Account Management in Proof of Stake Ledgers

Dimitris Karakostas
University of Edinburgh and IOHK
dimitris.karakostas@ed.ac.uk

Aggelos Kiayias
University of Edinburgh and IOHK
akiayias@inf.ed.ac.uk

Mario Larangeira
Tokyo Institute of Technology and IOHK
mario@c.titech.ac.jp

May 5, 2020

Abstract

Blockchain protocols based on Proof-of-Stake (PoS) depend — by nature — on the active participation of stakeholders. If users are offline and abstain from the PoS consensus mechanism, the system's security is at risk, so it is imperative to explore ways to both maximize the level of participation and minimize the effects of non-participation. One such option is stake representation, such that users can delegate their participation rights and, in the process, form "stake pools". The core idea is that stake pool operators always participate on behalf of regular users, while the users retain the ownership of their assets. Our work provides a formal PoS wallet construction that enables delegation and stake pool formation. While investigating the construction of addresses in this setting, we distil and explore *address malleability*, a security property that captures the ability of an attacker to manipulate the delegation information associated with an address. Our analysis consists of identifying multiple levels of malleability, which are taken into account in our paper's core result. We then introduce the first ideal functionality of a PoS wallet's core which captures the PoS wallet's capabilities and is realized as a secure protocol based on standard cryptographic primitives. Finally, we cover how to use the wallet core in conjunction with a PoS ledger, as well as investigate how delegation and stake pools affect a PoS system's security.

1 Introduction

One of Bitcoin's [35] novelties was combination of Proof-of-Work (PoW) with a hash-chain to solve the consensus problem. As shown in subsequent works [25, 26, 36], these elements enable Bitcoin to solve Byzantine Agreement (BA) under open participation. PoW is central in identifying the eligible party that acts at any given time. Specifically, the consensus participants, who generate blocks, are the *miners* which run the PoW mechanism. In turn, the users manage private keys with which they control their assets by signing and publishing transactions on the ledger. PoW-based ledgers observe a decoupling between miners and users, as some miners may not own digital assets and many users don't participate in mining.

The costly nature of PoW though gave rise to alternative mechanisms, most notably Proof-of-Stake (PoS). In PoS, the eligible party, or "block minter", is also a "stakeholder" and is selected proportionally to its stake, i.e. its assets. Stakeholders can arbitrarily join and leave, while also remaining pseudonymous. Thus, the assets of a PoS ledger are dual in nature, acting as both transaction means and participation rights in the consensus protocol.

This fundamental property of PoS systems raises two major considerations. First, using the same key for multiple operations increases its attack surface. For example, using the same key multiple times, e.g. to participate in consensus in a PoS setting, enables quantum attacks, given that most implementations employ non-post-quantum secure signature schemes. Furthermore, frequently using and keeping a key online counters security enhancements like hardware wallets [2]. Second, users need to be constantly online and perform complicated actions. In an environment where the majority of users are often offline and abstain from the protocol's execution, the security guarantees of the ledger are thus weakened.

The above issues are well known. For instance, the usage of multiple keys has been proposed to address the first consideration[1]. Regarding reduced participation, a possible countermeasure is to enable the delegation of participation in the PoS protocol. Users are then organized in "stake pools", i.e. consortiums managed by a single party, the pool's "leader", that runs the PoS protocol as a delegate of the pool's members. Stake pools also bring efficiency advantages, since the set of stake pool leaders is typically smaller than the entire stakeholders' set and thus overall the system can operate with better cost efficiency, since a smaller number of parties have to invest in running a transaction processing service.

PoS systems are increasingly gaining momentum. For instance, Cardano and EOS, both PoS-based systems, are among the top cryptocurrencies by market capitalization[2], while Ethereum [41], the second-biggest blockchain system, slowly transitions to a PoS protocol, Casper [8]. However, the literature lacks a comprehensive and formal treatment of a PoS system's account management. Due to the little systematization of PoS wallets, developers often resort to ad hoc solutions which, as our malleability attack showcases, may be vulnerable. Formalizing the PoS wallet is an important step, since wallets are the gateway through which users interact with a distributed ledger and a core element of consensus itself. Our research fills this gap and aims to act as a guideline for PoS systems' designers, providing a composable design which allows future research to use it in a black-box manner.

Finally, an important motivation is the low level of decentralization in PoS systems. Even in cases where stakeholders are arguably in control, they choose their representatives from a very narrow set of accounts; for instance, the EOS admits only 21 block producers at any given time. Our work aims to alleviate centralization tendencies by enabling every user to either participate on their own or assign their stake to *any* delegate of their choice.

