A Formal Treatment of Deterministic Wallets

Poulami Das* Sebastian Faust† Julian Loss‡

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 lead to prominent examples where millions of dollars worth in cryptocurrency have been 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 provable 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) is 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, and changes of 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 $\mathtt{pk_A}$ to Bob's address $\mathtt{pk_B}$. To this end, she creates a transaction $\mathtt{tx_{AB}}$ that informally says: "Transfer 1 $\mathtt{BTC}$ from $\mathtt{pk_A}$ to $\mathtt{pk_B}$". To ensure that only Alice can send her coins to Bob, we require that $\mathtt{tx_{AB}}$ is accompanied by a valid signature of $\mathsf{H}(\mathtt{tx_{AB}})$. Since only the owner of the corresponding $\mathtt{sk_A}$ – here Alice – can produce a valid signature, control over $\mathtt{sk_A}$ implies full control over the funds assigned to $\mathtt{pk_A}$. 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, alone in 2018, 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 makes them often an easy target for attackers, e.g., by exploiting software vulnerabilities via malware or phishing. Thus, it is generally recommended to store only a small amount of cryptocurrency on a hot wallet, while large

*Technische Universität Darmstadt, Email: poulami.das@crisp-da.de
†Technische Universität Darmstadt, Email: sebastian.faust@cs.tu-darmstadt.de
‡Ruhr University Bochum, Email: julian.loss@rub.de

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

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 at a secure place.

A simple way to construct a hot/cold wallet is to generate a key pair $(\mathtt{pk}_{\mathsf{cold}}, \mathtt{sk}_{\mathsf{cold}})$ and store the secret key $\mathtt{sk}_{\mathsf{cold}}$ on the cold wallet, while the corresponding public key $\mathtt{pk}_{\mathsf{cold}}$ 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 $\mathtt{pk}_{\mathsf{cold}}$. 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 $\mathtt{pk}_{\mathsf{cold}}$, the cold wallet may collect over time a large amount of money. Moreover, all transactions are publicly recorded on the blockchain, and thus $\mathtt{pk}_{\mathsf{cold}}$ becomes an attractive target for an attack the next time when the wallet goes online (which will happen 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 $(\mathtt{sk}_1, \mathtt{pk}_1), \ldots, (\mathtt{sk}_\ell, \mathtt{pk}_\ell)$. Then, the ℓ public keys are sent to the hot wallet, while the corresponding ℓ secret keys $\mathtt{sk}_i$ 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* $\mathtt{msk}$ together with a matching *master public key* $\mathtt{mpk}$ 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 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 cannot be forged, and thus funds are safe. Second, an *unlinkability* property, which guarantees that public keys generated from the same master public key $\mathtt{mpk}$ 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). 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 $\mathtt{msk}$ is stored on the cold wallet, while the corresponding master public key $\mathtt{mpk}$ 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 $\mathtt{ID}$, the hot wallet computes $w \leftarrow \mathsf{H}(ch, \mathtt{ID}), \mathtt{pk}_{\mathtt{ID}} \leftarrow mpk + w \cdot G$ and the cold wallet computes the corresponding session secret key as $w \leftarrow \mathsf{H}(ch, \mathtt{ID}), \mathtt{sk}_{\mathtt{ID}} \leftarrow msk + w$. As argued, e.g., in [MB18], this construction

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

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 $\mathtt{mpk}$ and ch the corresponding session public key $\mathtt{pk}_{\mathtt{ID}}$ for any $\mathtt{ID}$ of its choice. Even worse as we discuss in Section 4.2 (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. 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 shall 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 $\mathsf{swal} = (\mathsf{MGen}, \mathsf{SKDer}, \mathsf{PKDer}, \mathsf{WSign}, \mathsf{WVerify})$ be a wallet scheme, where MGen denotes the master key generation algorithm, $(\mathsf{SKDer}, \mathsf{PKDer})$ are used for deriving session keys and $(\mathsf{WSign}, \mathsf{WVerify})$ represent the signing and verification algorithm 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 $\mathtt{pk}_{\mathtt{ID}} \leftarrow \mathsf{PKDer}(\mathtt{mpk}, ch, \mathtt{ID})$ derived *prior* to the hot wallet breach, can not be linked with $\mathtt{mpk}$. Notably, our unlinkability property even holds against an adversary that sees a polynomial number of session public keys generated from $\mathtt{mpk}$ and signatures for adversarial 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 overcome 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 $\mathtt{mpk}$). This hinders a straightforward reduction to the security of the underlying signature scheme used by the cryptocurrency. Even worse we show that for certain naive instantiations of our wallet scheme 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 notion 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 $\mathtt{St}$ that is (deterministically) updated for every new session key pair. More concretely, the master key generation algorithm MGen outputs together with the master key pair $(\mathtt{mpk}, \mathtt{msk})$ an initial state $\mathtt{St}_0$ that will be stored on the hot and cold wallet, respectively. Then, to derive new session keys, the secret/public key derivation algorithms SKDer and PKDer take as input additionally the current state $\mathtt{St}_{i-1}$ and output the new state $\mathtt{St}_i$, while the old state $\mathtt{St}_{i-1}$ is erased from the hot/cold wallet. The update mechanism for deriving the new state has to guarantee that $\mathtt{St}_i$ looks random even if future states $\mathtt{St}_j$ (for $j > i$) gets revealed. Together with a mechanism to deriving new session key pairs, our scheme achieves the strong notion of forward unlinkability.

Modular approach for provably secure wallets. To securely instantiate our stateful deterministic wallets, we provide a modular approach that uses DLog-type signature schemes that exhibit an efficient homomorphism between the secret key and public key space. Concretely, for DLog-type schemes we use the simple homomorphism $F : \mathbb{Z}_p \to \mathbb{G}$, where $\mathbb{G}$ is a cyclic group of prime order p and $F(x) := g^x$ with g being a generator of $\mathbb{G}$. If the underlying cryptocurrency uses a signature scheme with this property, we can build a stateful deterministic wallet scheme in the following way. Let $\mathtt{St}$ be the current state of the hot/cold wallet. The public key derivation algorithm $\mathsf{PKDer}(\mathtt{mpk}, \mathtt{St}, \mathtt{ID})$ first computes $(\omega_{\mathtt{ID}}, \mathtt{St}') = \mathsf{H}(\mathtt{St}, \mathtt{ID})$. Then, it derives the new session public key $\mathtt{pk}_{\mathtt{ID}}$ as $\mathtt{mpk} \cdot g^{\omega_{\mathtt{ID}}}$ and erases the old state $\mathtt{St}$. Correspondingly, the cold wallet can compute $\mathtt{sk}_{\mathtt{ID}}$ as $\mathtt{msk} + \omega_{\mathtt{ID}}$, where the additive operation is in $\mathbb{Z}_p$. If H is modeled as a random oracle, then the above is a secure state update mechanism (i.e., it

satisfies the uniformity property mentioned above). Furthermore, this 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. It is well known that popular DLog-type signature schemes such as the Schnorr signatures [Sch89] and BLS [BLS04] exhibit such an homomorphism F between secret and public key space, and thus can be used for instantiating our stateful wallet scheme.

PROVABLY SECURE ECDSA-BASED WALLETS. Our main technical contribution is to propose the first provable secure construction of stateful deterministic wallets that work together with ECDSA-based cryptocurrencies such as Bitcoin. To achieve this, we slightly deviate from the above presented DLog-based paradigm and make several subtle changes to the current way hot/cold wallets are built for Bitcoin. An important goal of our construction is that all changes that we introduce 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 the formal security analysis of the ECDSA-based wallet. It is well known that ECDSA is a rather "contrived" signature scheme, and in our analysis we have to combine several "tricks" including an involved programming strategy to establish a formal security argument.

Formally, we prove wallet unforgeability by a reduction to the security of the underlying ECDSA signature scheme. The main challenge for this reduction is that the adversary may break into the hot wallet thereby learning $\mathtt{mpk}$ and the current state $\mathtt{St}_i$. Once these values are revealed the session public keys $\mathtt{pk}_{\mathtt{ID}}$ that are generated via PKDer are known and related to $\mathtt{mpk}$; moreover, the particular relation between the session public keys and $\mathtt{mpk}$ is known to the adversary. While for ECDSA-based wallets we do not know how to turn this knowledge into an actual attack that enables an adversary to steal money, it significantly complicates the security proof. More precisely, in the reduction we need to embed the target public key $\mathtt{pk}^*$ of the underlying ECDSA signature scheme into the simulation of the wallet unforgeability game. Once $\mathtt{pk}^*$ has been embedded, the reduction may have to answer signing queries for *any* of the session public keys.[3] Unfortunately, for these session public keys neither the corresponding secret keys are known nor can the reduction answer these queries in using the underlying ECDSA signing oracle. Thus, a straightforward approach ceases to work.

To overcome this challenge, we use an efficient method that transfers ECDSA signatures wrt. $\mathtt{pk}^*$ to signatures wrt. a related public key. Such "transformation algorithms" are well known in the context of related key attacks against signature schemes such as Schnorr and DSA [MSM+15].[4] Our main technical contribution is to integrate this "ECDSA transformation algorithm" into the formal reduction. This turns out to be quite challenging and requires us to combine several tricks including a rather involved random oracle programming strategy to complete the proof.

PRACTICAL CONSIDERATIONS. As a final contribution, we explore the practical implications of our work. First, we show that a careless implementation of hot/cold wallets using an underlying signature scheme, e.g., Schnorr or BLS, may result into a severe security vulnerability if the hot wallet is compromised. This may be 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 $\mathtt{pk}_{\mathtt{ID}}$ and $\mathtt{pk}_{\mathtt{ID}'}$ can transform a signature $\sigma_{\mathtt{ID}}$ scheme under $\mathtt{pk}_{\mathtt{ID}}$ to a signature $\sigma_{\mathtt{ID}'}$ for $\mathtt{pk}_{\mathtt{ID}'}$. This may have severe consequences because once an adversary sees a signature $\sigma_{\mathtt{ID}}$ that transfers funds assigned to $\mathtt{pk}_{\mathtt{ID}}$, it can also transfer the funds held by $\mathtt{pk}_{\mathtt{ID}'}$.

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. Fortunately, Bitcoin supports a simple scripting language such that these changes can be integrated at very low additional costs. For ease of space most details on how to integrate our wallet into Bitcoin are moved to Appendix A.

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

[3] In practice, this may happen when the cold wallet sends money to some other address.

[4] While the attack for Schnorr may be carried out in practice, the attack against DSA/ECDSA is impractical as it requires to break the intractability assumption of the hash function that is used in the scheme.

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 cannot 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 ciphertext, 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 later 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 $\mathtt{pk}$, one can produce a signature with respect to a related key $\mathtt{pk}'$. 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. As discussed above using such a transformation for something "useful" is one of the main technical challenges of our work. The abstraction of this property that we introduce in Section 2.1 for DLog-based schemes (Schnorr and BLS) is inspired by the signature malleability property of [ZCC+15].

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 $(y_1, ...) \xleftarrow{\$} \mathsf{A}(x_1, ...)$ to denote that A returns output $(y_1, ...)$ when run on input $(x_1, ...)$. 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 usually write $\mathtt{par}$ to denote (probabilistically generated) public parameters that 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.

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 $\mathsf{H}(x)$.

main $\mathbf{uf\text{-}cma}_{\mathsf{Sig}}$	Oracle $\mathsf{Sign}(m)$
00 $(sk, pk) \leftarrow \mathsf{Gen}(\mathtt{par})$	06 $\sigma \leftarrow \mathsf{Sign}(m, sk)$
01 $\mathsf{bad} \leftarrow \mathsf{false}$	07 $\mathtt{Sigs} \leftarrow \mathtt{Sigs} \cup \{m\}$
02 $(m^*, \sigma^*) \leftarrow \mathsf{C}^{\mathsf{Sign}}(pk)$	08 Return σ
03 If $m^* \in \mathtt{Sigs}$ set $\mathsf{bad} = \mathsf{true}$	
04 $b' \leftarrow \mathsf{Verify}(m^*, pk^*, \sigma^*)$	
05 Return $b' \wedge \neg \mathsf{bad}$	

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

ELLIPTIC CURVE CRYPTOGRAPHY. We denote an elliptic curve group as $\mathbb{E} = \mathbb{E}(\mathtt{par})$ with order $\mathcal{N}$. The base point of the group $\mathbb{E}$ is denoted as $G := (x_b, y_b)$. Any point $P := (x_p, y_p)$ in the group $\mathbb{E}$ can be written as $P = aG$, where $a \in [\mathcal{N}]$ and we use additive notation.

2.1 Definitions and Building Blocks

Before introducing our wallet model, we first go through some definitions and building blocks which are essential in formally presenting our model.

Definition 2.1 (Signature Scheme). A signature scheme Sig is a triple of algorithms $\mathsf{Sig} = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$. The randomized key generation algorithm Gen takes as input public parameters $\mathtt{par}$ and returns a pair $(pk,\ sk)$, of public and secret keys. The randomized signing algorithm Sign takes as input a secret key sk and a message m and returns a signature σ. The deterministic verification algorithm 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*:

$$\forall (pk, sk) \in \mathsf{Gen}(\mathtt{par}) \ \ \forall m\colon \mathsf{Verify}(pk, \mathsf{Sign}(sk, m), m) = 1.$$

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 via the **uf-cma** game (Figure 1). In this game the adversary is given the public key of a signature scheme and can adaptively sign messages of its choice under the corresponding secret key via an oracle Sign. It wins if it can produce a signature (forgery) on a fresh message m^* that was not previously queried to Sign. For a signature scheme $\mathsf{Sig} = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ and an algorithm C, we define C's advantage in the $\mathbf{uf\text{-}cma}_{\mathsf{Sig}}$ game as $\mathsf{Adv}^{\mathsf{C}}_{\mathbf{uf\text{-}cma},\mathsf{Sig}} = \Pr\left[\mathbf{uf\text{-}cma}^{\mathsf{C}}_{\mathsf{Sig}} = 1\right]$.

Definition 2.2 (DL-Type Signature Scheme). A *DL-Type Signature Scheme* $\mathsf{Sig} = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ is a signature scheme where the key generation algorithm Gen has the following special property. Gen takes as input parameters $\mathtt{par}$ which define a cyclic group $\mathbb{G} = \mathbb{G}(par)$ of prime order p with generator g. It samples $\mathtt{sk} \xleftarrow{\$} \mathbb{Z}_p$, and sets $\mathtt{pk} := g^{\mathtt{sk}}$. It returns $(\mathtt{sk}, \mathtt{pk})$.

We present some examples of well-known DL-type signature schemes which will be used throughout this work. In the following, let $\mathbb{G} = \mathbb{G}(\mathtt{par}), \mathbb{G}_T = \mathbb{G}_T(\mathtt{par})$ be cyclic groups of prime order p with generator g. Similarly, let $\mathbb{E} = \mathbb{E}(\mathtt{par})$ be an elliptic curve of prime order with base point G. Furthermore, let $e\colon \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$ denote a bilinear mapping. We present the ECDSA, Schnorr, and BLS schmes in Figures 5, 2, and 3, respectively.

RELATED KEY DERIVABLE SIGNATURE SCHEMES. Next, we introduce the notion of *related key derivable* signature schemes, which will be an essential tool in proving security of various wallet constructions in Section 4, Section 5. In the following, let again $\mathbb{G}$ be a cyclic group of prime order p with generator g associated with some DL-type signature scheme Sig. Informally, a DL-type signature scheme Sig is called related key derivable if there exists an efficient algorithm $\mathsf{Trf}_{\mathsf{Sig}}$ that converts signatures under a public key $X \in \mathbb{G}$ into a valid signature under a related key Xg^{ω} (here, $\omega \in \mathbb{Z}_p$). We focus here on signature schemes that internally use a hash function G, which we make explicit as $\mathsf{Sig}[\mathsf{G}]$

Definition 2.3 Let $\mathsf{Sig}[\mathsf{G}] = \left(\mathsf{Gen}, \mathsf{Sign}^{\mathsf{G}}, \mathsf{Verify}^{\mathsf{G}}\right)$ be a DL-type signature scheme. $\mathsf{Sig}[\mathsf{G}]$ is called *related key derivable* if there exists a deterministic algorithm $\mathsf{Trf}^{\mathsf{G}}_{\mathsf{Sig}}$ with the following behavior.

$\mathsf{SchnGen}^{\mathsf{H}}(par)$	$\mathsf{SchnSign}^{\mathsf{H}}(\mathtt{sk}=x,m)$	$\mathsf{SchnVer}^{\mathsf{H}}(\mathtt{pk}=X,\sigma,m)$
00 $x \leftarrow \mathbb{Z}_p$	05 $r \xleftarrow{\$} \mathbb{Z}_p$	10 Parse $(c,s) \leftarrow \sigma$
01 $X \leftarrow g^x$	06 $R \leftarrow g^r$	11 $c \equiv_p \mathsf{H}(m, g^s X^{-c})$
02 $sk \leftarrow x$	07 $c \leftarrow \mathsf{H}(m,R)$	
03 $pk \leftarrow X$	08 $s \leftarrow (r+xc) \mod p$	
04 Return (pk, sk)	09 Return $\sigma := (c,s)$	

Figure 2: $\mathsf{Schnorr}[\mathsf{H}] = \left(\mathsf{SchnGen}, \mathsf{SchnSign}^{\mathsf{H}}, \mathsf{SchnVer}^{\mathsf{H}}\right)$: Schnorr signature scheme relative to group $\mathbb{G}$ and the hash function $\mathsf{H}\colon \{0,1\}^* \to \mathbb{Z}_p$.

