ \mathtt{OP\_EQUAL\ \ OP\_CHECKSIG}\end{aligned}$$

SEGWIT FORMAT (P2WSH): The key distinction of the Seggregated witness format to the previous format is that, the Unlocking Script is moved to an entity called the witness which is not stored as part of the transaction. This enhances Bitcoin scalability [Seg18], prevents transaction malleability [Wui17], and has other benefits. Here the Locking Script or ScriptPubKey is computed as follows

Table 2: Executing a matching Unlocking-Locking Script in Bitcoin

Steps	Cursor reads	Stack
Step 1	Sig	
Step 2	pubKey	Sig
Step 3	OP_DUP	pubKey Sig
Step 4	OP_HASH160	pubKey pubKey Sig
Step 5	$<$ hash160(pubKey) $>$	hash160(pubKey) pubKey Sig
Step 6	OP_EQUAL	$<$ hash160(pubKey) $>$ hash160(pubKey) pubKey Sig
Step 7	OP_CHECKSIG	pubKey Sig
Step 8		0/1

1. Define witness Script.
2. Set scriptHash = hash of (witness Script).
3. Compute

   ScriptPubKey = OP_HASH160 hash160(scriptHash) OP_EQUAL.

The witness Script contains the witness data which is embedded in the hash. The run of the Locking Script along with the Unlocking Script follows same as before, where the transaction passes as valid only when the Script evaluates to true.

Problem of lacking randomization. As was mentioned above, the underlying signature scheme in Bitcoin is ECDSA. Unfortunately the Bitcoin wallet in practice using ECDSA is not provably secure in our model. However, as discussed in section 5, our construction of a Bitcoin wallet instantiated with ECDSA achieves the notion of **wunf** security. Our proof technique crucially relies on prefixing any message with a random salt (denoted as ψ) before signing it. In any cryptocurrency network, messages are essentially the hash of the the entire transaction. To randomize the message, henceforth the underlying transaction needs to be randomized. However, one of the problems in existing Bitcoin transaction formats discussed above is that currently all the fields are of some specific form and contain no randomness. Although the public key value pubKey should be generated from the PKDer algorithm within the wallet and should look random to the user, it is not an acceptable source of randomness, as the random salt must be chosen freshly for every newly signed transaction. Note that a public key pubKey on the other hand may show up in multiple transactions if the user deliberately or unknowingly provides the same destination address which is used in a previous transaction. We provide a proposal to solve the lack of randomness problem in a transaction in the next section.

B.2 Integrating our Wallet Solution in Bitcoin

The Locking Script or the ScriptPubKey field in a transaction contains a Bitcoin script which later needs to be executed with a matching Unlocking Script. However this Locking Script can support much more complicated code, which, e.g., allows for mutli-signature payments. It is also a key ingredient to support payment channels in Bitcoin [Lig18a], [Lig18b]. We propose to use the Locking Script to integrate the salting process. For the following Bitcoin transaction formats, the Locking Script can be modified in the following way.

Pay-to-PubKey-Hash format (P2PKH): The idea here is to add a random seed in the Locking Script, essentially drop the seed using operator OP_DROP, then continue evaluating the rest of the script.

This helps in randomizing the `Locking Script` field in the transaction. This would require modification of the `ScriptPubKey` as

$$\begin{aligned}\texttt{ScriptPubKey} = \psi\ &\texttt{OP\_DROP OP\_DUP OP\_HASH160}\\ &< \texttt{hash160(pubKey)} > \quad \texttt{OP\_EQUAL OP\_CHECKSIG}\end{aligned}$$

where, $\psi \xleftarrow{\$} \{0,1\}^\kappa$ is the randomness. Here, the length of the transaction would increase by κ bits.

Segwit format (`P2WSH`): Similarly, in case of the Segwit format, we propose to include randomness in the `witness Script`. The `witness Script` has a size limitation of 3600 bytes [Seg19], thus allowing enough space for including more involved commands, and subsequently hashes to a 32 bytes value – `scriptHash`. So unlike Pay-to-PubKey-Hash, Segwit allows the possibility of randomized transaction without blowing up the length of the transaction. The modified script will have the following form: $\texttt{witness}' := r$ `OP_DROP witness`, where again $\psi \xleftarrow{\$} \{0,1\}^\kappa$.