1.1 Our Contributions and Roadmap

Our core contribution is a black-box, composable treatment of PoS wallets, which perform account management in distributed ledgers. Our work aims to be the milestone that allows the discussion and analysis of existing and future wallet designs and implementations.

First, we explore various requirements and distill the desiderata of a PoS wallet (Section 2), on itself a novel contribution. In turn, we describe the malleability attack (Section 3). Malleability is significant because it enables an adversary to artificially inflate its delegated stake, potentially getting financial gains (e.g., in a PoS based financial system). Section 3 shows that malleability is such attack, enabling the adversary to acquire the delegation rights of some assets against the decision of the assets' owner.

A typical example of such adversary is a malicious stake pool leader. A pool leader is a party which receives the delegation rights from other parties and participates in the PoS protocol on their behalf. This adversary would operate under the natural assumption that the reward amount, from the participation in the PoS protocol, is proportional to the stake

[1] For one such discussion see `https://reddit.com/r/ethereum/comments/6idf2c`.

[2] `https://coinmarketcap.com` [April 2020]

delegated to the pool. This assumption is amplified by various incentive mechanisms which mandate that larger stake pools receive greater rewards. Therefore, a successful attack against delegation would enable the pool leader to artificially increase its rewards. Our work shows a generalized malleability attack enables a malicious pool leader to achieve such higher rewards. However, as proven in Theorem 1, as long as malleability is mitigated and the utilized cryptographic primitives are sound, our wallet design is secure.

Our analysis explores various levels of protection against malleability, each offering security and performance guarantees suitable for a wide range of systems. Notably protection against malleability comes with relatively small cost in the size of addresses; specifically, a completely non malleable address scheme amounts to 129 bytes. The severity of malleability is shown via potential threat against Cardano's incentivized testnet, although no malleability attacks have been recorded yet. Interestingly, malleability may be of independent interest in any scheme which combines multiple attributes in a single identifier, for instance Decentralized Identifiers (DIDs) [38].

Our second main contribution is a composable ideal functionality for the core of a Proof-of-Stake wallet (Section 4). Our analysis is based on the UC Framework [9] and is inspired by Canetti [10]. The ideal functionality $\mathcal{F}_{\text{CoreWallet}}$ epitomizes in a concise way a PoS wallet's properties. We realize $\mathcal{F}_{\text{CoreWallet}}$ in the form of the protocol $\pi_{\text{CoreWallet}}$. Both the functionality and the protocol are highly parametric, enabling different address and wallet recovery schemes to be employed. We also describe an address scheme which enables the recovery of the wallet's addresses and their balance given a master key and the ledger in $O(n \log(m))$ time complexity, n being the wallet's addresses and m all addresses in the ledger.

Finally, we combine the core wallet with a PoS ledger to complete staking operations, like stake delegation and the formation of stake pools, as well as payments and block production (in the context of a PoS system). We also analyze the effects of delegation and stake pools on the security of a "vanilla" PoS system; our results confirm that, as long as the stake majority is managed by honest parties, be it pools or stakeholders, a PoS system is secure.

1.2 Related Work

Cryptographic literature has seen a multitude of PoS protocols in the past years. The first provably secure PoS protocol, which initiated a family of such protocols, was Ouroboros [32], followed by Ouroboros Praos [16], Genesis [3], Crypsinous [31], and Hydra [11]. These protocols are similar to Bitcoin, in the sense that they offer eventual guarantees of liveness and persistence, and cover issues including security, privacy, and availability. Another popular PoS protocol is Algorand [12, 27], which employs Byzantine Agreement to achieve the necessary properties of a PoS setting with transaction finality in expected constant time. Similarly, Snow White [6, 37] uses the notion of "robustly reconfigurable consensus", which is specially designed to cope with the lack of participation of users in the consensus protocol. Our work is complementary to these PoS protocols, offering the wallet interface that can be used in conjunction with them to construct a robust and secure PoS system; Appendix D provides an instantiation of our scheme with Ouroboros Praos.

However, formal wallet research has been sparse and limited on PoW. The widely implemented HD Wallet Standard BIP32 [42], based on deterministic wallets [33], was studied by Gutoski and Stebila [30] under partial key leakage. We also employ hierarchical key generation akin to BIP32 for address generation and recovery. Our work also builds on Courtois *et al.* [14], who investigated wallets and key management for Bitcoin [35]. Finally, Arapinis *et al.* [2] analyze specialized hardware wallets under UC, although focused only on PoW.