$\mathsf{BLSGen}^{\mathsf{H}}(par)$	$\mathsf{BLSSign}^{\mathsf{H}}(\mathtt{sk}=x,m)$	$\mathsf{BLSVer}^{\mathsf{H}}(\mathtt{pk}=X,\sigma,m)$
00 $x \leftarrow \mathbb{Z}_p$	05 Return $\sigma := H(m)^x$	06 Return $(e(\sigma,g) = e(H(m),X))$
01 $X \leftarrow g^x$		
02 $sk \leftarrow x$		
03 $pk \leftarrow X$		
04 Return (pk, sk)		

Figure 3: $\mathsf{BLS}[\mathsf{H}] = \left(\mathsf{BLSGen}, \mathsf{BLSSign}^{\mathsf{H}}, \mathsf{BLSVer}^{\mathsf{H}}\right)$: BLS Signature scheme relative to groups $\mathbb{G}, \mathbb{G}_T$ with bilinear mapping $e\colon \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$, and hash function $\mathsf{H}\colon \{0,1\}^* \to \mathbb{G}$.

- $\mathsf{Trf}^{\mathsf{G}}_{\mathsf{Sig}}$ takes as input a message $m \in \{0,1\}^*$, a signature σ_1, a scalar $\omega \in \mathcal{S}$, and public keys $X_0, X_1 \in \mathbb{G}$.
- If $X_1 \neq X_0 g^{\omega}$ or $\mathsf{Verify}(X_1, \sigma_1, m) = 0$, it returns $\perp$.
- Otherwise, $\mathsf{Trf}^{\mathsf{G}}_{\mathsf{Sig}}$ returns a signature σ_0 such that $\mathsf{Verify}^{\mathsf{G}}(X_0, \sigma_0, m) = 1$.
- $\mathsf{Trf}^{\mathsf{G}}_{\mathsf{Sig}}$ runs in roughly the same time as the maximum of the running times of $\mathsf{Sign}^{\mathsf{G}}, \mathsf{Verify}^{\mathsf{G}}$.

The notion of related key derivability is closely related to that of *related key attacks* (RKA). In its most simple form, a RKA consists of first observing a signature σ_1 on message m under some public key X_1 and then coming up with a valid signature σ_0 under a 'related key' X_0 on the same message. The Schnorr scheme was shown to be vulnerable to RKAs in [MSM+15], where the relation between X_0 and X_1 for the Schnorr scheme is as described above, i.e., $X_1 = X_0 g^{\omega}$. Clearly, the RKA on Schnorr directly leads to an efficient algorithm $\mathsf{Trf}_{\mathsf{Schnorr}}$ (according to Definition 2.3). We observe that there is also a simple RKA on the BLS scheme, which uses the same relation between the public keys X_0 and X_1 as the RKA against the Schnorr scheme and similarly leads to an efficient algorithm $\mathsf{Trf}_{\mathsf{BLS}}$. Therefore, both the Schnorr and BLS schemes are related key derivable. We describe $\mathsf{Trf}_{\mathsf{Schnorr}}$ and $\mathsf{Trf}_{\mathsf{BLS}}$ in Figure 4. We prove the correctness of $\mathsf{Trf}_{\mathsf{BLS}}$ in Lemma 2.4 (correctness of $\mathsf{Trf}_{\mathsf{Schnorr}}$ follows immediately from the RKA in [MSM+15]).

Lemma 2.4 *Consider the algorithm* $\mathsf{Trf}^{\mathsf{G}}_{\mathsf{BLS}}$ *depicted in Figure 4. Suppose that:*

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

Then $\mathsf{BLSVer}^{\mathsf{G}}(\sigma_0, X_0, m) = 1$.

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

$\mathsf{Trf}^{\mathsf{H}}_{\mathsf{Schnorr}}(m, \sigma_1, \omega, X_0, X_1)$	$\mathsf{Trf}^{\mathsf{G}}_{\mathsf{BLS}}(m, \sigma_1, \omega, X_0, X_1)$
00 If $\mathsf{Verify}^{\mathsf{H}}(\sigma_1, X_1, m) = 0 \vee X_1 \neq X_0 g^{\omega}$	06 If $\mathsf{Verify}^{\mathsf{G}}(\sigma_1, X_1, m) = 0 \vee X_1 \neq X_0 \cdot g^{\omega}$
01 Return $\perp$	07 Return $\perp$
02 $(c, s) \leftarrow \sigma_1$	08 $h \leftarrow \mathsf{G}(m)$
03 $s' \leftarrow s - \omega \cdot c \mod p$	09 $\sigma_\omega \leftarrow h^{-\omega}$
04 $\sigma_0 \leftarrow (c, s')$	10 $\sigma_0 \leftarrow \sigma_1 \cdot \sigma_\omega$
05 Return σ_0	11 Return σ_0

Figure 4: Behaviour of algorithms $\mathsf{Trf}^{\mathsf{G}}_{\mathsf{BLS}}$ and $\mathsf{Trf}^{\mathsf{H}}_{\mathsf{Schnorr}}$. The public keys $X_0, X_1 \in \mathbb{G}$ satisfy the relation $X_1 = X_0 g^{\omega}$. σ_1 is a signature on m under public key X_1 and σ_0 is a signature on m under public key X_0. As introduced above, $\mathsf{G}\colon \{0,1\}^* \rightarrow \mathbb{G}$ and $\mathsf{H}\colon \{0,1\}^* \rightarrow \mathbb{Z}_p$.

RKA of ECDSA. It is well-known that ECDSA signatures admit a related key attack [MSM+15]. In practice, it seems however impossible to exploit this attack since it requires to break certain intractability assumptions of hash functions. Informally speaking, this is because the RKA against ECDSA, requires an additional relation to hold between (the hash values of) the messages for which a signature can be forged. Although the RKA of ECDSA is impractical, we show that it can be utilized to instantiate in the random oracle model the related key derivability property mentioned above. While for Schnorr and BLS instantiating this property is straightforward, for ECDSA it gets slightly more complicated as we discuss below.

Related Key Derivability of ECDSA. The algorithm associated with the *related key derivability* of ECDSA is $\mathsf{Trf}_{\mathsf{ECDSA}}$ presented in Figure 5. 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 $\mathsf{Trf}_{\mathsf{ECDSA}}$ does not result into a practical RKA as both signatures σ_0 and σ_1 are valid with respect to different hash functions and the consistency checks in $\mathsf{Trf}_{\mathsf{ECDSA}}$ strongly restrict on what messages the related signature can be computed.[5] The following lemma formalizes the properties of $\mathsf{Trf}_{\mathsf{ECDSA}}$.

Lemma 2.5 *Consider the algorithm* $\mathsf{Trf}^{\mathsf{H,G}}_{\mathsf{ECDSA}}$ *in Figure 5. Suppose that:*

[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.

$\mathsf{ECDSAGen}^{\mathsf{H}}(par)$	$\mathsf{ECDSASign}^{\mathsf{H}}(\mathsf{sk} = x, m)$	$\mathsf{ECDSAVer}^{\mathsf{H}}(\mathsf{pk} = X, \sigma, m)$	$\mathsf{Trf}^{\mathsf{H,G}}_{\mathsf{ECDSA}}(m_0, m_1, \sigma_1, \omega, X_0, X_1)$
00 $x \leftarrow \mathbb{Z}_p$	05 $z \leftarrow \mathsf{H}(m)$	15 Parse $(r, s) \leftarrow \sigma$	00 $z_0 \leftarrow \mathsf{H}(m_0)$
01 $X \leftarrow x \cdot G$	06 $t \xleftarrow{\$} \mathbb{Z}_p$	16 If $(r, s) \notin \mathbb{Z}_p$	01 $z_1 \leftarrow \mathsf{G}(m_1)$
02 $sk \leftarrow x$	07 $(e_x, e_y) \leftarrow t \cdot G$	17 Return 0	02 If $\mathsf{Verify}^{\mathsf{G}}(\sigma_1, m_1, X_1) = 0 \vee \omega \neq \frac{z_1}{z_0} \vee X_1 \neq X_0 \cdot \omega$
03 $pk \leftarrow X$	08 $r \leftarrow e_x \mod p$	18 $w \leftarrow s^{-1}$	03 Return $\perp$
04 Return (pk, sk)	09 If $r \equiv_p 0$	19 $z \leftarrow \mathsf{H}(m)$	04 $(r, s_1) \leftarrow \sigma_1$
	10 Goto Step 3	20 $u_1 \leftarrow zw \mod p$	05 $s_0 \leftarrow \frac{s_1}{\omega}$
	11 $s \leftarrow t^{-1}(z + rx)$	21 $u_2 \leftarrow rw \mod p$	06 $\sigma_0 \leftarrow (r, s_0)$
	12 If $s \equiv_p 0$	22 $(e_x, e_y) \leftarrow u_1 \cdot G + u_2 \cdot X$	07 Return σ_0
	13 Goto Step 3	23 If $(e_x, e_y) = (0, 0)$	
	14 Return $\sigma := (r, s)$	24 Return 0	
		25 Return $r \equiv_p e_x$	

Figure 5: $\mathsf{ECDSA}[\mathsf{H}] = \left(\mathsf{ECDSAGen}, \mathsf{ECDSASign}^{\mathsf{H}}, \mathsf{ECDSAVer}^{\mathsf{H}}\right)$: ECDSA Signature scheme relative to elliptic curve $\mathbb{E}$ and hash function $\mathsf{H}\colon \{0,1\}^* \rightarrow \mathbb{Z}_p$. The right hand side of the figure shows the $\mathsf{Trf}_{\mathsf{ECDSA}}$ algorithm where $\mathsf{G}\colon \{0,1\}^* \rightarrow \mathbb{Z}_p$.

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

Then $\mathsf{ECDSAVer}^{\mathsf{H}}(X_0, \sigma_0, m_0) = 1$.

Proof. Let $\sigma_1 = (r, s_1)$ be a valid signature on m_1 relative to G and public key X_1, i.e., $\mathsf{ECDSAVer}^{\mathsf{G}}(X_1, \sigma_1, m_1) = 1$. We have to show that $\sigma_0 = (r, \frac{s_1}{\omega}) = \mathsf{Trf}^{\mathsf{H,G}}_{\mathsf{ECDSA}}(m_0, m_1, \sigma_1, \omega, X_0, X_1)$ is a valid signature on m_0 relative to H and public key X_0, i.e., $\mathsf{ECDSAVer}^{\mathsf{H}}(X_0, \sigma_0, m_0) = 1$. 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 $\mathsf{ECDSAVer}^{\mathsf{H}}(X_0, \sigma_0, m_0) = 1$. The algorithm $\mathsf{ECDSAVer}^{\mathsf{H}}$ 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 $\mathsf{Trf}^{\mathsf{H,G}}_{\mathsf{ECDSA}}(m_0, m_1, \sigma_1, \omega, X_0, X_1)$ 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 $z_0 = \mathsf{H}(m_0)$.

$\mathsf{ECDSAVer}^{\mathsf{H}}$ 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

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

To ensure that $\mathsf{ECDSAVer}^{\mathsf{H}}(X_0, \sigma_0, m_0) = 1$, it remains to show that $r \equiv_p e_x$, where r is the first component of the ECDSA signature. To this end, consider the computation performed via $\mathsf{ECDSAVer}^{\mathsf{G}}(X_1, \sigma_1, m_1)$. 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 $u_{1,1} \equiv_p z_1 w_1 \equiv_p \mathsf{G}(m_1) w_1$, $u_{2,1} \equiv_p r w_1$,

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

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

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 6, 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 knows a corresponding secret key and to keep track of the order they where created in, session keys also have an identifier $\mathtt{ID} \in \{0,1\}^*$ which is used as an argument for the key derivation procedures. We now proceed to give the syntax of a stateful wallet scheme.

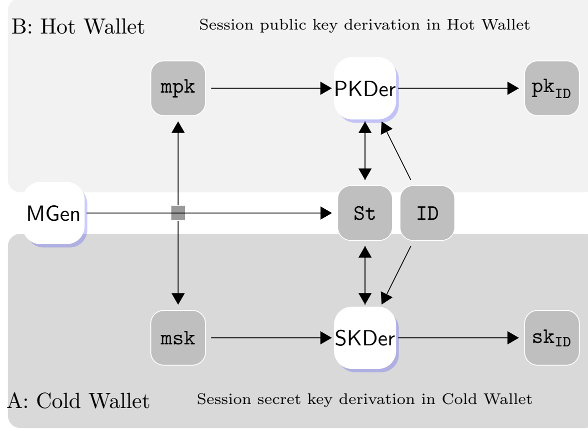

Figure 6: Both Hot/ Cold wallet internally stores the common state $\texttt{St}$. The master keys are stored in the respective wallets. When a session secret key is generated within the cold wallet as $(sk_{\texttt{ID}}, \texttt{St}) \leftarrow \mathsf{SKDer}(\texttt{msk}, \texttt{ID}, \texttt{St})$, the state $\texttt{St}$ gets refreshed. The session public key pk_{ID} is generated within the hot wallet as $(pk_{\texttt{ID}}, \texttt{St}) \leftarrow \mathsf{PKDer}(\texttt{mpk}, \texttt{ID}, \texttt{St})$, and the corresponding state $\texttt{St}$ is refreshed in the same manner.

Definition 3.1 (Stateful Wallet). A *stateful wallet* is a tuple of algorithms $\mathsf{swal} = (\mathsf{MGen}, \mathsf{SKDer}, \mathsf{PKDer}, \mathsf{WSign}, \mathsf{WVerify})$ defined as follows. The randomized *master key generation algorithm* $\mathsf{MGen}(\texttt{par})$ takes public parameters $\texttt{par}$ as input and outputs a tuple $(\texttt{St}_0, \texttt{mpk}, \texttt{msk})$ consisting of an *initial state* $\texttt{St}_0$, a *master public key* $\texttt{mpk}$ and a *master secret key* $\texttt{msk}$. The deterministic *secret key derivation algorithm* SKDer takes as input a master secret key $\texttt{msk}$, an identity $\texttt{ID}$, and a state $\texttt{St}$. It outputs a session secret key $\texttt{sk}_{\texttt{ID}}$ and an updated state $\texttt{St}'$. The deterministic *public key derivation algorithm* PKDer takes as input a master public key $\texttt{mpk}$, an identity $\texttt{ID}$, and a state $\texttt{St}$. It outputs a session public key $\texttt{pk}_{\texttt{ID}}$ and an updated state $\texttt{St}'$. The randomized signing algorithm WSign takes as input a (session) secret key sk and a message m and returns a signature σ. The deterministic verification algorithm $\mathsf{WVerify}$ takes as input a (session) public key pk, a signature σ, and a message m. It returns 1 (accept) or 0 (reject). The signing/ verification functions within a wallet are presented in Figure 7.

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 $\texttt{ID}_0, ..., \texttt{ID}_{n-1} \in \{0,1\}^*$ 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). Suppose that $(\texttt{St}_0, \texttt{msk}, \texttt{mpk}) \xleftarrow{\$} \mathsf{MGen}(\texttt{par})$ and let $\texttt{St}'_0 = \texttt{St}_0$. We say that $\mathsf{swal} = (\mathsf{MGen}, \mathsf{SKDer}, \mathsf{PKDer}, \mathsf{WSign}, \mathsf{WVerify})$ is *correct* if $\forall n \in \mathbb{N}$, $\texttt{ID}_0, ..., \texttt{ID}_{n-1} \in \{0,1\}^*$, and $\forall i \in [n]$, $(\texttt{St}_i, \texttt{sk}_{\texttt{ID}_{i-1}}) \leftarrow \mathsf{SKDer}(\texttt{msk}, \texttt{ID}_{i-1}, \texttt{St}_{i-1})$ and $(\texttt{St}'_i, \texttt{pk}_{\texttt{ID}_{i-1}}) \leftarrow \mathsf{PKDer}(\texttt{mpk}, \texttt{ID}_{i-1}, \texttt{St}'_{i-1})$ implies that:

- $\texttt{St}'_i = \texttt{St}_i$,
- $\forall m : \mathsf{WVerify}(\texttt{pk}_{\texttt{ID}_{i-1}}, \mathsf{WSign}(\texttt{sk}_{\texttt{ID}_{i-1}}, m), m) = 1$.

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. For the remainder of this section, let $\mathsf{swal} = (\mathsf{MGen}, \mathsf{SKDer}, \mathsf{PKDer}, \mathsf{WSign}, \mathsf{WVerify})$ denote a stateful wallet according to Definition 3.1.

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

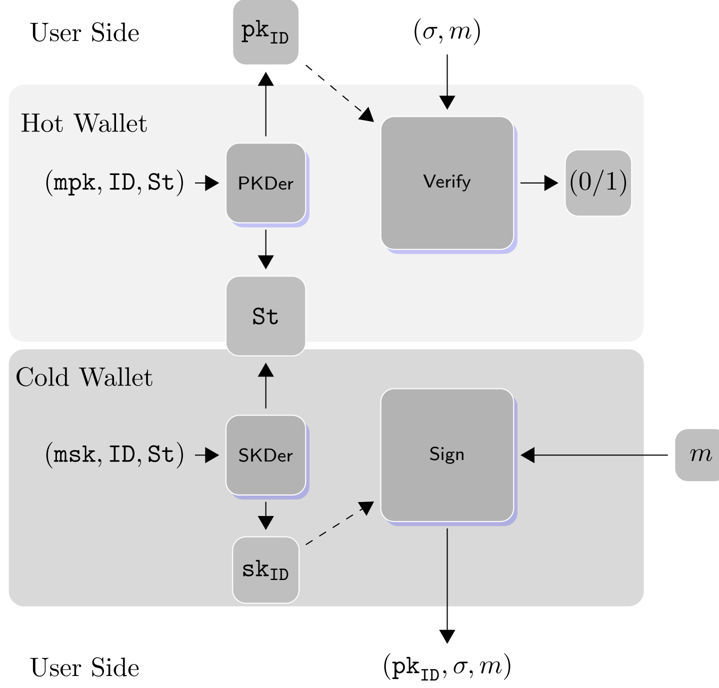

Figure 7: (1) The cold wallet signs a message m with its session secret key $sk_{\mathtt{ID}}$ as $\sigma \leftarrow \mathsf{Sign}(sk_{\mathtt{ID}}, m)$. (2) Anyone can later verify the validity of a signature σ on message m as $(0/1) \leftarrow \mathsf{Verify}(pk_{\mathtt{ID}}, \sigma, m)$.

public session keys is computationally indistinguishable from session keys that are generated from a fresh (independently chosen) master public key. Unfortunately, there is little hope to achieve this guarantee for keys for which the adversary knows the state $\mathtt{St}$ used to derive them. Therefore, our notion of unlinkability satisfies the form of *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 $\mathtt{mpk}$.