C Missing Proofs

Proof of Lemma 5.2

Proof. Let $\sigma_1 = (r, s_1)$ be a valid signature on m_1 relative to G and public key X_1, i.e., $\mathsf{EC[G].Verify}(X_1, \sigma_1, m_1) = 1$. We have to show that $\sigma_0 = (r, \frac{s_1}{\omega}) = \mathsf{Trf[H,G]_{EC}}(m_0, m_1, \sigma_1, \omega, X_0, X_1)$ is a valid signature on m_0 relative to H and public key X_0, i.e., $\mathsf{EC[H].Verify}(X_0, \sigma_0, m_0) = 1$. To this end, let $z_1 = \mathsf{G}(m_1)$ and suppose that s_1 was computed as $s_1 = \frac{z_1 + r\omega x}{t}$ for some $t \in \mathbb{Z}_p$. We show that $\mathsf{EC[H].Verify}(X_0, \sigma_0, m_0) = 1$. The algorithm $\mathsf{EC[H].Verify}$ on input (X_0, σ_0, m_0) first computes $w_0 = (s_0)^{-1} = \frac{\omega}{s_1} = \frac{\omega t}{z_1 + r\omega x} = \frac{\omega t}{\omega z_0 + r\omega x} = \frac{t}{z_0 + rx} = \frac{t}{\mathsf{H}(m_0) + rx}$, where the last equation follows, because $\mathsf{Trf[H,G]_{EC}}(m_0, m_1, \sigma_1, \omega, X_0, X_1)$ did not return $\perp$ (by the prerequisites of the lemma). Therefore, since $z_0 = z_1/\omega = \mathsf{G}(m_1)/\omega$, it must hold that $z_0 = \mathsf{H}(m_0)$.

$\mathsf{EC[H].Verify}$ next computes $u_{1,0} \equiv_p z_0 w_0 \equiv_p \mathsf{H}(m_0) w_0, u_{2,0} \equiv_p r w_0$ and

$$\begin{aligned} u_{1,0} \cdot G + u_{2,0} \cdot X_0 =& \mathsf{H}(m_0) w_0 \cdot G + r w_0 \cdot x \cdot G \\ =& \mathsf{H}(m_0) w_0 \cdot G + x r w_0 \cdot G \\ =& (w_0 (\mathsf{H}(m_0) + xr)) \cdot G \\ =& t \cdot G =: (e_x, e_y) \end{aligned} \tag{3}$$

To ensure that $\mathsf{EC[H].Verify}(X_0, \sigma_0, m_0) = 1$, it remains to show that $r \equiv_p e_x$, where r is the first component of the signature. To this end, consider the computation performed via $\mathsf{EC[G].Verify}(X_1, \sigma_1, m_1)$. First, the algorithm computes

$$w_1 = (s_1)^{-1} = \frac{t}{z_1 + r\omega x} = \frac{t}{\mathsf{G}(m_1) + \omega r x}.$$

Next it computes $u_{1,1} \equiv_p z_1 w_1 \equiv_p \mathsf{G}(m_1) w_1, u_{2,1} \equiv_p r w_1$,

$$\begin{aligned} u_{1,1} \cdot G + u_{2,1} \cdot X_1 =& \mathsf{G}(m_1) w_1 \cdot G + r w_1 \cdot x\omega \cdot G \\ =& \mathsf{G}(m_1) w_1 \cdot G + x\omega r w_1 \cdot G \\ =& (w_1 (\mathsf{G}(m_1) + x\omega r)) \cdot G \\ =& t \cdot G = (e_x, e_y), \end{aligned} \tag{4}$$

Therefore, since $\mathsf{EC[G].Verify}(X_1, \sigma_1, m_1) = 1$, we have that $r \equiv_p e_x$. It follows now that also $\mathsf{EC[H].Verify}(X_0, \sigma_0, m_0) = 1$. ■