Various PoS systems employ delegation with varied results. "Delegated PoS", as deployed on Steem [39], EOS [13], and (with some amendments) Tezos [29], enables the voting of

delegates. However, all use a single key for both payment and voting. Also Steem and EOS limit the number of potential delegates to 21 at any moment, while Tezos offers a more open setting, where users can vote for any delegate of their choice, but requires a delegate to own (and lock in a deposit) at least 8.25% of its delegated stake. On the other hand, Cardano[3] (at present) pins all stake to a closed set of block production nodes, i.e. does not offer open participation, while NEO[4] enables only 7 consensus nodes, 5 of which are controlled by a single entity. An alternative approach is taken by Decred [19], which uses a ticketing system, i.e. stakeholders buy a ticket for participation akin to using a separate key. However, like Tezos, it requires the locking of funds, i.e. it does not allow concurrent payments and staking. Our work provides a formal model that, combined with a PoS framework, can help avoid the security, centralization, and usability issues of these systems.

Finally, the game theoretic analysis of reward allocation and stake pool formation is an interesting, though orthogonal, problem. Specifically, our work provides the cryptographic infrastructure, on top of which a reward scheme (e.g. [7, 24]) can be deployed.

1.3 Preliminaries

Definitions and Notation. A ledger records and manages a set of fungible assets. For example, a "satoshi" is the asset maintained by the Bitcoin ledger. The users create **addresses** to interact with the ledger's assets. An **account**, i.e. a set of **addresses**, is denoted by $\Lambda \subseteq \{0,1\}^{p(\lambda)}$ for a given, fixed polynomially bounded function $p(\cdot)$ and the security parameter λ. An **attribute** δ is an object which identifies a property, so an address is associated with a number g of attributes of different types: i) **public** attributes are identifiable and used without any interaction with the account's owner; ii) **semi-public** attributes become public when a transaction that spends from the address is issued; iii) **private** attributes never become public. For instance, Bitcoin addresses comprise of the hash of a verification key. Therefore, the verification key is *semi-public*, its hash is *public*, whereas the private key which signs transactions is *private*. Finally, $|\alpha|$ denotes the length (in bits) of the object α, $A||B$ denotes the concatenation of two objects A and B, and $head(C)$ denotes the last block in a chain C, i.e. $head(C||B) = B$.

Threat Model and Adversarial Motivation. We assume that the adversary *does not* control a majority of the stake; Otherwise attacking the PoS protocol would be trivial. Instead, the adversary aims to inflate its stake by exploiting "staking operations" like delegation as described previously. We also assume that the adversary is "adaptive", i.e. it can corrupt parties on dynamically while the protocol is being executed, and "rushing", i.e. it retrieves and (possibly) delays the honest parties' messages before deciding its strategy.

2 General Desiderata

Before presenting our framework, we first identify the properties that the wallet in a PoS setting should offer. This investigation is an important step in understanding the restrictions in designing such systems, as well as evaluating the choices that a PoS protocol's designer should make, given that, as we show, a number of desirable properties may be conflicting.

In a PoS system each account manages a set of addresses, which own a non-negative amount of cryptocurrency assets. A PoS system should offer at minimum two basic operations for each user's account: i) *paying* and ii) *staking*. Addresses, simply put, are strings

[3] `https://www.cardano.org/`

[4] `https://neo.org`

which have cryptocurrency balances associated with them. They may also contain metadata, in the form of arbitrary attributes, which are useful for particular system operations. We identify the following desiderata for addresses in a PoS setting:

Address Non-malleability: Given an address (and possibly also transactions associated with that address), it should be infeasible for an attacker to construct a different address that shares *only some* of its attributes, most importantly its payment key.

Address Uniqueness: It should be unlikely for the address generation process to produce two equal addresses for different attributes, i.e. the addresses should be unique.

Short Addresses: The addresses should be relatively short, in order to be usable and storage efficient.

Multiple Types of Addresses: It should be possible to construct more than one type of addresses, with each type supporting a different subset of basic operations, e.g. to ban some addresses from staking or delegating to a stake pool.

Multiple Device Support: An account should be able to exist on multiple devices that share *no joint internal state*.

Address Recovery: An account should be able to identify its addresses, given the ledger and the payment keys which it controls.

Privacy and unlinkability: Addresses should be indistinguishable and not publicly linkable to the account which manages them.