The wallet unlinkability game $\mathbf{unl}_{\mathsf{swal}}$ is presented in Figure 8. Initially, A receives as input a master public key $\mathtt{mpk}$ generated via $\mathsf{MGen}\,(\mathtt{par})$ and subsequently interacts with oracles PK, $\mathsf{WalSign}$ and Chall that reflect A's capabilities. The game internally maintains a state $\mathtt{St}$, 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 $\mathtt{St}$ 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 $\mathtt{pk_{ID}}$ for a user identity $\mathtt{ID}$ of its choice. This challenge public key is either "real" or "random", i.e., it depends on $\mathtt{mpk}$ or was sampled freshly and independently of $\mathtt{mpk}$ (see below for details). A's goal is to distinguish these two scenarios. However, A is only considered successful if it obtains $\mathtt{St}$ (via oracle getSt) *after* being given the challenge public key $\mathtt{pk_{ID}}$.[6] We now proceed in explaining the oracles to which A has access in more detail.

$\mathsf{PK}\,(\mathtt{ID})$: The oracle PK takes as input an $\mathtt{ID}$ and returns a corresponding session public key $\mathtt{pk_{ID}}$. It models A's capability to observe transactions stored on the blockchain that transfer money to $\mathtt{pk_{ID}}$. A typical setting where this may occur is when funds are sent via the blockchain to the cold wallet. We require wlog. (for simplicity of bookkeeping) that A can call PK only once per $\mathtt{ID}$.

$\mathsf{WalSign}(m, \mathtt{ID})$: The oracle $\mathsf{WalSign}$ takes as input an identity $\mathtt{ID}$ and a message m and returns the corresponding signature if $\mathtt{pk_{ID}}$ has been previously returned as a result from a $\mathsf{PK}\,(\mathtt{ID})$ 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. $\mathsf{WalSign}$ models that an adversary A may obtain signatures that are produced by the cold wallet with $\mathtt{sk_{ID}}$, when funds are spent from the cold wallet (e.g., when the owner of the cold wallet buys something with the collected coins).

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

main $\mathbf{unl}_{\mathsf{swal}}$
00 $(\mathtt{St}, \mathtt{msk}, \mathtt{mpk}) \xleftarrow{\$} \mathsf{MGen}(\mathtt{par})$
01 $b \xleftarrow{\$} \{0,1\}$ $\mathtt{Orc} \leftarrow \{\mathsf{PK}, \mathsf{WalSign}, \mathsf{Chall}, \mathsf{getSt}\}$
02 $b' \xleftarrow{\$} \mathsf{A}^{\mathtt{Orc}}(\mathtt{mpk})$
03 Return $b' = b$

Oracle $\mathsf{WalSign}(m, \mathtt{ID})$
04 If $\mathtt{SSNKeys}[\mathtt{ID}] = \bot$: Return $\bot$
05 $(\mathtt{pk}_{\mathtt{ID}}, \mathtt{sk}_{\mathtt{ID}}) \leftarrow \mathtt{SSNKeys}[\mathtt{ID}]$
06 $\sigma \xleftarrow{\$} \mathsf{WSign}(m, \mathtt{sk}_{\mathtt{ID}})$
07 Return σ

Oracle $\mathsf{PK}(\mathtt{ID})$ // *Once per* $\mathtt{ID}$
08 If $\mathtt{SSNKeys}[\mathtt{ID}] \neq \bot$
09 $\quad (\cdot, \mathtt{pk}_{\mathtt{ID}}) \leftarrow \mathtt{SSNKeys}[\mathtt{ID}]$
10 Else
11 $\quad \mathtt{St}' \leftarrow \mathtt{St}$
12 $\quad (\mathtt{sk}_{\mathtt{ID}}, \mathtt{St}) \leftarrow \mathsf{SKDer}(\mathtt{msk}, \mathtt{ID}, \mathtt{St}')$
13 $\quad (\mathtt{pk}_{\mathtt{ID}}, \mathtt{St}) \leftarrow \mathsf{PKDer}(\mathtt{mpk}, \mathtt{ID}, \mathtt{St}')$
14 $\quad \mathtt{SSNKeys}[\mathtt{ID}] \leftarrow (\mathtt{pk}_{\mathtt{ID}}, \mathtt{sk}_{\mathtt{ID}})$
15 Return $\mathtt{pk}_{\mathtt{ID}}$

Oracle $\mathsf{Chall}(\mathtt{ID})$ //One time
16 If $\mathtt{StateQuery}$: Return $\bot$
17 If $\mathtt{SSNKeys}[\mathtt{ID}] \neq \bot$: Return $\bot$

// *Generate real key*
18 $\mathtt{St}' \leftarrow \mathtt{St}$
19 $(\mathtt{sk}^0_{\mathtt{ID}}, \mathtt{St}) \leftarrow \mathsf{SKDer}(\mathtt{msk}, \mathtt{ID}, \mathtt{St}')$
20 $(\mathtt{pk}^0_{\mathtt{ID}}, \mathtt{St}) \leftarrow \mathsf{PKDer}(\mathtt{mpk}, \mathtt{ID}, \mathtt{St}')$

// *Generate random key*
21 $(\hat{\mathtt{St}}, \hat{\mathtt{msk}}, \hat{\mathtt{mpk}}) \xleftarrow{\$} \mathsf{MGen}(\mathtt{par})$
22 $(\mathtt{sk}^1_{\mathtt{ID}}, \cdot) \leftarrow \mathsf{SKDer}(\hat{\mathtt{msk}}, \mathtt{ID}, \hat{\mathtt{St}})$
23 $(\mathtt{pk}^1_{\mathtt{ID}}, \cdot) \leftarrow \mathsf{PKDer}(\hat{\mathtt{mpk}}, \mathtt{ID}, \hat{\mathtt{St}})$

24 $\mathtt{SSNKeys}[\mathtt{ID}] \leftarrow (\mathtt{pk}^b_{\mathtt{ID}}, \mathtt{sk}^b_{\mathtt{ID}})$
25 Return $\mathtt{pk}^b_{\mathtt{ID}}$

Oracle getSt
26 $\mathtt{StateQuery} \leftarrow \mathsf{true}$
27 Return $\mathtt{St}$

Figure 8: Adversary A playing in Game $\mathbf{unl}_{\mathsf{swal}}$.

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

Chall (ID): The oracle Chall takes as input an $\mathtt{ID}$ and returns a public key $\mathtt{pk}^b_{\mathtt{ID}}$ 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 $b = 0$, $\mathtt{pk}^0_{\mathtt{ID}}$ is derived from the current state $\mathtt{St}$ and $\mathtt{mpk}$ as $\mathtt{pk}^0_{\mathtt{ID}} \leftarrow \mathsf{PKDer}(\mathtt{mpk}, \mathtt{ID}, \mathtt{St})$. If $b = 1$, $\mathtt{pk}^1_{\mathtt{ID}}$ is derived from a freshly generated master public key and state for the same identity $\mathtt{ID}$, i.e., via the sequence of steps:

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

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

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

3.2 Wallet Unforgeability

In this subsection we describe the *wallet unforgeability* notion. In Game $\mathbf{wunf}^{\mathsf{A}}_{\mathsf{swal}}$ (depicted in Figure 9) we consider again an adversary A that receives as input a master public key $\mathtt{mpk}$ and has subsequently access to oracles PK and WalSign, which work as their corresponding oracles in the unlinkability game. In addition, A gets as input the *initial state* $\mathtt{St}$. A wins if it can produce a triple $(m^*, \sigma^*, \mathtt{ID}^*)$ such that σ^* is a valid forgery on message m^* under a public key $\mathtt{pk}_{\mathtt{ID}^*}$ previously obtained from a call to PK. Here, valid means that no signature on message m^* under $\mathtt{pk}_{\mathtt{ID}^*}$ was previously obtained from a call to WalSign. We denote A's advantage in $\mathbf{wunf}_{\mathsf{swal}}$ as

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

main $\mathbf{wunf}_{\mathsf{swal}}$

00 $(\mathtt{St}, \mathtt{msk}, \mathtt{mpk}) \xleftarrow{\$} \mathsf{MGen}(\mathtt{par})$
01 $(m^*, \sigma^*, \mathtt{ID}^*) \xleftarrow{\$} \mathsf{A}^{\mathsf{PK},\mathsf{WalSign}}(\mathtt{mpk}, \mathtt{St})$
02 If $\mathtt{SSNKeys}[\mathtt{ID}^*] = \bot$
03 Return 0
04 $(\mathtt{pk}_{\mathtt{ID}^*}, \mathtt{sk}_{\mathtt{ID}^*}) \leftarrow \mathtt{SSNKeys}[\mathtt{ID}^*]$
05 If $m^* \in \mathtt{Sigs}[\mathtt{ID}^*]$
06 Return 0
07 If $\mathsf{WVerify}(\mathtt{pk}_{\mathtt{ID}^*}, \sigma^*, m^*) = 0$
08 Return 0
09 Return 1

Oracle $\mathsf{WalSign}(m, \mathtt{ID})$

10 If $\mathtt{SSNKeys}[\mathtt{ID}] = \bot$: Return $\bot$
11 $(\mathtt{pk}_{\mathtt{ID}}, \mathtt{sk}_{\mathtt{ID}}) \leftarrow \mathtt{SSNKeys}[\mathtt{ID}]$
12 $\sigma \xleftarrow{\$} \mathsf{WSign}(\mathtt{sk}_{\mathtt{ID}}, m)$
13 $\mathtt{Sigs}[\mathtt{ID}] \leftarrow \mathtt{Sigs}[\mathtt{ID}] \cup \{m\}$
14 Return σ

Oracle $\mathsf{PK}(\mathtt{ID})$ // *Once per* $\mathtt{ID}$

15 $\mathtt{St}' \leftarrow \mathtt{St}$
16 $(\mathtt{sk}_{\mathtt{ID}}, \mathtt{St}) \leftarrow \mathsf{SKDer}(\mathtt{msk}, \mathtt{ID}, \mathtt{St}')$
17 $(\mathtt{pk}_{\mathtt{ID}}, \mathtt{St}) \leftarrow \mathsf{PKDer}(\mathtt{mpk}, \mathtt{ID}, \mathtt{St}')$
18 $\mathtt{SSNKeys}[\mathtt{ID}] \leftarrow (\mathtt{pk}_{\mathtt{ID}}, \mathtt{sk}_{\mathtt{ID}})$
19 $\mathtt{Sigs}[\mathtt{ID}] \leftarrow \emptyset$
20 Return $\mathtt{pk}_{\mathtt{ID}}$

Figure 9: Adversary A playing in Game $\mathbf{wunf}$.

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* $\mathtt{St}$ 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 $\mathtt{St}$ and the master public key $\mathtt{mpk}$, we also give him access to the PK and $\mathsf{WalSign}$ oracle. The first can be queried by the adversary on identity $\mathtt{ID}$ to derive a new key pair $(\mathtt{pk}_{\mathtt{ID}}, \mathtt{sk}_{\mathtt{ID}})$ from the master keys and the current state, and is used mainly for bookkeeping purpose.[7] The second oracle $\mathsf{WalSign}$ is as in the $\mathbf{unl}_{\mathsf{swal}}$ game except that we also keep track of the messages that were already signed via the map $\mathtt{Sigs}[\mathtt{ID}]$.

As already mentioned above, since the adversary receives $\mathtt{mpk}$ and the initial state $\mathtt{St}$ in the $\mathbf{wunf}_{\mathsf{swal}}$ game, it can derive all possible $\mathtt{pk}_{\mathtt{ID}}$ (even without calling $\mathsf{PK}(\mathtt{ID})$). 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 $\sigma_{\mathtt{ID}}$ with respect to $\mathtt{pk}_{\mathtt{ID}}$ to a valid signature $\sigma_{\mathtt{ID}^*}$ for $\mathtt{pk}_{\mathtt{ID}^*}$. 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 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 $\mathtt{mpk}, \mathtt{St}$. This break-in is modeled in $\mathbf{wunf}_{\mathsf{swal}}$ by giving the adversary $\mathtt{mpk}, \mathtt{St}$ 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 $\mathsf{WalSign}$ oracle to transfer some fraction of funds – say the funds stored under $\mathtt{pk}_{\mathtt{ID}}$ – 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 $\sigma_{\mathtt{ID}}$ 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 $\sigma_{\mathtt{ID}}$, the master public key $\mathtt{mpk}$ and $\mathtt{St}$ it can produce valid signatures $\sigma_{\mathtt{ID}^*}$ for $\mathtt{pk}_{\mathtt{ID}^*}$ where $\mathtt{pk}_{\mathtt{ID}^*}$ resulted from a previous call to PK on input $\mathtt{ID}^*$.

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 his funds to the cold wallet would assume that once the funds are transferred to the cold wallet,

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

```
DMGen(par)                                DSKDer^H(msk, ID, St)
00 St <-$ {0,1}^κ                         00 (ω_ID, St) <- H(St, ID)
01 (mpk, msk) <- Gen(par)                 01 sk_ID <- msk + ω_ID  mod p
02 Return (St, mpk, msk)                  02 Return (sk_ID, St)

DWSign^G(m, sk, pk)                       DPKDer^H(mpk, ID, St)
03 m' <- (pk, m)                          03 (ω_ID, St) <- H(St, ID)
04 σ <- Sign^G(sk, m')                    04 pk_ID <- mpk · g^ω_ID
05 Return σ                               05 Return (pk_ID, St)

DWVerify^G(pk, σ, m)
06 m' <- (pk, m)
07 Return Verify^G(pk, σ, m')
```

Figure 10: Construction of $\mathsf{swal}[\mathsf{G},\mathsf{H}]$ from DL-type signature scheme $\mathsf{Sig}[\mathsf{G}] = (\mathsf{Gen}, \mathsf{Sign}^{\mathsf{G}}, \mathsf{Verify}^{\mathsf{G}})$.

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 $\mathsf{Sign}(\mathtt{sk},(\mathtt{pk},m))$. 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 an ECDSA-based wallet scheme in the above unforgeability game to the security of the underlying ECDSA signature scheme.

4 Generic Construction from Related Key Derivable Signature Schemes

In this section, we show how to realize a stateful wallet from any DL-type signature scheme. 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 DL-type signature scheme which internally uses the hash function G, denoted as $\mathsf{Sig}[\mathsf{G}] = (\mathsf{Gen}, \mathsf{Sign}^{\mathsf{G}}, \mathsf{Verify}^{\mathsf{G}})$. Our construction $\mathsf{swal}[\mathsf{G},\mathsf{H}]$ of a stateful wallet which internally uses the hash functions $\mathsf{H}\colon \{0,1\}^* \to \mathbb{Z}_p \times \{0,1\}^\kappa$ (for state updates) and G (as part of $\mathsf{Sig}[\mathsf{G}]$) is depicted in Figure 10.

4.1 Unlinkability

We begin by proving unlinkability of our generic construction. The proof is rather simple and follows in a straight-forward manner from collision resistance of H and that H is modeled as a random oracle.