An additional and equally important concern in the PoS setting is the "nothing at stake" problem [23]. As opposed to PoW, in PoS a player can easily create blocks that extend multiple parallel chains, an adversarial behavior which diverges from every PoS protocol's rules. Thus, an adversary may profit by attacking the system in this manner, even as they own a large amount of the very assets which they attack. This problem becomes more apparent in the cases of custodian services, which manage assets on behalf of their clients, but assume no financial risk themselves. In this work, we sidestep this problem by introducing "exile" addresses, i.e. addresses excluded from the protocol's execution.

The two basic types of operations, i.e. *payment* and *staking*, can be performed independently by two separate pieces of information, which we denote $\mathcal{I}_p$ and $\mathcal{I}_s$ respectively. The main advantage of this approach is that stake delegation does not require the use of $\mathcal{I}_p$, which is rather reserved only for transferring funds. Another desirable result is the ability to recover all addresses given a master key, e.g. as implemented by HD Wallets [30, 42]. This feature is particularly important in case the equipment which hosts the wallet is lost. We summarize the above, with some additions, as follows:

Account Master Key: There should exist a master key (or seed), that can be used to generate all of the account's management information.

Staking and Payment Separation: Compromising the staking operation should not affect the payment operation (and vice-versa).

Payment Key Information Safety: Apart from a cryptographic hash, no other information about the payment key $\mathcal{I}_p$ should be public prior to issuing a payment.

Key Exposure Mitigation: Ownership of the account's assets and staking ability should be recoverable in case the staking information $\mathcal{I}_s$ is compromised.

Finally, the delegation mechanism boils down to the ability of a user to give the rights over her stake to another user. This action should be distinguishable from other actions, like payment, in order to protect the users and also facilitate automatic reward schemes to be implemented by the ledger. The desiderata for the delegation mechanism are as follows:

Cost Effective Delegation: Stake delegation, as well as changing an account's delegation profile, should be cost effective.

Chain Delegation Restriction: A limit to the number of permitted chain delegation

assignments may be enforced.

Delegation Verification: Participants in the system should be able to verify the status of delegation assignments.

3 Address Malleability

Before designing our PoS wallet, a brief motivation behind researching malleability and its relation to our setting is needed. *Address malleability* is similar to the malleability notions that have been explored in existing literature (cf. [20]) and is intrinsically tied to address generation. Malleability here is observed in the relation between the payment and staking keys. Thus, this family of malleability attacks enables an adversary to construct addresses on behalf of honest users, which may correspond to the honest payment keys and adversarial staking keys. Such addresses are called *forgeries*. A forgery is successful if an honest wallet accepts a forgery as its own and can spend its assets, whereas it fails if it is impossible to send money to it or funds that it owns are unspendable.

We assume two types of adversaries, depending on the information to which they have access. The first is the *network adversary*, who can view the ledger's contents and the addresses produced by an honest party. The second is the *targeting adversary*, who accesses the same information as the network adversary, as well as the semi-public attributes of the addresses of the "victim", i.e. the honest user for which it attempts to produce a forgery. We showcase the different types of adversaries with the following example. Assume a company C that receives regular payments in cryptocurrencies. For security reasons, C stores the private keys of its addresses on an offline server, whereas the public keys are stored online, in order to easily compute and share new addresses with its clients. An adversary that pretends to be a client is a "network" adversary, i.e. may access some of C's addresses. Instead, an adversary that infiltrates the online server and accesses the public keys is "targeting". If the system does not protect against targeting adversaries, then malleability may be used to mount a covert attack, i.e. to obtain the staking rights of C's assets without raising flags.

The formal analysis of malleability takes the form of the malleability predicate M. The predicate returns 1 or 0 to denote whether an address is valid or not. We stress that, in this context, a valid address is either honestly-generated *or* is a successful forgery. The *fully non-malleable* construction is instantiated with the predicate $M_{\mathsf{NM}}^{\mathbb{L},\mathbb{T},P}$ of Algorithm 1, while Appendix A covers the other malleability levels. The predicate verifies that an recipient's address has been generated by some party following the correct process. Upon issuing a transaction (resp. upon verifying), the malleability predicate checks the address of the receiver (resp. sender) to ensure its legitimate and thus if the transaction is acceptable.

Malleability attacks depend on who can mount them and to which extent. We identify these levels, ranging from *full*, with no inherent protection against malleability, to *non-malleable*. These levels are distinguished based on the following properties: i) the adversarial types, i.e. whether the adversary is on the network level or targeting, as described above; ii) "self-verification", i.e. whether a wallet can recognize a forgery for one of its own addresses; iii) "cross-verification", i.e. whether a wallet can recognize a forgery for *any* address.

Although high levels of protection are more desirable, there is a performance trade-off. Specifically, a fully malleable address scheme typically produces short addresses, thus is suitable for applications focused on performance rather than security. In contrast, a security-oriented project would rather aim for the higher levels of malleability protection. Following we briefly describe each level and offer indicative address implementations, assuming that SHA256 is the employed hash function and the signature scheme is ECDSA on secp256r1. We

Algorithm 1 The *fully non-malleable* predicate.

```
function M_NM^{L,T,P}(aux, α)
    switch "aux" do
        case "issue"
            if ∃P' such that ∃l_α : (α, l_α) ∈ L_{P'} then
                return 1
            end if
        case "verify" OR "recover"
            if ∃l_α : (α, l_α) ∈ L_P then
                return 1
            end if
    return 0
end function
```

also assume that each address is associated with two keys, $(\mathsf{vks}, \mathsf{sks})$ for staking and $(\mathsf{vkp}, \mathsf{skp})$ payments, the latter also enabling its recovery; with foresight we note that all schemes are suitable for usage with the core-wallet protocol of Section 4.2.

Level 1, Full Malleability: Address schemes of this level enable forgeries from both network and targeting adversaries. To accept an address, a wallet only checks whether it controls its payment key. For instance, the address is constructed as the concatenation of the hashes of the payment and staking keys: $\alpha = \mathcal{H}(\mathsf{vkp})||\mathcal{H}(\mathsf{vks})$. Thus, given α, an adversary can replace $\mathcal{H}(\mathsf{vks})$ with $\mathcal{H}(\mathsf{vks}')$, for some adversarial key $\mathcal{H}(\mathsf{vks}')$. The length of fully malleable addresses is 64 bytes, i.e. the sum of two hashes.

Level 2, Ex Post Malleability: This level prohibits network adversaries, though targeting adversaries can create successful forgeries. Ex post malleable addresses are constructed as follows: $\alpha = \mathcal{H}(\mathsf{vkp}||\mathsf{vks}||ht)||\mathcal{H}(\mathsf{vks})||ht$. The first part of the address is the hashed root of its attribute tree. The attributes are the payment key vkp, the staking key vks, and the recovery tag ht (for an extended discussion on the generation of recovery tags we refer to Section 4.4 and Appendix C.1). Now, given only α an attacker can no longer replace vks since, in doing so, the first part of the address would be invalidated. However, if the attacker also knows the payment key vkp, it can produce successful forgeries. The length of ex post malleable addresses is 96 bytes, i.e. the sum of three hashes.

Level 3, "Sink" Malleability: Sink malleable address schemes protect against both network and targeting adversaries. Now, a forgery is always rejected by the wallet and its funds are "burnt". Intuitively, imagine transactions as a graph, where the graph's nodes are addresses and its edges are transactions. Forgeries are "sinks", which trap all funds sent to them and no transaction can spend from them. Thus, a wallet can identify forgeries for one of its payment keys, but cannot identify whether an address with a key that it does not own is a forgery. To construct a sink malleable address, the payment key certifies the staking key as follows: $\alpha = \mathcal{H}(\mathsf{vkp})||\mathsf{vks}||\mathsf{Sign}(\mathsf{skp}, \mathsf{vks})$. Clearly, even if an attacker knows the public payment key vkp, it cannot create a forgery, unless it forges a signature. Since the ECDSA signature is 512 bits, the address is 128 bytes.

Level 4, Non-Malleability: Non-malleable address schemes offer the highest level of protection. Now, any party can identify whether *any* address is honestly-generated or is a forgery. Our non-malleable proposal achieves this by utilizing the signature of the sink malleable scheme while also revealing the public payment key: $\alpha = \mathsf{vkp}||\mathcal{H}(\mathsf{vks})||\mathsf{Sign}(\mathsf{skp}, \mathsf{vks})$. Therefore, the address's length is about 129 bytes (33 bytes for the payment key, 32 bytes for the hash, and 64 for the signature). We also refer to Appendix C.2 for a full security

Malleability Level	Network protection	Targeting protection	Self-verification	Cross-verification	Address length (bytes)
1. Full	✗	✗	✗	✗	64
2. Ex post	✓	✗	✗	✗	96
3. Sink	✓	✓	✓	✗	128
4. Non-malleable	✓	✓	✓	✓	129

Table 1: Comparison of the malleability levels, from malleable to fully non-malleable. The comparison is based on the types of adversaries that can successfully forge an address, the ability of a wallet to identify a forgery, and the address length. We note that the non-malleable scheme reveals the public payment key by default, whereas all other schemes reveal only its hash (and the public key is revealed only upon conducting a transaction).

analysis of this scheme.