Theorem 4.1 *Let $\mathsf{swal}[\mathsf{G},\mathsf{H}]$ be the construction defined in Figure 10. Then for any adversary A playing in game $\mathbf{unl}_{\mathsf{swal}[\mathsf{G},\mathsf{H}]}$, we have*

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

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 $\mathbf{unl}_{\mathsf{swal}[\mathsf{G},\mathsf{H}]}$. A interacts with oracles $\mathsf{PK}, \mathsf{WalSign}, \mathsf{getSt}, \mathsf{Chall}$, and the random oracles G, H. We can assume without loss of generality that A always calls $\mathsf{Chall}(\mathtt{ID})$ *before* calling getSt and exclusively on an identity $\mathtt{ID}$ that was never previously queried to PK; otherwise,

$\mathsf{Adv}^{\mathsf{A}}_{\mathbf{unl},\mathsf{swal[G,H]}} = 0$ and the theorem holds trivially. In the following, let $\mathcal{S}$ denote the set of values taken by the variables $\mathtt{St}, \hat{\mathtt{St}}$ *before* A calls Chall (ID). Furthermore, let $\mathtt{pk}^0_{\mathtt{ID}}, \mathtt{pk}^1_{\mathtt{ID}}$ denote the keys internally sampled in $\mathbf{unl}_{\mathsf{swal[G,H]}}$ upon A's call Chall (ID). Note that by definition of DPKDer, unless A manages to make a query of the form $\mathsf{H}(\mathtt{St}', \mathtt{ID})$ where $\mathtt{St}' \in \mathcal{S}$, $\mathtt{pk}^0_{\mathtt{ID}}$ and $\mathtt{pk}^1_{\mathtt{ID}}$ are identically distributed from its point of view. In this case we again have that $\mathsf{Adv}^{\mathsf{A}}_{\mathbf{unl},\mathsf{swal[G,H]}} = 0$. It therefore remains to argue that A makes such a call to H with probability at most $(q_H(q_P+2))/2^\kappa$. This can be seen as follows. Since A makes at most q_P queries to PK throughout $\mathbf{unl}_{\mathsf{swal[G,H]}}$, in particular $|\mathcal{S}| \leq q_P + 2$. Since we have assumed that A always calls 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 $\mathtt{St}' \in \mathcal{S}$ (note that after such $\mathtt{St}'$ becomes known to A, it is able to infer all values that were added to $\mathcal{S}$ after $\mathtt{St}'$). Therefore, the probability that for any particular query of the form $\mathsf{H}(\mathtt{St}', \mathtt{ID})$, $\mathtt{St}' \in \mathcal{S}$, is at most $(q_P+2)/2^\kappa$. Since A makes at most q_H such queries of the form $\mathsf{H}(\mathtt{St}', \mathtt{ID})$, the probability that for any of them, $\mathtt{St}' \in \mathcal{S}$, is at most $(q_H(q_P+2))/2^\kappa$, which proves the lemma. ■

4.2 Unforgeability

We now turn towards the unforgeability of our construction. Before giving the proof, we provide some intuition about the main difficulties that arise and the techniques that we employ to overcome them. At a high level, the idea is to reduce the security of the stateful wallet scheme swal[G, H] (relative to $\mathbf{wunf}_{\mathsf{swal[G,H]}}$) to the security of Sig[G] (relative to $\mathbf{uf\text{-}cma}_{\mathsf{Sig[G]}}$). 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 $\mathbf{uf\text{-}cma}_{\mathsf{Sig[G]}}$ by simulating $\mathbf{wunf}_{\mathsf{swal[G,H]}}$ to an adversary A. C obtains a public key $\mathtt{pk}_{\mathsf{C}}$ from its challenger in $\mathbf{uf\text{-}cma}_{\mathsf{Sig[G]}}$ and can query a signing oracle Sign which provides signatures on messages of C's choice under $\mathtt{pk}_{\mathsf{C}}$. It also can query the random oracle G. C's goal is to simulate the oracles in the $\mathbf{wunf}_{\mathsf{swal[G,H]}}$ experiment and to suitably embed $\mathtt{pk}_{\mathsf{C}}$ into the key $\mathtt{pk}_{\mathtt{ID}^*}$ under which A eventually returns a forgery (σ^*, m^*). The hope is that it can use (σ^*, m^*) to win $\mathbf{uf\text{-}cma}_{\mathsf{Sig[G]}}$.

Strawman One. A first attempt towards this goal would be to embed $\mathtt{pk}_{\mathsf{C}}$ into a randomly chosen key that C returns as answer to a query to PK. More precisely, C chooses the master key pair msk, mpk in its simulation as $\mathtt{msk} \xleftarrow{\$} \mathbb{Z}_p, \mathtt{mpk} \leftarrow g^{\mathtt{msk}}$. Throughout the simulation, it keeps the variable St, which it updates every time it receives a query from A to oracle PK. To answer such a query of the form PK(ID), C simply derives the corresponding public key and secret key $\mathtt{pk}_{\mathtt{ID}}, \mathtt{sk}_{\mathtt{ID}}$ in an honest fashion, using its knowledge of msk and St. This ensures that for any public key $\mathtt{pk}_{\mathtt{ID}}$ of this type, C can easily answer subsequent signing queries. For (exactly one) randomly selected query $\mathsf{PK}(\mathtt{ID}^*)$, C returns, instead of the honestly generated $\mathtt{pk}_{\mathtt{ID}^*}$, the public key $\mathtt{pk}_{\mathsf{C}}$. When A asks a signing query of the form $\mathsf{WalSign}(m, \mathtt{ID}^*)$ (recall that C doesn't know $\mathtt{sk}_{\mathtt{ID}^*}$), C can answer this query as $\sigma \leftarrow \mathsf{Sign}(m)$, i.e., by using the signing oracle from $\mathbf{uf\text{-}cma}_{\mathsf{Sig[G]}}$. However, this approach fails for the following reason. Since A receives the initial state as part of its inputs, it knows the value of the variable St throughout the game. Therefore, when C outputs the public key $\mathtt{pk}_{\mathsf{C}}$ as an answer to the query $\mathsf{PK}(\mathtt{ID}^*)$, it is forced to program the random oracle H in such a way that $\mathsf{H}(\mathtt{St}, \mathtt{ID}^*)$ is consistent with the values of mpk and $\mathtt{pk}_{\mathsf{C}}$. Concretely, it must program H such that $(\omega_{\mathtt{ID}^*}, \cdot) = \mathsf{H}(\mathtt{St}, \mathtt{ID}^*)$ and $g^{\mathtt{msk}+\omega_{\mathtt{ID}^*}} = \mathtt{pk}_{\mathsf{C}}$, since otherwise, A can trivially distinguish C's simulation from a real execution of $\mathbf{wunf}_{\mathsf{swal[G,H]}}$. To do so, A first computes $(\omega'_{\mathtt{ID}^*}, \cdot) \leftarrow \mathsf{H}(\mathtt{St}, \mathtt{ID}^*)$. It can then check that $\mathtt{mpk} \cdot g^{\omega'_{\mathtt{ID}^*}} \neq \mathtt{pk}_{\mathsf{C}}$ to distinguish the simulation from the actual experiment. Because $\mathtt{pk}_{\mathsf{C}} = g^{\mathtt{sk}_{\mathsf{C}}}$, this programming strategy implies that $\mathtt{msk} + \omega_{\mathtt{ID}^*} \equiv_p \mathtt{sk}_{\mathsf{C}}$. However, this means that C would need to compute $\mathtt{sk}_{\mathsf{C}}$, which is assumed to be infeasible (by security of Sig[G]).

Strawman Two. A more promising approach is therefore to embed $\mathtt{pk}_{\mathsf{C}}$ into the master public key mpk within the simulation. This way, every answer to a query PK(ID) can easily be computed as $(\omega_{\mathtt{ID}}, \cdot) \leftarrow \mathsf{H}(\mathtt{St}, \mathtt{ID}^*), \mathtt{pk}_{\mathtt{ID}} \leftarrow \mathtt{mpk} \cdot g^{\omega_{\mathtt{ID}}}$. However, this approach significantly complicates the simulation of signing queries for C, since it does not know the secret keys to any of the session keys computed in this fashion. Even worse, it appears that C cannot even use its signing oracle Sign in the way sketched in the first strawman approach, as Sign returns signature *exclusively* valid under $\mathtt{pk}_{\mathsf{C}}$ and not under any of the keys returned via the oracle PK. However, C can overcome the latter issue related to the answering of signing queries via the related key derivability of Sig[G]. Namely, since it knows for each key $\mathtt{pk}_{\mathtt{ID}}$ returned via PK a value $\omega_{\mathtt{ID}}$ such that $\mathtt{mpk} \cdot g^{\omega_{\mathtt{ID}}}$ it can use the related key derivability of Sig[G] as follows: Upon receiving a signing query of the form $\mathsf{WalSign}(m, \mathtt{ID})$ C first asks for the signature $\sigma \leftarrow \mathsf{Sign}(m)$

(which is valid under $\mathtt{pk}_{\mathsf{C}} = \mathtt{mpk}$). It then retrieves $\omega_{\mathtt{ID}}$ and uses the algorithm $\mathsf{Trf}_{\mathsf{Sig[G]}}$ to convert σ into a signature $\hat{\sigma}$ that is valid under the related key $\mathtt{pk}_{\mathtt{ID}}$. Similarly, it can convert a forgery (σ^*, m^*) under an arbitrary related key $\mathtt{pk}_{\mathtt{ID}^*}$ into a forgery that is valid under $\mathtt{mpk}$, using $\mathsf{Trf}_{\mathsf{Sig[G]}}$ in the 'reverse' direction.

THE SOLUTION. To make the above approach work however, we need to address one more subtle issue. While the related key derivability works in C's favor during the simulation, it may become a real-world security vulnerability when C attempts to convert A's forgery (σ^*, m^*) under the public key $\mathtt{pk}_{\mathtt{ID}^*}$ into a forgery (σ, m) that is valid under $\mathtt{pk}_{\mathsf{C}}$. Namely, without any further countermeasures, A can easily launch a RKA against $\mathsf{swal[G,H]}$: It first queries $\mathsf{WalSign}(m^*)$ to obtain a signature $\sigma_{\mathtt{ID}}$ for the message m^* under some public key $\mathtt{pk}_{\mathtt{ID}}$. Since A can efficiently compute ψ such that $\mathtt{pk}_{\mathtt{ID}} = \mathtt{pk}_{\mathtt{ID}^*} \cdot g^{\psi}$ (using its knowledge of $\mathtt{St}$), it can subsequently compute σ^* as $\sigma^* \overset{\$}{\leftarrow} \mathsf{Trf}_{\mathsf{Sig[G]}}(m^*, \sigma^*, \psi, \mathtt{pk}_{\mathtt{ID}^*}, \mathtt{pk}_{\mathtt{ID}})$. The question is now where does the above attack make the simulation strategy of C in our second strawman solution fail. The call $\mathsf{WalSign}(m^*, \mathtt{ID})$ forces C to query the oracle Sign on m^*, in order to compute $\sigma_{\mathtt{ID}} \leftarrow \mathsf{Trf}_{\mathsf{Sig[G]}}(m^*, \sigma_{\mathtt{ID}}, -\omega_{\mathtt{ID}}, \mathtt{pk}_{\mathtt{ID}}, \mathtt{mpk})$, where $\mathtt{pk}_{\mathtt{ID}} = \mathtt{mpk} \cdot g^{\omega_{\mathtt{ID}}}$. However, since m^* has now been queried to Sign, a forgery on m^* no longer constitutes a valid forgery in game $\mathbf{uf\text{-}cma}_{\mathsf{Sig[G]}}$. Thus, C can not convert A's forgery into a forgery to win $\mathbf{uf\text{-}cma}_{\mathsf{Sig[G]}}$ as sketched in strawman two.

In the following, we will drop the distinction between queries to the oracle PK and G, H to simplify the exposition of the theorem and the proof. This is justified by the fact that every query to PK entails a query to H.

Theorem 4.2 *Let* A *be an algorithm that plays in the unforgeability game* $\mathbf{wunf}_{\mathsf{swal[G,H]}}$*, where* $\mathsf{swal[G,H]}$ *denotes the construction defined in Figure 10. Then if* $\mathsf{Sig[G]}$ *is related key derivable, there exists an algorithm* C *running in roughly the same time as* A*, such that*

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

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

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

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

GAME $\mathbf{G}_1$: $\mathbf{G}_1$ behaves as $\mathbf{G}_0$, but aborts whenever $\mathtt{flag}$ is set to true. We let E_1 denote the event that $\mathtt{flag} = \mathsf{true}$ during the execution of $\mathbf{G}_1$.

Claim 4.3 $\Pr[E_1] \leq \frac{q^2}{p}$.

Proof. By definition of DPKDer, the event E_1 implies a collision of the form $\omega_{\mathtt{ID}} = \omega'_{\mathtt{ID}}$ where $(\omega_{\mathtt{ID}}, \cdot) = \mathsf{H}(\mathtt{St}, \mathtt{ID})$, $(\omega'_{\mathtt{ID}}, \cdot) = \mathsf{H}(\mathtt{St}', \mathtt{ID}')$, and $\mathtt{ID} \neq \mathtt{ID}'$. As there are at most q queries to H, the claim immediately follows. ■

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

We now reduce winning in game $\mathbf{G}_1$ to the security of the underlying signature scheme. To this end, we describe an algorithm $\mathsf{C}^{\mathsf{Sign},\mathsf{G}}$ (depicted in Figure 11) that plays in game $\mathbf{uf\text{-}cma}_{\mathsf{Sig[G]}}$. C obtains as input a public key $\mathtt{pk}_{\mathsf{C}}$ and is given access to the signing oracle WSign to obtain signatures under $\mathtt{pk}_{\mathsf{C}}$. Furthermore, C has access to the random oracle G. C simulates $\mathbf{G}_1$ to A as described in the following.

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

SIMULATION OF RANDOM ORACLE QUERIES. C has to answer two types of random oracle queries made by A :

main $\mathsf{C}^{\mathsf{Sign},\mathsf{G}}_{\mathbf{wunf}}(\mathsf{pk}_{\mathsf{C}})$
00 $\mathtt{St} \xleftarrow{\$} \{0,1\}^{\kappa}, \mathtt{mpk} \leftarrow \mathsf{pk}_{\mathsf{C}}$
01 $(m^*, \sigma^*, \mathtt{ID}^*) \leftarrow \mathsf{A}^{\mathsf{PK},\mathsf{WSign},\mathsf{H}}(\mathtt{mpk}, \mathtt{St})$
02 If $\mathtt{SSNKeys}[\mathtt{ID}^*] = \perp \vee m^* \in \mathtt{Sigs}[\mathtt{ID}^*]$: Abort
03 $(\mathtt{pk}_{\mathtt{ID}^*}, \omega_{\mathtt{ID}^*}) \leftarrow \mathtt{SSNKeys}[\mathtt{ID}^*]$
04 If $\mathsf{DWVerify}^{\mathsf{G}}(\mathtt{pk}_{\mathtt{ID}^*}, \sigma^*, m^*) = 0$: Abort
05 $\hat{m}^* \leftarrow (\mathtt{pk}_{\mathtt{ID}^*}, m^*)$
06 $\hat{\sigma}^* \xleftarrow{\$} \mathsf{Trf}^{\mathsf{G}}_{\mathsf{Sig}[\mathsf{G}]}(\hat{m}^*, \sigma^*, \omega_{\mathtt{ID}^*}, \mathtt{mpk}, \mathtt{pk}_{\mathtt{ID}^*})$
07 Return $(\hat{m}^*, \hat{\sigma}^*)$

Procedure $\mathsf{H}(s)$
08 If $H[s] \neq \perp$
09 Return $H[s]$
10 $H[s] \xleftarrow{\$} \{0,1\}^{2\kappa}$
11 Return $H[s]$

Procedure $\mathsf{WalSign}(m, \mathtt{ID})$
12 If $\mathtt{SSNKeys}[\mathtt{ID}] = \perp$: Return $\perp$
13 $(\mathtt{pk}_{\mathtt{ID}}, \omega_{\mathtt{ID}}) \leftarrow \mathtt{SSNKeys}[\mathtt{ID}]$
14 $\hat{m} \leftarrow (\mathtt{pk}_{\mathtt{ID}}, m)$
15 $\hat{\sigma} \leftarrow \mathsf{Sign}^{\mathsf{G}}(\hat{m})$
16 $\sigma \leftarrow \mathsf{Trf}^{\mathsf{G}}_{\mathsf{lsig}}(\hat{m}, \hat{\sigma}, -\omega_{\mathtt{ID}}, \mathtt{pk}_{\mathtt{ID}}, \mathtt{mpk})$
17 $\mathtt{Sigs}[\mathtt{ID}] \leftarrow \mathtt{Sigs}[\mathtt{ID}] \cup \{\hat{m}\}$
18 Return σ

Procedure $\mathsf{PK}(\mathtt{ID})$ //Once per ID
19 $(\omega_{\mathtt{ID}}, \mathtt{St}) \leftarrow \mathsf{H}(\mathtt{St}, \mathtt{ID})$
20 $\mathtt{pk}_{\mathtt{ID}} \leftarrow \mathtt{mpk} \cdot g^{\omega_{\mathtt{ID}}}$
21 If $(\mathtt{pk}_{\mathtt{ID}}, \omega_{\mathtt{ID}}) \in \mathtt{SSNKeys}$: Abort
22 $\mathtt{SSNKeys}[\mathtt{ID}] \leftarrow (\mathtt{pk}_{\mathtt{ID}}, \omega_{\mathtt{ID}})$
23 Return $\mathtt{pk}_{\mathtt{ID}}$

Figure 11: C's simulation of $\mathbf{wunf}_{\mathsf{swal}[\mathsf{G},\mathsf{H}]}$ to A.

- A query of the form $\mathsf{G}(m)$: To simulate a query of this form, C simply forwards m to its own oracle G, i.e., answers A's query by returning the value $\mathsf{G}(m)$.
- A query of the form $\mathsf{H}(m)$: Queries of this type are simulated in a straight forward way as explained in Figure 11.

Simulation of Public Key Queries. C answers a call of A to $\mathsf{PK}(\mathtt{ID})$ by computing $pk_{\mathtt{ID}}$ as $pk_{\mathtt{ID}} = \mathtt{mpk} \cdot g^{\omega_{\mathtt{ID}}}$, where $(\omega_{\mathtt{ID}}, \mathtt{St}) \leftarrow \mathsf{H}(\mathtt{St}, \mathtt{ID})$ (note that the variable $\mathtt{St}$ is updated in this way). If it detects a collision among $(\mathtt{pk}_{\mathtt{ID}}, \omega_{\mathtt{ID}})$ and a value previously stored in $\mathtt{SSNKeys}$, C aborts the simulation. Otherwise, it sets $\mathtt{SSNKeys}[\mathtt{ID}] \leftarrow (\mathtt{pk}_{\mathtt{ID}}, \omega_{\mathtt{ID}})$ and returns $\mathtt{pk}_{\mathtt{ID}}$.