The user also needs to reveal the keys themselves to conduct payments and staking. In all cases the user reveals the staking key, whereas in all levels except 4 they need to also reveal the payment key, in addition to publishing the address. Thus the overall storage requirement is $|\alpha| + |\mathsf{vkp}| + |\mathsf{vks}|$.

Table 1 summarizes the above comparison. As expected, higher levels of malleability protection induce a higher performance cost, in terms of address length. Additionally, the fully non-malleable scheme reveals the public key by default, which opens the system to quantum attacks, assuming a non-post-quantum secure signature scheme like ECDSA is employed. Future work will explore the usage of non-interactive zero-knowledge proofs of knowledge, which enable the same level of security without revealing the public key.

Address malleability is adjacent, though more severe than, transaction malleability in Bitcoin [18, 1]. The latter enables an attacker to modify a transaction, thus changing its identifier (i.e. its hash), such that the issuer is led to believe that the original transaction is unconfirmed. Although this attack may lead to loss of funds (e.g. as claimed in the Mt. Gox incident [18]), it does not directly compromise the ledger's consensus mechanism, due to the decoupling of the mining and the transaction processes. In contrast, address malleability directly affects a PoS ledger's security, since an attacker can use it to artificially inflate their delegated stake to the extent of (potentially) violating the honest majority assumption.

Finally, we present a real-world case of the address malleability hazard. On December 2019, Cardano released its Incentivized Testnet[5], a testnet aimed at enabling Cardano stakeholders to create stake pools and delegate their stake on a preliminary level. In accordance to our setting, addresses are associated with two keys, for payments and staking operations, nevertheless addresses are fully malleable; specifically, they are constructed as the (Bech32-encoded) concatenation of the payment and staking keys[6], as in our malleable scheme. Although no real-world attacks have been recorded yet, our paper raises awareness about their possibility and the mitigation strategies that are available for practical systems.

[5] `https://staking.cardano.org/` [April 2020]

[6] The Rust code of Cardano's testnet node is open source on Github. Address construction and key concatenation can be found at `https://git.io/Jvppa` and `https://git.io/Jvppr`.

4 The Core Proof-of-Stake Wallet

We now focus on defining the core of a PoS wallet. With malleability in mind, we first build the Core-Wallet ideal functionality, i.e. the security definition of the basic key-management operations that PoS wallet should complete. Following, we realize the functionality via a protocol which can be constructed using standard cryptographic primitives. The Core-Wallet protocol employs a number of sub-routines and, as we show in Theorem 1, securely realizes the ideal functionality, as long as a set of standard assumptions are ensured. By the end of the section we briefly discuss the construction of addresses and wallet recovery, i.e. the identification of the wallet's addresses and its balance given its private keys and the public ledger. Firstly though, let us outline the core wallet's inner workings:

Initialization: The wallet is bootstrapped during the initialization phase. In the ideal functionality $\mathcal{F}_{\text{CoreWallet}}$, various global lists are initialized. In the protocol $\pi_{\text{CoreWallet}}$, at this stage the master secret key is created, i.e. the wallet's *seed* which will be used to generate all keys and addresses; $\pi_{\text{CoreWallet}}$ also maintains lists for the wallet's keys.

Address Generation: During address generation, $\mathcal{F}_{\text{CoreWallet}}$ leaks to the adversary the new address; in fact the adversary chooses the address, which is then checked by $\mathcal{F}_{\text{CoreWallet}}$ for validity, e.g. to ensure that it is unique. This choice implies that the adversary knows the addresses of the wallet, i.e. there is no address privacy; future versions of the core wallet will aim to provide privacy guarantees. The address is then linked with a list of attributes, i.e. the payment and staking keys and helper attributes like the recovery tag, and are stored in the global arrays. In $\pi_{\text{CoreWallet}}$, address generation is done by the helper function $\mathsf{GenAddr}$, which is given the child attributes which are generated using the master key.

Wallet Recovery: Wallet recovery is the process of retrieving the recovery tags of the wallet's first i addresses. In $\mathcal{F}_{\text{CoreWallet}}$ these attributes are maintained in a list, whereas in $\pi_{\text{CoreWallet}}$ they are reconstructed using the master key and corresponding indices.

Address Recovery: This interface enables a wallet to verify whether an address α was among its first i constructed addresses; the verification process is as above. Interestingly, during address recovery the malleability predicate is also used; thus, depending on the level of malleability that we want to achieve, at this point the wallet may accept forgeries as valid.

Issue Transaction: To issue a transaction, the wallet receives the sender and receiver addresses and the amount of assets to be transferred. Following the UC model of signatures [10], it retrieves from the adversary the signature object and, after a few checks (including for malleability), registers the signed transaction. Thus, although the adversary can partially affect the signature's structure, $\mathcal{F}_{\text{CoreWallet}}$ ensures that security is maintained. On the protocol's side, transaction issuing is achieved by simply signing the transaction with the payment key that corresponds to the sender's address.

Verify Transaction: Transaction verification is also similar to signature verification. Specifically, $\mathcal{F}_{\text{CoreWallet}}$ ensures that i) the malleability level is preserved (using the predicate), and ii) completeness, unforgeability, and consistency are ensured, i.e. a secure digital signature scheme's needed properties. On the other hand, $\pi_{\text{CoreWallet}}$ simply uses the Verify algorithm of the employed signature scheme to check the transaction's signature's validity.

Issue and Verify Staking: Issuing and verification of staking actions, e.g. the certificates which we explore in Section 5, is achieved similarly to payment transactions, with the usage of staking keys instead of payment keys.

4.1 The Core-Wallet Functionality

The goal of the ideal functionality is to distill in a concise way the necessary properties of a PoS wallet and provide a formal model as a base of discussion. The ideal functionality

$\mathcal{F}_{\text{CoreWallet}}$, inspired by Canetti [10], interacts with the ideal adversary $\mathcal{S}$ and a set of parties denoted by $\mathbb{P}$ and is parameterized by the malleability predicate $M(\cdot,\cdot,\cdot) \to \{0,1\}$. It also keeps the, initially empty, lists S of staking actions and $\mathcal{T}$ of transactions. Without loss of generality, we assume that, given a list of attributes $l_{\alpha,Gen} = (\delta_1, \ldots, \delta_g)$, δ_1 is the staking key's information and δ_2 is the recovery tag, which is further investigated in Appendix C.1. We note that the functionality distinguishes the addresses in three types, the "base", "pointer", and "exile" addresses, each with a specific utility. Briefly, base addresses bootstrap a wallet, pointer addresses are shorter, and exile addresses help bypass the nothing-at-stake problem described in Section 2.

Functionality $\mathcal{F}^{M}_{\text{CoreWallet}}$

Initialization: Upon receiving $(\textsc{Init}, sid)$ from $P \in \mathbb{P}$, forward it to $\mathcal{S}$ and wait for $(\textsc{InitOk}, sid)$. Then initialize the empty lists L_P of addresses and attribute lists and K_P of staking keys, and send $(\textsc{InitOk}, sid)$ to P.

Address Generation: Upon receiving $(\textsc{GenerateAddress}, sid, aux)$ from $P \in \mathbb{P}$, forward it to $\mathcal{S}$. Upon receiving $(\textsc{Address}, sid, \alpha, l_\alpha)$ from $\mathcal{S}$, parse l_α as $(\delta_1, \ldots, \delta_g)$ and $\forall P' \in \mathbb{P}$ check if $\forall(\alpha', (\delta'_1, \ldots, \delta'_g)) \in L_{P'}$ it holds that $\alpha \neq \alpha'$, $\delta'_2 \neq \delta_2$, and $\forall j \in [i, \ldots, g] : \delta'_j \neq \delta_j$, i.e. the address, recovery tag, and private attributes are unique. If so, then:

- if $aux =$ ("base"), check that $\forall(\alpha', (\delta'_1, \ldots, \delta'_g)) \in L_P : \delta'_1 \neq \delta_1$,
- else if $aux =$ ("pointer", vks), check that $\delta_1 = \mathsf{vks}$,
- else if $aux =$ ("exile"), check that $\delta_1 = \perp$.

If the checks hold or P is corrupted, then insert (α, l_α) to L_P and return $(\textsc{Address}, sid, \alpha)$ to P. If $aux =$ ("base") also insert δ_1 to K_P and return $(\textsc{StakingKey}, sid, \delta_1)$ to P.

Wallet Recovery: Upon receiving $(\textsc{RecoverWallet}, sid, i)$ from $P \in \mathbb{P}$, for the first i elements in L_P return $(\textsc{Tag}, sid, \delta_2)$.