Simulation of Signing Queries. When A queries WSign on input $(m, \mathtt{ID})$, C first recovers the pair $(\mathtt{pk}_{\mathtt{ID}}, \omega_{\mathtt{ID}}) \leftarrow \mathtt{SSNKeys}[\mathtt{ID}]$ (it returns $\perp$ if $\mathtt{SSNKeys}[\mathtt{ID}] = \perp$). Next, C sets $\hat{m} = (\mathtt{pk}_{\mathtt{ID}}, m)$ and obtains $\hat{\sigma} \xleftarrow{\$} \mathsf{Sign}(\hat{m})$ by querying its own challenge signing oracle. By the related key derivability property of $\mathsf{Sig}[\mathsf{G}]$ and since both $\mathtt{pk}_{\mathtt{ID}} = \mathtt{mpk} \cdot g^{\omega_{\mathtt{ID}}}$ and $\mathsf{Verify}^{\mathsf{G}}(\mathtt{mpk}, \hat{\sigma}, \hat{m}) = 1$, we have that

$$\sigma \leftarrow \mathsf{Trf}^{\mathsf{G}}_{\mathsf{Sig}[\mathsf{G}]}(\hat{m}, \hat{\sigma}, -\omega_{\mathtt{ID}}, \mathtt{pk}_{\mathtt{ID}}, \mathtt{mpk}).$$

such that $\mathsf{Verify}^{\mathsf{G}}(\mathtt{pk}_{\mathtt{ID}}, \sigma, \hat{m}) = \mathsf{DWVerify}^{\mathsf{G}}(\mathtt{pk}_{\mathtt{ID}}, \sigma, m) = 1$.

Extracting the forgery. When A returns the tuple $(m^*, \sigma^*, \mathtt{ID}^*)$, C aborts if it encounters any of the cases in which $\mathbf{G}_1$ would return 0 at this point (c.f. Figure 11). Otherwise it proceeds as follows. It first recovers the pair $(\mathtt{pk}_{\mathtt{ID}^*}, \omega_{\mathtt{ID}^*}) \leftarrow \mathtt{SSNKeys}[\mathtt{ID}^*]$, and then computes $\hat{\sigma}^*$ as

$$\hat{\sigma}^* \xleftarrow{\$} \mathsf{Trf}^{\mathsf{G}}_{\mathsf{Sig}[\mathsf{G}]}(\hat{m}^*, \hat{\sigma}^*, \omega_{\mathtt{ID}^*}, \mathtt{mpk}, \mathtt{pk}_{\mathtt{ID}^*}).$$

Similar to the above case, related key derivability of $\mathsf{Sig}[\mathsf{G}]$ ensures that $\mathsf{Verify}^{\mathsf{G}}(\mathtt{mpk}, \hat{\sigma}^*, \hat{m}^*) = 1$. This is due to the fact $\mathsf{DWVerify}^{\mathsf{G}}(\mathtt{pk}_{\mathtt{ID}^*}, \sigma^*, m^*) = 1$ and $\mathtt{pk}_{\mathtt{ID}^*} = \mathtt{mpk} \cdot g^{\omega^*}$. C finally returns the tuple $(\hat{m}^*, \hat{\sigma}^*)$. Note that since $(m^*, \sigma^*, \mathtt{ID}^*)$ constitutes a valid forgery in $\mathbf{wunf}_{\mathsf{swal}[\mathsf{G},\mathsf{H}]}$, the query $\mathsf{WalSign}(m^*, \mathtt{ID}^*)$ was never posed by A and thus C never made the query $\mathsf{Sign}(\hat{m}^*)$, where $\hat{m}^* = (\mathtt{pk}_{\mathtt{ID}^*}, m^*)$. Therefore, $(\hat{m}^*, \hat{\sigma}^*)$ is a valid forgery in the game $\mathbf{uf\text{-}cma}_{\mathsf{Sig}[\mathsf{G}]}$.

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

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

which implies the theorem.

■

5 A Construction from ECDSA

In this section, we prove security of a construction based on the $\mathsf{ECDSA}[\mathsf{H}]$ scheme (cf. Figure 5). For the following discussion, let $\mathbb{E}(\mathtt{par})$ denote an elliptic curve with base point G and prime order p. Furthermore, assume hash functions $\mathsf{G}\colon \{0,1\}^* \to \mathbb{Z}_p$ and $\mathsf{H}_0\colon \{0,1\}^* \to \mathbb{Z}_p$ and $\mathsf{H}_1\colon \{0,1\}^* \to \mathbb{Z}_p \times \{0,1\}^\kappa$ (modeled as random oracles). Our construction $\mathsf{swal}[\mathsf{ECDSA}, \mathsf{H}_0, \mathsf{H}_1]$ is shown in Figure 12. It consists of a tuple of algorithms ($\mathsf{ECMGen}, \mathsf{ECSKDer}, \mathsf{ECPKDer}, \mathsf{ECWSign}, \mathsf{ECWVerify}$). Algorithms ($\mathsf{ECMGen}$, $\mathsf{ECSKDer}$, $\mathsf{ECPKDer}$) works as (MGen, SKDer, PKDer) defined in Def. 3.1. The $\mathsf{ECWSign}$ and $\mathsf{ECWVerify}$ algorithms work slightly different compared to standard ECDSA signing and verification. When requested to sign a message m under key $\mathtt{pk}$, the wallet signing algorithm $\mathsf{ECWSign}$ first samples a random value $\psi \xleftarrow{\$} \{0,1\}^\kappa$, then generates a message $\hat{m} \leftarrow (\psi, \mathtt{pk}, m)$, where m is prefixed with $(\psi, \mathtt{pk})$. Then, $\mathsf{ECWSign}$ signs message $\hat{m}$ by making a call to the underlying ECDSA signing algorithm $\mathsf{ECDSASign}^{\mathsf{H}_0}$. To verify a signature σ of message m and public key $\mathtt{pk}$, the signature needs to be parsed as $(\psi, \sigma') \leftarrow \sigma$. Message $\hat{m}$ is then reconstructed from m as $\hat{m} \leftarrow (\psi, \mathtt{pk}, m)$. The ECDSA verification algorithm $\mathsf{ECDSAVer}^{\mathsf{H}_0}$ is then run on input $\mathsf{ECDSAVer}^{\mathsf{H}_0}(\mathtt{pk}, \sigma', \hat{m})$ outputting 0 or 1.

Procedure $\mathsf{ECMGen}(\mathtt{par})$	Procedure $\mathsf{ECSKDer}^{\mathsf{H}_1}(\mathtt{msk}, \mathtt{ID}, \mathtt{St})$
00 $\mathtt{St} \xleftarrow{\$} \{0,1\}^\kappa$	00 $(\omega_{\mathtt{ID}}, \mathtt{St}) \leftarrow \mathsf{H}_1(\mathtt{St}, \mathtt{ID})$
01 $(\mathtt{mpk}, \mathtt{msk}) \leftarrow \mathsf{ECDSAGen}(\mathtt{par})$	01 $\mathtt{sk}_{\mathtt{ID}} \leftarrow \mathtt{msk} \cdot \omega_{\mathtt{ID}} \bmod p$
02 Return $(\mathtt{St}, \mathtt{mpk}, \mathtt{msk})$	02 Return $(\mathtt{sk}_{\mathtt{ID}}, \mathtt{St})$
Procedure $\mathsf{ECWSign}^{\mathsf{H}_0}(m, \mathtt{sk}, \mathtt{pk})$	Procedure $\mathsf{ECPKDer}^{\mathsf{H}_1}(\mathtt{mpk}, \mathtt{ID}, \mathtt{St})$
03 $\psi \xleftarrow{\$} \{0,1\}^\kappa$	03 $(\omega_{\mathtt{ID}}, \mathtt{St}) \leftarrow \mathsf{H}_1(\mathtt{St}, \mathtt{ID})$
04 $\hat{m} \leftarrow (\psi, \mathtt{pk}, m)$	04 $\mathtt{pk}_{\mathtt{ID}} \leftarrow \mathtt{mpk} \cdot \omega_{\mathtt{ID}}$
05 $\sigma' \leftarrow \mathsf{ECDSASign}^{\mathsf{H}_0}(\mathtt{sk}, \hat{m})$	05 Return $(\mathtt{pk}_{\mathtt{ID}}, \mathtt{St})$
06 Return $\sigma = (\psi, \sigma')$	
Procedure $\mathsf{ECWVerify}^{\mathsf{H}_0}(\mathtt{pk}, \sigma, m)$	
07 $(\psi, \sigma') \leftarrow \sigma$	
08 $\hat{m} \leftarrow (\psi, \mathtt{pk}, m)$	
09 Return $\mathsf{ECDSAVer}^{\mathsf{H}_0}(\mathtt{pk}, \sigma', \hat{m})$	

Figure 12: Construction of $\mathsf{swal}[\mathsf{ECDSA}, \mathsf{H}_0, \mathsf{H}_1]$ from the ECDSA signature scheme $\mathsf{ECDSA}[\mathsf{H}_0] = \left(\mathsf{ECDSAGen}, \mathsf{ECDSASign}^{\mathsf{H}_0}, \mathsf{ECDSAVer}^{\mathsf{H}_0}\right)$ and hash functions $\mathsf{H}_0\colon \{0,1\}^* \to \mathbb{Z}_p$ and $\mathsf{H}_1\colon \{0,1\}^* \to \mathbb{Z}_p \times \{0,1\}^\kappa$.

5.1 Security Analysis of Our Construction

The security analysis of $\mathsf{swal}[\mathsf{ECDSA}, \mathsf{H}_0, \mathsf{H}_1]$ consist of showing wallet unlinkability and wallet unforgeability. We start with the unlinkability property, which is formalized in Theorem 5.1:

Theorem 5.1 *Let* $\mathsf{swal}[\mathsf{ECDSA}, \mathsf{H}_0, \mathsf{H}_1]$ *be the construction defined in Figure 12. Then for any adversary* A *playing in unlinkability game* $\mathbf{unl}_{\mathsf{swal}[\mathsf{ECDSA}, \mathsf{H}_0, \mathsf{H}_1]}$*, we have*

$$\mathsf{Adv}^{\mathsf{A}}_{\mathbf{unl}, \mathsf{swal}[\mathsf{ECDSA}, \mathsf{H}_0, \mathsf{H}_1]} \leq \frac{q_H(q_P + 1)}{2^\kappa},$$

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

The proof of Theorem 5.1 is almost identical to the proof of Theorem 4.1, and thus omitted.

We now proceed to the main technical contribution of this paper, and analyze the unforgeability security of the construction $\mathsf{swal}[\mathsf{ECDSA}, \mathsf{H}_0, \mathsf{H}_1]$ presented in Figure 12. We prove the following theorem.

Theorem 5.2 *Let* $\mathsf{G}, \mathsf{H}_0 \colon \{0,1\}^* \to \mathbb{Z}_p$ *and* $\mathsf{H}_1 \colon \{0,1\}^* \to \mathbb{Z}_p \times \{0,1\}^\kappa$ *be hash functions (modeled as random oracles). Let* A *be an algorithm that plays in game* $\mathbf{wunf}_{\mathsf{swal}[\mathsf{ECDSA},\mathsf{H}_0,\mathsf{H}_1]}$. *Then there exists an algorithm* C *running in roughly the same time as* A*, such that*

$$\mathsf{Adv}^{\mathsf{A}}_{\mathbf{wunf},\mathsf{swal}[\mathsf{ECDSA},\mathsf{H}_0,\mathsf{H}_1]} \leq \mathsf{Adv}^{\mathsf{C}}_{\mathbf{uf\text{-}cma},\mathsf{ECDSA}[\mathsf{G}]} + \frac{4q^2}{p},$$

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

Before giving the formal proof, we give some intuition about the main difficulties that arise during C's simulation. At a high level, the idea is to reduce the security of the stateful wallet scheme $\mathsf{swal}[\mathsf{ECDSA}, \mathsf{H}_0, \mathsf{H}_1]$ (relative to $\mathbf{wunf}_{\mathsf{swal}[\mathsf{ECDSA},\mathsf{H}_0,\mathsf{H}_1]}$) to the security of $\mathsf{ECDSA}[\mathsf{G}]$ (relative to $\mathbf{uf\text{-}cma}_{\mathsf{ECDSA}[\mathsf{G}]}$). 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 $\mathbf{uf\text{-}cma}_{\mathsf{ECDSA}[\mathsf{G}]}$ by simulating $\mathbf{wunf}_{\mathsf{swal}[\mathsf{ECDSA},\mathsf{H}_0,\mathsf{H}_1]}$ to an adversary A. C obtains a public key $\mathtt{pk}_{\mathsf{C}}$ from its challenger and can query a signing oracle $\mathsf{ECDSASign}$ which provides signatures on messages of C's choice under $\mathtt{pk}_{\mathsf{C}}$. It also can query the random oracle G. C's goal is to simulate the oracles in the $\mathbf{wunf}_{\mathsf{swal}[\mathsf{ECDSA},\mathsf{H}_0,\mathsf{H}_1]}$ experiment and to suitably embed $\mathtt{pk}_{\mathsf{C}}$ into the key $\mathtt{pk}_{\mathtt{ID}^*}$ under which A eventually returns a forgery (σ^*, m^*). The hope is that it can use (σ^*, m^*) to win $\mathbf{uf\text{-}cma}_{\mathsf{ECDSA}[\mathsf{G}]}$.

As a first attempt, C generates a master key pair as $\mathtt{msk} \xleftarrow{\$} \mathbb{Z}_p$, $\mathtt{mpk} \leftarrow \mathtt{msk} \cdot G$, embeds the challenge public key pk_{C} into one of the generated session public keys as $pk_{\mathtt{ID}^*} \leftarrow pk_{\mathsf{C}}$ during its simulation. But this simulation attempt fails when A makes a query of the form $\mathsf{PK}(\mathtt{ID}^*)$ to C, and C returns pk_{C} as an answer. Since A knows the initial wallet state $\mathtt{St}$, A can compute $pk_{\mathtt{ID}^*} \leftarrow \mathtt{mpk} \cdot \omega_{\mathtt{ID}^*}$, where $(\mathtt{St}, \omega_{\mathtt{ID}^*}) \leftarrow \mathsf{H}_1(\mathtt{St}, \mathtt{ID}^*)$ and then check if $pk_{\mathsf{C}} = \mathtt{mpk} \cdot \omega_{\mathtt{ID}^*}$. To make pk_{C} indistinguishable from any $pk_{\mathtt{ID}}$ from A's perspective, C needs to set $\mathtt{msk} \cdot \omega_{\mathtt{ID}^*} = sk_{\mathsf{C}}$. The simulation fails because C does not know sk_{C}.

As an alternative approach C embeds pk_{C} into the master public key $\mathtt{mpk}$. This allows C to answer any $\mathsf{PK}(\mathtt{ID})$ query to A. However, there are several issues with this approach. Firstly, C is not aware of any of the session secret keys for the session public keys generated as $pk_{\mathtt{ID}} \leftarrow \mathtt{mpk} \cdot \omega_{\mathtt{ID}} (= pk_{\mathsf{C}} \cdot \omega_{\mathtt{ID}})$. Secondly, the signatures obtained by making a query $\mathsf{ECDSASign}(\cdot)$ to C's challenger are only valid under pk_{C}, so cannot be directly used to simulate signature queries of the form $\mathsf{WalSign}(m, \mathtt{ID})$ to A. To solve the latter problem, C converts signature $\sigma \leftarrow \mathsf{ECDSASign}(m')$ under pk_{C} into a signature $\hat{\sigma}$ valid under $pk_{\mathtt{ID}}$, under message $\hat{m}$ using algorithm $\mathsf{Trf}^{\mathsf{H},\mathsf{G}}_{\mathsf{ECDSA}}$, where pk_{C} and $pk_{\mathtt{ID}}$ are related as $pk_{\mathtt{ID}} = pk_{\mathsf{C}} \cdot \omega_{\mathtt{ID}}$, and $\omega_{\mathtt{ID}} \leftarrow \frac{G(m')}{\mathsf{H}_0(\hat{m})}$. Similarly, it can convert a forgery (σ^*, m^*) under an arbitrary related key $\mathtt{pk}_{\mathtt{ID}^*}$ into a forgery that is valid under pk_{C}, using $\mathsf{Trf}_{\mathsf{ECDSA}}$ in the "reverse" direction. 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 5.2.

Proof. Consider an adversary A playing in Game $\mathbf{wunf}_{\mathsf{swal}[\mathsf{ECDSA},\mathsf{H}_0,\mathsf{H}_1]}$. As such A is granted access to the oracles $\mathsf{PK}, \mathsf{WalSign}$, and the random oracles $\mathsf{H}_0 \colon \{0,1\}^* \to \mathbb{Z}_p, \mathsf{H}_1 \colon \{0,1\}^* \to \mathbb{Z}_p \times \{0,1\}^\kappa$. 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 represented in a figure with description of the oracles which are modified with respect to the previous game $\mathbf{G}_{i-1}$. The exact differences of game $\mathbf{G}_i$ to previous game $\mathbf{G}_{i-1}$ is highlighted in the form of boxed pseudocode. Moreover, we denote by $E_{i-1,i}$ an 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 $\mathbf{G}_0$ essentially corresponds to the unforgeability game $\mathbf{wunf}$ in figure 9 instantiated with the ECDSA-based wallet construction $\mathsf{swal}[\mathsf{ECDSA}, \mathsf{H}_0, \mathsf{H}_1]$. This implies $\mathbf{G}_0 := \mathbf{wunf}_{\mathsf{swal}[\mathsf{ECDSA},\mathsf{H}_0,\mathsf{H}_1]}$. Due to space constraints, $\mathbf{G}_0$ is presented in Figure 13 in the Appendix. Since we are in the random oracle model, we explicitly list the random oracles H_0 and H_1 in $\mathbf{G}_0$ in contrast to figure 9.

GAME $\mathbf{G}_1$: In $\mathbf{G}_1$, the way in which public key queries from A are answered, is internally modified as follows. When A asks a query of the form $\mathsf{PK}(\mathtt{ID})$, the answer $\mathtt{pk}_{\mathtt{ID}}$ is computed as $\mathtt{pk}_{\mathtt{ID}} \leftarrow \mathtt{mpk} \cdot \omega_{\mathtt{ID}}$, where $(\omega_{\mathtt{ID}}, \mathtt{St}) \leftarrow \mathsf{H}_1(\mathtt{St}', \mathtt{ID})$. For every pair of derived key $(\mathtt{sk}_{\mathtt{ID}}, \mathtt{pk}_{\mathtt{ID}})$, $\mathbf{G}_1$ stores the value $\omega_{\mathtt{ID}}$ as $\mathtt{SSNKeys}'[\mathtt{ID}] \leftarrow (\mathtt{sk}_{\mathtt{ID}}, \mathtt{pk}_{\mathtt{ID}}, \omega_{\mathtt{ID}})$ in a list $\mathtt{SSNKeys}'$. $\mathbf{G}_1$ aborts if the tuple $(\mathtt{sk}_{\mathtt{ID}}, \mathtt{pk}_{\mathtt{ID}}, \omega_{\mathtt{ID}})$ already exists

```
main G_0                                              Oracle PK(ID) // Once per ID
00 ctr ← 0                                            15 St' ← St
// ctr is a counter required for programming         16 (sk_ID, St) ← ECSKDer^H1 (msk, ID, St')
oracle H_0                                            17 (pk_ID, St) ← ECPKDer^H1 (mpk, ID, St')
01 (St, msk, mpk) ←$ MGen(par)                        18 SSNKeys[ID] ← (pk_ID, sk_ID)
02 (m*, σ*, ID*) ←$ A^{H0,H1,PK,WalSign} (mpk, St)    19 Sigs[ID] ← ∅
03 If SSNKeys[ID*] = ⊥                                20 Return pk_ID
04     Return 0
05 (pk_ID*, sk_ID*) ← SSNKeys[ID*]                    Oracle WalSign(m, ID)
06 If m* ∈ Sigs[ID*]                                  21 If SSNKeys[ID] = ⊥ : Return ⊥
07     Return 0                                       22 (pk_ID, sk_ID) ← SSNKeys[ID]
08 If ECWVerify^H0 (pk_ID*, σ*, m*) = 0               23 σ ←$ ECWSign^H0 (m, sk_ID, pk_ID)
09     Return 0                                       24 Sigs[ID] ← Sigs[ID] ∪ {m}
10 Return 1                                           25 Return σ

Oracle H_0(m)                                         Oracle H_1(St, ID)
11 If H_0[m] ≠ ⊥                                      26 If H_1[St, ID] ≠ ⊥
12     Return H_0[m]                                  27     Return H_1[St, ID]
13 H_0[m] ←$ Z_p                                      28 H_1[St, ID] ←$ Z_p × {0,1}^κ
14 Return H_0[m]                                      29 Return H_1[St, ID]
```

Figure 13: Game $\mathbf{G}_0 = \mathbf{wunf}_{\mathsf{swal}[\mathsf{ECDSA},\mathsf{H}_0,\mathsf{H}_1]}$

in the list $\mathtt{SSNKeys}'$. Besides this abort the main differences between $\mathbf{G}_0$ and $\mathbf{G}_1$ is the way in which queries to $\mathsf{PK}(\mathtt{ID})$ are answered and the way bookkeeping is done. Firstly regarding key derivation, in $\mathbf{G}_0$ within the $\mathsf{PK}(\mathtt{ID})$ oracle, the sessions keys $\mathtt{sk}_{\mathtt{ID}}$, $\mathtt{pk}_{\mathtt{ID}}$ are derived by making a call to the key derivation algorithms ECSKDer and ECPKDer at line 16, 17, figure 13. On the other hand, in game $\mathbf{G}_1$, session keys are derived by computing $\mathtt{sk}_{\mathtt{ID}}$, $\mathtt{pk}_{\mathtt{ID}}$ from $\omega_{\mathtt{ID}}$ at line 17, 18, figure 14. Secondly, as for the bookkeeping, in $\mathbf{G}_0$ we store the pair $(\mathtt{pk}_{\mathtt{ID}}, \mathtt{sk}_{\mathtt{ID}})$ in the list SSNKeys, while in $\mathbf{G}_1$ we additionally store $\omega_{\mathtt{ID}}$ in the tuple $(\mathtt{sk}_{\mathtt{ID}}, \mathtt{pk}_{\mathtt{ID}}, \omega_{\mathtt{ID}})$ in list $\mathtt{SSNKeys}'$. Note that this difference in key derivation and bookkeeping is a syntactical difference only.

Claim 5.3 Let $E_{0,1}$ denote the event that $\mathbf{G}_1$ aborts during a public key query, if the tuple $(\cdot, \cdot, \omega_{\mathtt{ID}})$ already exists in the list $\mathtt{SSNKeys}'$. Then $\Pr[E_{0,1}] \leq \frac{q^2}{p}$.

Proof. The proof of this claim follows in a similar way as the corresponding claim in the proof of Theorem 4.1. ■

Since the games $\mathbf{G}_0, \mathbf{G}_1$ are equivalent unless the event $E_{0,1}$ occurs, it follows from the claim that $\mathbf{Adv}^{\mathsf{A}}_{\mathbf{G_0},\mathsf{swal}[\mathsf{ECDSA},\mathsf{H}_0,\mathsf{H}_1]} \leq \mathbf{Adv}^{\mathsf{A}}_{\mathbf{G_1},\mathsf{swal}[\mathsf{ECDSA},\mathsf{H}_0,\mathsf{H}_1]} + \Pr[E_{0,1}] \leq \mathbf{Adv}^{\mathsf{A}}_{\mathbf{G_1},\mathsf{swal}[\mathsf{ECDSA},\mathsf{H}_0,\mathsf{H}_1]} + \frac{q^2}{p}$.

Game $\mathbf{G}_2$: In $\mathbf{G}_2$, the way that random oracle queries to H_0 from A are answered, is internally modified as follows. To answer queries to H_0, $\mathbf{G}_2$ 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 $\mathtt{pk}_{\mathtt{ID}}$ which previously was an answer to the query $\mathsf{PK}(\mathtt{ID})$, it programs $H_0[m]$ and $H_0'[m]$ in two different possible ways. We now analyze the three types of queries to H_0 that can occur.

- $\mathsf{H}_0[m] \neq \perp$: In this case, $\mathbf{G}_2$ returns $H_0[m]$.
- $\mathsf{H}_0[m] = \perp$ and m is of the form $m = (\cdot, \mathtt{pk}_{\mathtt{ID}}, \cdot)$, s.t. $\mathtt{pk}_{\mathtt{ID}}$ was previously returned as answer to the query $\mathsf{PK}(\mathtt{ID})$: In this case, $\mathbf{G}_2$ fetches $(\cdot, \cdot, \omega_{\mathtt{ID}}) \leftarrow \mathtt{SSNKeys}'[\mathtt{ID}]$. Then, it computes $h \leftarrow \mathsf{G}(ctr)$, where $ctr \in \{0,1\}^*$ is a global counter. ctr is initialized to 0 and incremented by 1 every time H_0 is queried (step 04, figure 15). Consequently, $\mathbf{G}_2$ sets $H_0[m] \leftarrow \omega_{\mathtt{ID}} \cdot h \mod p$ and $H_0'[m] \leftarrow ctr$. It returns $H_0[m]$.
- Otherwise, $\mathbf{G}_2$ samples $h \xleftarrow{\$} \mathbb{Z}_p$ and sets $H_0[m]$ to h, $H_0'[m]$ to 0. It then returns $H_0[m]$.

It is easy to see that, all answers for queries to H_0 that $\mathbf{G}_2$ returns are uniformly distributed from A's perspective. This follows from the uniformity of output h computed via random oracle G. Therefore, $\mathbf{G}_2$ behaves exactly as $\mathbf{G}_1$.

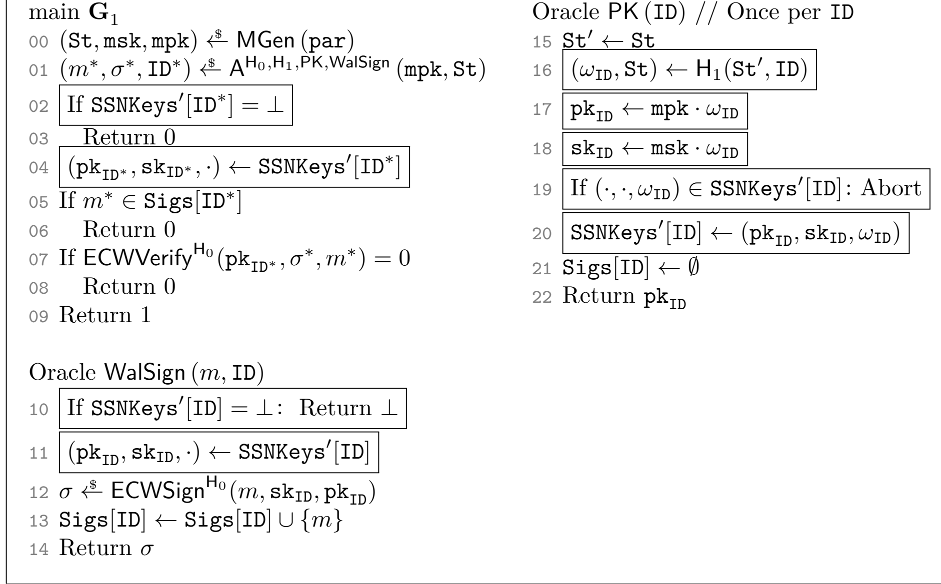

Figure 14: Game $\mathbf{G}_1$

Claim 5.4 $\mathbf{Adv}^{\mathsf{A}}_{\mathbf{G_1},\mathsf{swal}[\mathsf{ECDSA},\mathsf{H}_0,\mathsf{H}_1]} = \mathbf{Adv}^{\mathsf{A}}_{\mathbf{G_2},\mathsf{swal}[\mathsf{ECDSA},\mathsf{H}_0,\mathsf{H}_1]}$.

Proof. This follows from the fact that in both games $\mathbf{G}_1$ and $\mathbf{G}_2$, the answers to oracle queries H_0 are identically distributed from A's perspective. In $\mathbf{G}_1$, A's view of $H_0[m]$ is uniform because $H_0[m]$ is computed as $H_0[m] \xleftarrow{\$} \mathbb{Z}_p$. In $\mathbf{G}_2$, this is because of uniformity of the computation $h \leftarrow \mathsf{G}(ctr)$ and the value $H_0[m] \leftarrow \omega_{\mathtt{ID}} \cdot h$. ■

From the claim 5.4, it follows that the games $\mathbf{G}_1$, $\mathbf{G}_2$ are equivalent. Hence, $\mathbf{Adv}^{\mathsf{A}}_{\mathbf{G_1},\mathsf{swal}[\mathsf{ECDSA},\mathsf{H}_0,\mathsf{H}_1]} = \mathbf{Adv}^{\mathsf{A}}_{\mathbf{G_2},\mathsf{swal}[\mathsf{ECDSA},\mathsf{H}_0,\mathsf{H}_1]}$.

GAME $\mathbf{G}_3$: In $\mathbf{G}_3$, the way in which public key queries from A are answered, is internally modified as follows. When A asks a query of the form $\mathsf{PK}(\mathtt{ID})$, the session public key $\mathtt{pk}_{\mathtt{ID}}$ is derived as $\mathtt{pk}_{\mathtt{ID}} \leftarrow \mathtt{mpk} \cdot \omega_{\mathtt{ID}}$. If a tuple of the form $(\cdot, \cdot, \omega_{\mathtt{ID}})$ already exists in the list $\mathtt{SSNKeys}'$, then the game aborts. In the following step, the game aborts if there exists a message of the form $m = (\cdot, \mathtt{pk}_{\mathtt{ID}}, \cdot)$ for which $H'_0[m]$ evaluates to 0. Next the list $\mathtt{SSNKeys}'$ is updated as $\mathtt{SSNKeys}'[\mathtt{ID}] \leftarrow (\mathtt{pk}_{\mathtt{ID}}, \mathtt{sk}_{\mathtt{ID}}, \omega_{\mathtt{ID}})$. Note that, the way $\mathsf{PK}(\mathtt{ID})$ queries from A are answered in game $\mathbf{G}_3$ is different from $\mathbf{G}_2$ in the following way. In $\mathbf{G}_3$, there is an additional abort at line 17, figure 15. In the following claim, it is explained why the probabiltiy of occurance of the above-mentioned abort scenario shall be low.

Claim 5.5 Let $E_{2,3}$ denote the event that $\mathbf{G}_3$ aborts during a public key query, for which $H'_0[m]$ evaluates to 0, where $m = (\cdot, \mathtt{pk}_{\mathtt{ID}}, \cdot)$. Then $\Pr[E_{2,3}] \leq \frac{q^2}{p}$.

Proof. This event can only occur when A makes a query of the form $\mathsf{H}_0(m)$, where $m = (\cdot, pk_{\mathtt{ID}}, \cdot)$ is a message which is prefixed with a valid session public key $pk_{\mathtt{ID}}$ self-derived by A. In usual manner, $pk_{\mathtt{ID}}$ is derived by making a query of the form $\mathsf{PK}(\mathtt{ID})$, where $pk_{\mathtt{ID}} \leftarrow \omega_{\mathtt{ID}} \cdot \mathtt{mpk}$, and the $\omega_{\mathtt{ID}}$ is derived via oracle H_1 as $(\omega_{\mathtt{ID}}, \mathtt{St}) \leftarrow \mathsf{H}_1(\mathtt{St}', \mathtt{ID})$. Hence self-derivation of $pk_{\mathtt{ID}}$ by A is only possible when A can correctly guess $\omega_{\mathtt{ID}}$ without calling H_1. This is possible with probability $\frac{q}{p}$. This is because each $pk_{\mathtt{ID}}$ can be correctly guessed with probability $\frac{1}{p}$. Since the number of $pk_{\mathtt{ID}}$s is bounded by q, the probability of correctly guessing any $pk_{\mathtt{ID}}$ is $\left(\sum_{i=1}^{q} \frac{1}{p}\right) = \frac{q}{p} \leq \frac{q^2}{p}$. ■

Since the games $\mathbf{G}_2$, $\mathbf{G}_3$ are equivalent unless the event $\Pr[E_{2,3}]$ occurs,

$$\mathbf{Adv}^{\mathsf{A}}_{\mathbf{G_2},\mathsf{swal}[\mathsf{ECDSA},\mathsf{H}_0,\mathsf{H}_1]} \leq \mathbf{Adv}^{\mathsf{A}}_{\mathbf{G_3},\mathsf{swal}[\mathsf{ECDSA},\mathsf{H}_0,\mathsf{H}_1]} + \Pr[E_{2,3}] \leq \mathbf{Adv}^{\mathsf{A}}_{\mathbf{G_3},\mathsf{swal}[\mathsf{ECDSA},\mathsf{H}_0,\mathsf{H}_1]} + \frac{q^2}{p}.$$

```
Oracle H_0(m) in G_2
00 If H_0[m] ≠ ⊥
01    Return H_0[m]
02 Parse m as (·, pk_ID, ·) and set ω_ID ← pk_ID · mpk^{-1}
03 If (ω_ID, ·) ∈ H_1
04    ctr = ctr + 1
05    h ← G(ctr) // ctr ∈ {0,1}* is input message
06    H_0[m] ← ω_ID · h mod p
07    H'_0[m] ← ctr
08 Else
09    h ←$ Z_p
10    H_0[m] ← h
11    H'_0[m] ← 0
12 Return H_0[m]

Oracle PK(ID) in G_3
13 St' ← St
14 (ω_ID, St) ← H_1(St', ID)
15 pk_ID ← mpk · ω_ID and sk_ID ← msk · ω_ID
16 If (·, ·, ω_ID) ∈ SSNKeys'[ID]: Abort
17 [If ∃m = (·, pk_ID, ·) s.t.: H'_0[m] = 0: Abort]
18 SSNKeys'[ID] ← (pk_ID, sk_ID, ω_ID)
19 Sigs[ID] ← ∅
20 Return pk_ID

Oracle WalSign(m, ID) in G_4
21 If SSNKeys'[ID] = ⊥: Return ⊥
22 (pk_ID, sk_ID, ω_ID) ← SSNKeys'[ID]
23 [ψ ←$ {0,1}^κ]
24 [m̂ ← (ψ, pk_ID, m)]
25 [If H'_0[m̂] ≠ ⊥: Abort]
26 [σ̂ ← ECDSASign^{H_0}(sk_ID, m̂)]
27 Sigs[ID] ← Sigs[ID] ∪ {m}
28 Return (ψ, σ̂)

Oracle WalSign(m, ID) in G_5
29 If SSNKeys'[ID] = ⊥: Return ⊥
30 (pk_ID, ·, ω_ID) ← SSNKeys'[ID]
31 ψ ←$ {0,1}^κ
32 m̂ ← (ψ, pk_ID, m)
33 If H_0[m̂] ≠ ⊥: Abort
34 [Query H_0(m̂)]
35 [m' ← H'_0[m̂]]
36 [σ' ← ECDSASign^G(msk, m')]
37 [σ̂ ← Trf^{H_0,G}_{ECDSA}(m̂, m', σ', ω_ID^{-1}, pk_ID, mpk)]
38 Sigs[ID] ← Sigs[ID] ∪ {m}
39 Return (ψ, σ̂)

Oracle PK(ID) in G_6
40 St' ← St
41 (ω_ID, St) ← H_1(St', ID)
42 pk_ID ← mpk · ω_ID
43 If (·, ·, ω_ID) ∈ SSNKeys'[ID]: Abort
44 If ∃m = (·, pk_ID, ·) s.t.: H'_0[m] = ⊥: Abort
45 [SSNKeys'[ID] ← (pk_ID, ⊥, ω_ID)]
46 Sigs[ID] ← ∅
47 Return pk_ID

main G_7
48 ctr ← 0
49 St ←$ {0,1}^κ and (mpk, msk) ← ECDSAGen(par)
50 (m*, (ψ, σ̂) = σ*, ID*)
   ←$ A^{H_0,H_1,PK,WalSign}(mpk, St), // where ψ may be the empty string
51 [If SSNKeys'[ID*] = ⊥ ∨ m* ∈ Sigs[ID*]]
52    Return 0
53 [(pk_ID*, ⊥, ω_ID*) ← SSNKeys'[ID*]]
54 If ECWVerify^{H_0}(pk_ID*, σ*, m*) = 0
55    Return 0
56 [m̂* ← (ψ, pk_ID*, m*)]
57 [If H'_0[m̂*] = 0: Abort]
58 Return 1
```

Figure 15: Games $\mathbf{G}_2$-$\mathbf{G}_7$

Game $\mathbf{G}_4$: In $\mathbf{G}_4$, the way in which signing queries from A are answered are internally modified as follows. When A makes a query of the form $\mathsf{WalSign}(m, \mathtt{ID})$, $\mathbf{G}_4$ first checks whether there exists a tuple of the form $(\mathtt{pk}_{\mathtt{ID}}, \cdot, \omega_{\mathtt{ID}}) \in \mathtt{SSNKeys}'$ and if not, returns $\perp$. Otherwise, it samples $\psi \xleftarrow{\$} \{0,1\}^\kappa$ and sets $\hat{m} \leftarrow (\psi, \mathtt{pk}_{\mathtt{ID}}, m)$. 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 signature $\hat{\sigma}$ is evaluated as $\hat{\sigma} \leftarrow \mathsf{ECDSASign}^{\mathsf{H}_0}(\mathtt{sk}_{\mathtt{ID}}, \hat{m})$. Following, the signature list for $\mathtt{sk}_{\mathtt{ID}}$ is updated as $\mathtt{Sigs}[\mathtt{ID}] \leftarrow \mathtt{Sigs}[\mathtt{ID}] \cup \{m\}$. Note that difference in game $\mathbf{G}_4$ from $\mathbf{G}_3$ is two-fold. Firstly, the difference is syntactical in lines 23, 24, 26 (figure 15). This is because in $\mathbf{G}_4$ we explicitly write down the steps for the call $\sigma \xleftarrow{\$} \mathsf{ECWSign}^{\mathsf{H}_0}(m, \mathtt{sk}_{\mathtt{ID}}, \mathtt{pk}_{\mathtt{ID}})$ present in $\mathbf{G}_3$ (or $\mathbf{G}_1$, line 12, figure 14. Note here that $\mathsf{WalSign}$ has not been modified since game $\mathbf{G}_1$). Secondly, in game $\mathbf{G}_4$, the game aborts at line 25 if $H'_0[m] \neq \perp$.

Claim 5.6 Let $E_{3,4}$ denote the event that $\mathbf{G}_4$ aborts during a signing query, when $H_0[\hat{m}] \neq \perp$, where $\hat{m} = (\psi, pk_{\mathtt{ID}}, m)$. Then $\Pr[E_{3,4}] \leq \frac{q^2}{p}$.

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 $\mathsf{WalSign}(m, \mathtt{ID})$ query. $\hat{m}$ is constructed as $\hat{m} = (\psi, pk_{\mathtt{ID}}, m)$ where ψ is uniformly sampled as $\psi \xleftarrow{\$} \{0,1\}^\kappa$. A can correctly guess each $\hat{m}$ with probability $\frac{1}{p}$. Since A makes atmost q signing queries to $\mathsf{WalSign}\,(m, \mathtt{ID})$, A can correctly guess any $\hat{m}$ with a probability bounded by $\sum_{i=1}^{q} \frac{1}{p} = \frac{q}{p} \leq \frac{q^2}{p}$. ∎

Since the games $\mathbf{G}_3$, $\mathbf{G}_4$ are equivalent unless the event $\Pr[E_{3,4}]$ occurs,

$$\mathbf{Adv}^{\mathsf{A}}_{\mathbf{G_3},\mathsf{swal}[\mathsf{ECDSA},\mathsf{H}_0,\mathsf{H}_1]} \leq \mathbf{Adv}^{\mathsf{A}}_{\mathbf{G_4},\mathsf{swal}[\mathsf{ECDSA},\mathsf{H}_0,\mathsf{H}_1]} + \Pr\left[E_{3,4}\right] \leq \mathbf{Adv}^{\mathsf{A}}_{\mathbf{G_4},\mathsf{swal}[\mathsf{ECDSA},\mathsf{H}_0,\mathsf{H}_1]} + \frac{q^2}{p}.$$

Game $\mathbf{G}_5$: In $\mathbf{G}_5$, the way that signing queries from A are answered, is internally modified as follows. When A makes a query of the form $\mathsf{WalSign}(m, \mathtt{ID})$, $\mathbf{G}_5$ first checks whether there exists a tuple of the form $(\mathtt{pk}_{\mathtt{ID}}, \cdot, \omega_{\mathtt{ID}}) \in \mathtt{SSNKeys}'$ and if not, returns $\perp$. Otherwise, it samples $\psi \xleftarrow{\$} \{0,1\}^\kappa$ and sets $\hat{m} \leftarrow (\psi, \mathtt{pk}_{\mathtt{ID}}, m)$. The game aborts at this point if $H_0[\hat{m}] \neq \perp$. If it does not abort, it proceeds answering the oracle query to H_0 on input message $\hat{m}$. This means it queries $h \leftarrow \mathsf{G}\,(ctr)$, where ctr is a counter initialized with 1. $\mathbf{G}_5$ internally sets $H_0[\hat{m}] \leftarrow \omega_{\mathtt{ID}} \cdot h \mod p$ and stores $H_0'[\hat{m}] \leftarrow ctr$. After answering query to H_0, $\mathbf{G}_5$ fetches $m' \leftarrow H_0'[\hat{m}]$, where m' was set to ctr during H_0 query. Since $\mathtt{msk}$ is known to the game, it can now compute the signature σ' as $\mathsf{ECDSASign}^{\mathsf{G}}(\mathtt{msk}, m')$. Finally, it computes and returns the signature $\hat{\sigma}$ as $\hat{\sigma} \leftarrow \mathsf{Trf}^{\mathsf{H}_0,\mathsf{G}}_{\mathsf{ECDSA}}(\hat{m}, m', \sigma', \omega_{\mathtt{ID}}^{-1}, \mathtt{pk}_{\mathtt{ID}}, \mathtt{mpk})$, where

- $\mathtt{mpk} = \mathtt{pk}_{\mathtt{ID}} \cdot \omega_{\mathtt{ID}}^{-1}$,
- $\mathsf{ECDSAVer}^{\mathsf{G}}(\mathtt{mpk}, \sigma', m') = 1$,
- $\frac{\mathsf{G}(m')}{H_0[\hat{m},0]} = \frac{h'}{H_0[\hat{m},0]} = \frac{h'}{\omega_{\mathtt{ID}} \cdot h'} = \omega_{\mathtt{ID}}^{-1} \mod p$,

Claim 5.7 $\mathbf{Adv}^{\mathsf{A}}_{\mathbf{G_4},\mathsf{swal}[\mathsf{ECDSA},\mathsf{H}_0,\mathsf{H}_1]} = \mathbf{Adv}^{\mathsf{A}}_{\mathbf{G_5},\mathsf{swal}[\mathsf{ECDSA},\mathsf{H}_0,\mathsf{H}_1]}$

Proof. We argue that in both games, the answers to signing queries are identically distributed. To this end, we analyze how $\mathbf{G}_5$ replies to a query of the form $\mathsf{WalSign}\,(m, \mathtt{ID})$. It begins by internally sampling a random value $\psi \xleftarrow{\$} \mathbb{Z}_q$ and subsequently querying $\mathsf{H}_0(\hat{m}) = \mathsf{H}_0(\psi, m)$. In turn, H_0 sets $H_0\,[m] \leftarrow \omega_{\mathtt{ID}} \cdot \mathsf{G}\,(ctr)$ and stores $H_0'[\hat{m}] \leftarrow ctr$. After the query to H_0 is completed, $\mathbf{G}_5$ fetches $m' \leftarrow H_0'[\hat{m}]$, i.e., m' is set to the value held by variable ctr *before* the query to H_0 was made. Next, $\mathbf{G}_5$ derives signature $(\psi, \hat{\sigma})$ on input $(m, \mathtt{ID})$ as $(r, s_0) = \hat{\sigma} \leftarrow \mathsf{Trf}^{\mathsf{H}_0,\mathsf{G}}_{\mathsf{ECDSA}}(\hat{m}, m', \sigma', \omega_{\mathtt{ID}}^{-1}, \mathtt{pk}_{\mathtt{ID}}, \mathtt{mpk})$, where $(r, s_1) = \sigma' \leftarrow \mathsf{ECDSASign}^{\mathsf{G}}(\mathtt{msk}, m')$. That is, s_0 is computed (via $\mathsf{Trf}^{\mathsf{H}_0,\mathsf{G}}_{\mathsf{ECDSA}}$) as $s_0 \leftarrow s_1 \cdot \left(\frac{\mathsf{H}_0(\hat{m})}{\mathsf{G}(m')}\right)^{-1}$. It follows from Lemma 2.5 that $(\psi, \hat{\sigma})$ constitutes a correct signature on message m and under public key $\mathtt{pk}_{\mathtt{ID}}$. Moreover, the value of ψ is identically distributed in games $\mathbf{G}_4, \mathbf{G}_5$, which concludes the proof. ∎

Game $\mathbf{G}_6$: In $\mathbf{G}_6$, the way in which public key queries from A are answered, is internally modified as follows. When A asks a query of the form $\mathsf{PK}\,(\mathtt{ID})$, $\mathtt{pk}_{\mathtt{ID}}$ is derived as $\mathtt{pk}_{\mathtt{ID}} \leftarrow \mathtt{mpk} \cdot \omega_{\mathtt{ID}}$. If a tuple of the form $(\cdot, \cdot, \omega_{\mathtt{ID}})$ already exists in the list $\mathtt{SSNKeys}'$, then $\mathbf{G}_6$ aborts. Otherwise if there exists a message m of the form $(\cdot, \mathtt{pk}_{\mathtt{ID}}, \cdot)$, for which $H_0'\,[m] = \perp$, $\mathbf{G}_6$ aborts. If $\mathbf{G}_6$ does not abort then, $\mathbf{G}_6$ stores the tuple $(\mathtt{pk}_{\mathtt{ID}}, \perp, \omega_{\mathtt{ID}})$ in list $\mathtt{SSNKeys}'$. The way public key queries from A are answered in $\mathbf{G}_6$ is different from $\mathbf{G}_5$ in the following way. Firstly in $\mathbf{G}_6$ only $\mathtt{pk}_{\mathtt{ID}}$ is derived at line 42 (figure 15), while in $\mathbf{G}_5$, both $\mathtt{pk}_{\mathtt{ID}}$ and $\mathtt{sk}_{\mathtt{ID}}$ are derived at line 15 (figure 15. Note: $\mathsf{PK}\,(\mathtt{ID})$ did not change since game $\mathbf{G}_3$). Secondly tuple $(\mathtt{pk}_{\mathtt{ID}}, \perp, \omega_{\mathtt{ID}})$ is stored in list $\mathtt{SSNKeys}'$ at line 45 in $\mathbf{G}_6$ as opposed to $(\mathtt{pk}_{\mathtt{ID}}, \mathtt{sk}_{\mathtt{ID}}, \omega_{\mathtt{ID}})$ in list $\mathtt{SSNKeys}'$ at line 18 in $\mathbf{G}_5$.

In games $\mathbf{G}_5$ and $\mathbf{G}_6$, the answers to public key queries are identically distributed. Moreover the difference in $\mathbf{G}_6$ from $\mathbf{G}_5$ is purely syntactical as can be observed in line 45, figure 15. Hence $\mathbf{Adv}^{\mathsf{A}}_{\mathbf{G_5},\mathsf{swal}[\mathsf{ECDSA},\mathsf{H}_0,\mathsf{H}_1]} = \mathbf{Adv}^{\mathsf{A}}_{\mathbf{G_6},\mathsf{swal}[\mathsf{ECDSA},\mathsf{H}_0,\mathsf{H}_1]}$. Since $\mathbf{G}_4$, $\mathbf{G}_5$ and $\mathbf{G}_6$ are equivalent, it follows that

$$\mathbf{Adv}^{\mathsf{A}}_{\mathbf{G_4},\mathsf{swal}[\mathsf{ECDSA},\mathsf{H}_0,\mathsf{H}_1]} = \mathbf{Adv}^{\mathsf{A}}_{\mathbf{G_5},\mathsf{swal}[\mathsf{ECDSA},\mathsf{H}_0,\mathsf{H}_1]} = \mathbf{Adv}^{\mathsf{A}}_{\mathbf{G_6},\mathsf{swal}[\mathsf{ECDSA},\mathsf{H}_0,\mathsf{H}_1]}.$$

Game $\mathbf{G}_7$: Upon receiving a forgery of the form $(m^*,\ \sigma^* = (\psi, \hat{\sigma}),\ \mathtt{ID}^*)$ from A, $\mathbf{G}_7$ returns 0 if the following conditions hold

- If $\mathtt{SSNKeys}'[\mathtt{ID}^*] = \bot$
- $m^* \in \mathtt{Sigs}[\mathtt{ID}^*]$

Otherwise for $\mathtt{ID}^*$, $\mathbf{G}_7$ fetches $(\mathtt{pk}_{\mathtt{ID}^*}, \bot, \omega_{\mathtt{ID}^*}) \leftarrow \mathtt{SSNKeys}'[\mathtt{ID}^*]$. At this point, $\mathbf{G}_7$ returns 0 if A returned an invalid forgery, or in other words, if $\mathsf{ECWVerify}^{\mathsf{H}_0}(\mathtt{pk}_{\mathtt{ID}^*}, \sigma^*, m^*) = 0$. Otherwise $\mathbf{G}_7$ proceeds as follows. $\mathbf{G}_7$ sets $\hat{m}^* \leftarrow (\psi, \mathtt{pk}_{\mathtt{ID}^*}, m^*)$, where $(\psi, \hat{\sigma}) = \sigma^*$. If $H_0'[\hat{m}^*] = 0$, then $\mathbf{G}_7$ aborts. If not, $\mathbf{G}_7$ returns 1. Apart from this, the oracle queries to $\mathsf{PK}(\mathtt{ID})$, $\mathsf{WalSign}(m, \mathtt{ID})$, $\mathsf{H}_0(m)$, $\mathsf{H}_1(m)$ are answered in the same way as in $\mathbf{G}_6$.

Claim 5.8 Let $E_{6,7}$ be the event that $\mathbf{G}_7$ aborts if $H_0'[\hat{m}^*] = 0$, where $\hat{m}^* \leftarrow (\psi, \mathtt{pk}_{\mathtt{ID}^*}, m^*)$. Then $\Pr[E_{6,7}] \leq \frac{q^2}{p}$.

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 H_1 to obtain the corresponding value of $\omega_{\mathtt{ID}}$. The proof of this claim follows in a similar way as the corresponding proof in claim 5.5. ■

Since the games $\mathbf{G}_6$, $\mathbf{G}_7$ are equivalent unless event $\Pr[E_{6,7}]$ occurs, $\mathbf{Adv}^{\mathsf{A}}_{\mathbf{G_6},\mathsf{swal}[\mathsf{ECDSA},\mathsf{H}_0,\mathsf{H}_1]} \leq \mathbf{Adv}^{\mathsf{A}}_{\mathbf{G_7},\mathsf{swal}[\mathsf{ECDSA},\mathsf{H}_0,\mathsf{H}_1]} + \frac{q^2}{p}$.

Reduction to UF-CMA security. We describe an algorithm $\mathsf{C}^{\mathsf{ECDSA},\mathsf{G}}$ (depicted in Figure 16) that plays in the $\mathbf{uf\text{-}cma}_{\mathsf{ECDSA[G]}}$ game. C obtains as input a public key $\mathtt{pk}_{\mathsf{C}}$ and is given access to the signing oracle ECDSASign to obtain signatures under $\mathtt{pk}_{\mathsf{C}}$ under messages of its choice. Furthermore, C has access to the random oracle G. C simulates $\mathbf{G}_7$ to A as described in Figure 16. We elaborate in a bit more detail on the algorithm C below.

Setup. C begins by sampling $\mathtt{St} \xleftarrow{\$} \{0,1\}^{\kappa}$. As shown in Figure 16 (and explained below), C continuously updates the variable St throughout its simulation. As in the proof of Theorem 4.2, C uses the public key $\mathtt{pk}_{\mathsf{C}}$ it obtains from its challenger in Game $\mathbf{uf\text{-}cma}_{\mathsf{ECDSA[G]}}$ as the master public key mpk in its simulation of $\mathbf{G}_7$, i.e., it runs A on input $(\mathtt{mpk} = \mathtt{pk}_{\mathsf{C}}, \mathtt{St})$ in $\mathbf{G}_7$.

Simulation of Public Key Queries. Since $\mathbf{G}_7$ answers a call of the form $\mathsf{PK}(\mathtt{ID})$ from A by computing $\mathtt{pk}_{\mathtt{ID}}$ without using the master secret key msk, C's simulation for this type of query is straight forward and exactly as described above in $\mathbf{G}_7$.

Simulation of Random Oracle Queries. C's simulation of random oracle queries also coincides with the above programming strategy that is already internally present in $\mathbf{G}_7$.

Simulation of Signing Queries. Recall that in $\mathbf{G}_7$, queries of the form $\mathsf{WalSign}(m, \mathtt{ID})$ internally prompts the computation of signature $\sigma' = \mathsf{ECDSASign}^{\mathsf{G}}(\mathtt{msk}, m')$, where $m' \leftarrow ctr$. Since C does not know msk, it needs to compute σ' via a call to its signing oracle, i.e., as $\sigma' \leftarrow \mathsf{ECDSASign}(m')$. Other than that C simulates such a query exactly as internally done for $\mathbf{G}_7$.

Extracting the forgery. When the tuple $(m^*, \sigma^*, \mathtt{ID}^*)$ is returned as an answer from A, C first carries out the computations as described above, i.e., parses it as $(m^*, \sigma^*, \mathtt{ID}^*) = (m^*, (\psi^*, \hat{\sigma}^*), \mathtt{ID}^*)$, checks whether it constitutes a valid forgery, and aborts otherwise (note that in this case, $\mathbf{G}_7$ would return 0, so C can safely abort). In case C does not abort, it recovers $\omega^*_{\mathtt{ID}}$ such that $\mathtt{pk}_{\mathtt{ID}^*} = \mathtt{pk}_{\mathsf{C}} \cdot \omega_{\mathtt{ID}^*}$. C computes $\hat{m}^* \leftarrow (\psi^*, \mathtt{pk}_{\mathtt{ID}^*}, m^*)$ and if $H_0'[\hat{m}^*] = 0$, it aborts. Otherwise, C fetches $m' \leftarrow H_0'[\hat{m}^*]$ and sets $h' \leftarrow \mathsf{G}(m'), \omega^* \leftarrow \frac{H_0[\hat{m}^*]}{h'} \mod p$. Since $H_0[\hat{m}^*] = \mathsf{G}(H_0'[\hat{m}^*]) \cdot \omega_{\mathtt{ID}^*} = \mathsf{G}(m') \cdot \omega_{\mathtt{ID}^*}$, the following holds

- $\mathtt{pk}_{\mathtt{ID}^*} = \mathtt{pk}_{\mathsf{C}} \cdot \omega_{\mathtt{ID}^*}$,
- $\mathsf{ECDSAVer}^{\mathsf{H}_0}(\mathtt{pk}_{\mathtt{ID}^*}, \hat{\sigma}^*, \hat{m}^*) = 1$,
- $\omega^* = \frac{H_0[\hat{m}^*]}{h'} = \frac{\mathsf{G}(m') \cdot \omega_{\mathtt{ID}^*}}{\mathsf{G}(m')} = \omega_{\mathtt{ID}^*}$.

Therefore, $\sigma' \leftarrow \mathsf{Trf}^{\mathsf{G},\mathsf{H}_0}_{\mathsf{ECDSA}}(m', \hat{m}^*, \hat{\sigma}^*, \omega^*, \mathtt{pk}_{\mathsf{C}}, \mathtt{pk}_{\mathtt{ID}^*})$ follows from Lemma 2.5 s.t. $\mathsf{ECDSAVer}^{\mathsf{G}}(\mathtt{pk}_{\mathsf{C}}, \sigma', m') = 1$.

Claim 5.9 (m', σ') constitutes a valid forgery in $\mathbf{uf\text{-}cma}_{\mathsf{ECDSA[G]}}$.

main $\mathsf{C}^{\mathsf{ECDSASign}^{\mathsf{G}},\mathsf{G}}(\mathtt{pk_C})$
00 $ctr \leftarrow 0$
01 $\mathtt{St} \xleftarrow{\$} \{0,1\}^\kappa$ and $\boxed{\mathtt{mpk} \leftarrow \mathtt{pk_C}}$
02 $(m^*, (\psi, \hat{\sigma}) = \sigma^*, \mathtt{ID}^*) \xleftarrow{\$} \mathsf{A}^{\mathsf{H_0,H_1,PK,WalSign}}(\mathtt{mpk}, \mathtt{St})$,
// *where* ψ *may be the empty string*
03 If $\mathtt{SSNKeys}'[\mathtt{ID}^*] = \perp \vee m^* \in \mathtt{Sigs}[\mathtt{ID}^*]$
04 Return 0
05 $(\mathtt{pk_{ID^*}}, \perp, \omega_{\mathtt{ID}^*}) \leftarrow \mathtt{SSNKeys}'[\mathtt{ID}^*]$
06 If $\mathsf{ECWVerify}^{\mathsf{H_0}}(\mathtt{pk_{ID^*}}, \sigma^*, m^*) = 0$
07 Return 0
08 $\hat{m} \leftarrow (\psi, \mathtt{pk_{ID^*}}, m^*)$
09 If $H_0'[\hat{m}] = 0$: Abort
10 $\boxed{m' \leftarrow H_0'[\hat{m}]}$
11 $\boxed{\sigma' \leftarrow \mathsf{Trf}^{\mathsf{G,H_0}}_{\mathsf{ECDSA}}(m', \hat{m}, \hat{\sigma}, \omega_{\mathtt{ID}^*}, \mathtt{pk_C}, \mathtt{pk_{ID^*}})}$
12 Return (m', σ')

Oracle $\mathsf{PK}(\mathtt{ID})$ // *Once per* ID
13 $\mathtt{St}' \leftarrow \mathtt{St}$
14 $(\omega_{\mathtt{ID}}, \mathtt{St}) \leftarrow \mathsf{H}(\mathtt{St}', \mathtt{ID})$
15 $\boxed{\mathtt{pk_{ID}} \leftarrow \mathtt{pk_C} \cdot \omega_{\mathtt{ID}}}$
16 If $(\mathtt{pk_{ID}}, \cdot, \omega_{\mathtt{ID}}) \in \mathtt{SSNKeys}'[\mathtt{ID}]$: Abort
17 If $\exists m = (\cdot, \mathtt{pk_{ID}}, \cdot)$ s.t.: $H_0'[m] = 0$: Abort
18 $\mathtt{SSNKeys}'[\mathtt{ID}] \leftarrow (\mathtt{pk_{ID}}, \perp, \omega_{\mathtt{ID}})$
19 $\mathtt{Sigs}[\mathtt{ID}] \leftarrow \emptyset$
20 Return $\mathtt{pk_{ID}}$

Oracle $\mathsf{H_1}(m)$
21 If $H_1[m] \neq \perp$
22 Return $H_1[m]$
23 $H_1[m] \xleftarrow{\$} \mathbb{Z}_p \times \{0,1\}^\kappa$
24 Return $H_1[m]$

Oracle $\mathsf{WalSign}(m, \mathtt{ID})$
25 If $\mathtt{SSNKeys}'[\mathtt{ID}] = \perp$: Return $\perp$
26 $(\mathtt{pk_{ID}}, \cdot, \omega_{\mathtt{ID}}) \leftarrow \mathtt{SSNKeys}'[\mathtt{ID}]$
27 $\psi \xleftarrow{\$} \{0,1\}^\kappa$
28 $\hat{m} \leftarrow (\psi, \mathtt{pk_{ID}}, m)$
29 If $H_0[\hat{m}] \neq \perp$: Abort
30 Query $\mathsf{H_0}(\hat{m})$
31 $m' \leftarrow H_0'[\hat{m}]$
32 $\boxed{\text{Query oracle } \mathsf{ECDSASign}^{\mathsf{G}} \text{ on } m' \text{ and obtain } \sigma'}$
33 $\hat{\sigma} \leftarrow \mathsf{Trf}^{\mathsf{H_0,G}}_{\mathsf{ECDSA}}(\hat{m}, m', \sigma', \omega_{\mathtt{ID}}^{-1}, \mathtt{pk_{ID}}, \mathtt{pk_C})$
34 $\mathtt{Sigs}[\mathtt{ID}] \leftarrow \mathtt{Sigs}[\mathtt{ID}] \cup \{m\}$
35 Return $(\psi, \hat{\sigma})$

Oracle $\mathsf{H_0}(m)$
36 If $H_0[m] \neq \perp$
37 Return $H_0[m]$
38 Parse m as $(\cdot, \mathtt{pk_{ID}}, \cdot)$ with $\boxed{\omega_{\mathtt{ID}} = pk_{\mathtt{ID}} \cdot \mathtt{pk_C}^{-1}}$
39 If $(\omega_{\mathtt{ID}}, \cdot) \in H_1$
40 $ctr = ctr + 1$
41 $h' \leftarrow \mathsf{G}(ctr)$
42 $H_0[m] \leftarrow \omega_{\mathtt{ID}} \cdot h' \mod p$
43 $H_0'[m] \leftarrow m'$
44 Else
45 $h \xleftarrow{\$} \mathbb{Z}_p$
46 $H_0[m] \leftarrow h$
47 $H_0'[m] \leftarrow 0$
48 Return $H_0[m]$

Figure 16: Reduction to UF-CMA game.

Proof. We have to show that the query $\mathsf{ECDSASign}(m')$ was not made by C during its simulation and hence (m', σ') is a valid forgery in $\mathbf{uf\text{-}cma}_{\mathsf{ECDSA[G]}}$. During C's simulation, C makes a query of the form $\mathsf{ECDSASign}(\cdot)$ when simulating a query $\mathsf{WalSign}(\cdot,\cdot)$ to A. Note that A has not made the query of the form $\mathsf{WalSign}(m^*, \mathtt{ID}^*)$ throughout the simulation. Namely, if it had, $(m^*, \sigma^*, \mathtt{ID}^*)$ is not a valid forgery in $\mathbf{G_7}$ and the simulation would have aborted at this point. This implies that C never had to simulate a query $\mathsf{WalSign}(m^*, \mathtt{ID}^*)$ to A, which entails a $\mathsf{H_0}$ query on message $\hat{m}^* \leftarrow (\psi^*, \mathtt{pk_{ID^*}}, m^*)$. By construction of $\mathsf{H_0}$, message m' is uniquely determined as $m' \leftarrow ctr$. Hence, m' associated with query $\mathsf{H_0}(\hat{m}^*)$ is also unique and was never queried by C in any query of the form $\mathsf{WalSign}(m, \mathtt{ID})$ during C's simulation. ■

From claims 5.3-5.8, we have $\mathbf{Adv}^{\mathsf{A}}_{\mathbf{G_0},\mathsf{swal[ECDSA,H_0,H_1]}} \leq \mathbf{Adv}^{\mathsf{A}}_{\mathbf{G_7},\mathsf{swal[ECDSA,H_0,H_1]}} + \frac{4q^2}{p}$.
Since C provides a perfect simulation of $\mathbf{G_7}$ to A, we obtain

$$\mathsf{Adv}^{\mathsf{A}}_{\mathbf{wunf},\mathsf{swal[ECDSA,H_0,H_1]}} = \mathbf{Adv}^{\mathsf{A}}_{\mathbf{G_0},\mathsf{swal[ECDSA,H_0,H_1]}} \leq \mathsf{Adv}^{\mathsf{A}}_{\mathbf{G_7}} + \frac{4q^2}{p} \leq \mathsf{Adv}^{\mathsf{C}}_{\mathbf{uf\text{-}cma},\mathsf{ECDSA[G]}} + \frac{4q^2}{p},$$

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. 6) 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 $\mathtt{ID}$ 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 $\mathtt{ID}$ structure, where the $\mathtt{ID}$ 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 $\mathtt{ID}$. To this end, the hot wallet generates a fresh session public key $pk_{\mathtt{ID}}$ for $\mathtt{ID}$ via $\mathsf{ECPKDer}^{\mathsf{H}_1}$. Then, $\mathtt{ID}$ 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 $\mathtt{ID}$ from $\mathtt{tx}$ to generate the corresponding secret key $sk_{\mathtt{ID}}$ on the cold wallet. Notice, of course, that the values for $\mathtt{ID}$ 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 $\mathtt{ID}$ and add the ciphertext to the transaction that sends money to the address $pk_{\mathtt{ID}}$.

The winning condition of wallet unforgeability. In Figure 9 the adversary wins the game if she manages to output a valid forgery $(\mathtt{pk}_{\mathtt{ID}^*}, \sigma^*, m^*)$ such that $\mathsf{WVerify}(\mathtt{pk}_{\mathtt{ID}^*}, \sigma^*, m^*) = 1$. 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 $\mathtt{pk}_{\mathtt{ID}^*}$ *and* is accepted by the miners. The latter is quite important because there is no reason why in legacy cryptocurrencies, miners should execute the $\mathsf{ECWVerify}$ algorithm of our ECDSA wallet. Fortunately, however, in Bitcoin miners implicitly execute $\mathsf{ECWVerify}$ when verifying transactions, and hence our scheme and its security analysis is compatible with Bitcoin.[8]

Transaction Cost Analysis. To integrate our scheme into Bitcoin, we have to make sure that (a) transactions are salted, (b) they are pre-fixed by the public key $\mathtt{pk}$ 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. A.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 $\mathtt{Satoshi}$ 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[9]. It is easy to see that extra costs are minor compared to standard Bitcoin transactions.

Table 1: Standard vs Randomized Transactions Costs

Transaction Type	Confirmation in next block Fees ($\mathtt{Satoshi}$/ USD)	Confirmation in next 6 blocks Fees ($\mathtt{Satoshi}$/ 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 Acknowledgments

This work was partly supported by the DFG CRC 1119 CROSSING (project S7) and the German Federal Ministry of Education and Research (BMBF) iBlockchain project. Additionally, the first two authors

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

[9]We have used the currency value from [Cur19] timestamped on 14th May, 2019.

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.

We thank Eike Kiltz for preliminary discussions on our model and Hendrik Amler for helping us in the practical evaluations.

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 5.)

[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 26.)

[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.)

[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 32.)

[Lig18b] Lightning RFC BOLT 3. `https://github.com/lightningnetwork/lightning-rfc/blob/master/03-transactions.md`, 2018. (Cited on page 32.)

[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 2, 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 4, 7, 8, 14.)

[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 31.)

[Seg19] Segregated Witness Wallet Development Guide. `https://bitcoincore.org/en/segwit_wallet_dev/`, 2019. (Cited on page 32.)

[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.)

[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 30.)

[Wui17] Pieter Wuille. Bitcoin Improvement Proposal 62. https://github.com/bitcoin/bips/blob/master/bip-0062.mediawiki, 2017. (Cited on page 31.)

[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 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 $\mathtt{BTC}$. When a user Alice with key pair $(\mathtt{pk_A}, \mathtt{sk_A})$ wants to pay x amount of $\mathtt{BTC}$ to Bob having key pair $(\mathtt{pk_B}, \mathtt{sk_B})$, then it first needs to create a Bitcoin transaction. Let us denote this transaction as $\mathtt{tx_{AB}}$. 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 – $\mathtt{pk_B}$. After the transaction $\mathtt{tx_{AB}}$ has been created, it is signed by Alice's secret key $\mathtt{sk_A}$ – as a result, a signature $\sigma_\mathtt{A}$ is generated. Once $\mathtt{tx_{AB}}$ 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 $\mathtt{pk_A}$. This signature generation, verification process in Bitcoin relies on the ECDSA signature scheme [Wik19]. Once $\mathtt{tx_{AB}}$ qualifies as a valid transaction , it is included within a block. After a subsequent number of blocks, transaction $\mathtt{tx_{AB}}$ gets confirmed in the Bitcoin network.

A.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 UTXO*s*, the user wants to spend (analogous with the $5 in the previous example) and a output part – specifying the newly created UTXO*s* 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.

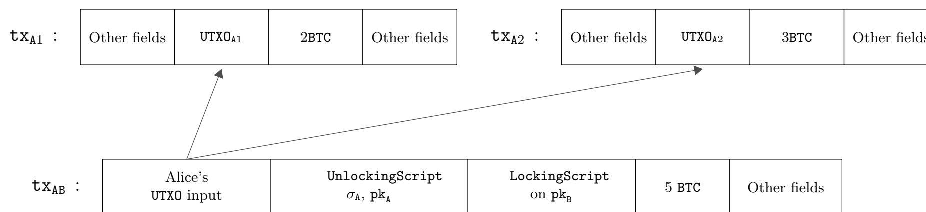

Figure 17: A payment of 5 BTC from Alice $(\mathtt{pk_A}, \mathtt{sk_A})$ to Bob $(\mathtt{pk_B}, \mathtt{sk_B})$ via $\mathtt{tx_{AB}}$.

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 $\mathtt{UTXO_{A1}} = 2\mathtt{BTC}$, $\mathtt{UTXO_{A2}} = 3\mathtt{BTC}$. To pay Bob, Alice creates a new transaction. Let us name this transaction as $\mathtt{tx_{AB}}$ (details in Figure 17). The newly created transaction $\mathtt{tx_{AB}}$ 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 - $\mathtt{UTXO_{A1}}$, $\mathtt{UTXO_{A2}}$.

- The `Unlocking Script` contains a) the signature of the owner of $\mathtt{tx_{AB}}$, i.e. the signature of Alice, computed as $\sigma_{\mathtt{A}} := \mathsf{Sign}(m = \mathsf{H}(\mathtt{tx_{AB}}), \mathtt{sk_A})$. b) The public key of Alice – $\mathtt{pk_A}$, later required for signature verification.

2. Output: The output field may contain a number of so-called `Locking Script`s, 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 $\mathtt{pk_B}$.

`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

  $$\texttt{OP_DUP OP_HASH160} \; < \texttt{hash160(pubKey)} > \; \texttt{OP_EQUAL}$$
  $$\texttt{OP_CHECKSIG}$$

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

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

  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.

  $$\texttt{Script} = \texttt{Sig pubKey OP_DUP OP_HASH160}$$
  $$< \texttt{hash160(pubKey)} > \quad \texttt{OP_EQUAL OP_CHECKSIG}$$

- 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

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

  $$\texttt{ScriptPubKey} = \texttt{OP_HASH160 hash160(scriptHash) OP_EQUAL}.$$

  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

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

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.

A.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 $\psi \xleftarrow{\$} \{0,1\}^{\kappa}$.