Address Recovery: Upon receiving $(\textsc{RecoverAddr}, sid, \alpha, i)$ from P, if (α, l) is one of the first i elements of L_P or $M(L_P, \text{"recover"}, \alpha) = 1$, return $(\textsc{RecoveredAddr}, sid, \alpha)$.

Issue Transaction: Upon receiving $(\textsc{Pay}, sid, \Theta, \alpha_{\mathsf{s}}, \alpha_{\mathsf{r}}, m)$ from $P \in \mathbb{P}$, if $\exists l_\alpha : (\alpha_{\mathsf{s}}, l_\alpha) \in L_P$ forward it to $\mathcal{S}$. Upon receiving $(\textsc{Transaction}, sid, tx, \sigma)$ from $\mathcal{S}$, such that $tx = (\Theta, \alpha_{\mathsf{s}}, \alpha_{\mathsf{r}}, m)$, check if $\forall(tx', \sigma', b') \in \mathcal{T} : \sigma' \neq \sigma$, $(tx, \sigma, 0) \notin \mathcal{T}$, and $M(L_P, \text{"issue"}, \alpha_r) = 1$. If all checks hold, then insert $(tx, \sigma, 1)$ to $\mathcal{T}$ and return $(\textsc{Transaction}, sid, tx, \sigma)$.

Verify Transaction: Upon receiving $(\textsc{VerifyPay}, sid, tx, \sigma)$ from $P \in \mathbb{P}$, with $tx = (\Theta, \alpha_{\mathsf{s}}, \alpha_{\mathsf{r}}, m)$ for a metadata string m, forward it to $\mathcal{S}$ and wait for a reply message $(\textsc{VerifiedPay}, sid, tx, \sigma, \phi)$. Then:

- if $M(L_P, \text{"verify"}, \alpha_s) = 0$, set $f = 0$
- else if $(tx, \sigma, 1) \in \mathcal{T}$, set $f = 1$
- else, if P is not corrupted and $(tx, \sigma, 1) \notin \mathcal{T}$, set $f = 0$ and insert $(tx, \sigma, 0)$ to $\mathcal{T}$
- else, if $(\Theta, \alpha_s, \alpha_r, m, \sigma, b) \in \mathcal{T}$, set $f = b$
- else, set $f = \phi$.

Finally, send $(\textsc{VerifiedPay}, sid, tx, \sigma, f)$ to P.

Issue Staking: Upon receiving $(\textsc{Stake}, sid, stx)$ from P, such that $stx = (\mathsf{vks}, m)$ for a metadata string m, forward the message to $\mathcal{S}$. Upon receiving $(\textsc{Staked}, sid, stx, \sigma)$ from $\mathcal{S}$, if $\forall(stx', \sigma', b') \in S : \sigma' \neq \sigma$, $(stx, \sigma, 0) \notin S$, and $\mathsf{vks} \in K_P$, add $(stx, \sigma, 1)$ to S and return $(\textsc{Staked}, sid, stx, \sigma)$ to P.

Verify Staking: Upon receiving $(\textsc{VerifyStake}, sid, stx, \sigma)$ from $P \in \mathbb{P}$, forward it to $\mathcal{S}$ and wait for $(\textsc{VerifiedStake}, sid, stx, \sigma, \phi)$, with $stx = (\mathsf{vks}, m)$. Then find P_{s}, such that $\mathsf{vks} \in K_{P_{\mathsf{s}}}$, and:

- if $(stx, \sigma, 1) \in S$, set $f = 1$

- else if P_{s} is not corrupted and $(stx, \sigma, 1) \notin S$, set $f = 0$ and insert $(stx, \sigma, 0)$ to S
- else if exists an entry $(stx, \sigma, f') \in S$, set $f = f'$
- else set $f = \phi$ and insert (stx, σ, ϕ) to S.

Finally, return $(\text{VERIFIEDSTAKE}, sid, stx, \sigma, f)$ to P.

We remark that, although $\mathcal{F}_{\text{CoreWallet}}$ satisfies our requirements, it does not offer any type of *forward security* in the sense of Bellare and Miner [5]. Thus, it does not fit protocols which require stronger security guarantees, e.g. Ouroboros Praos [16], which relies (among other primitives) on a forward secure digital signature scheme to provide security guarantees against fully-adaptive corruption in a semi-synchronous setting. Additionally, it does not cover protocols that allow arbitrary parties to operate the address generation interface, instead of restricting it to the wallet's owner, e.g. Cryptonote [40].

4.2 The Core-Wallet Protocol

We now introduce the core-wallet protocol $\pi_{\text{CoreWallet}}$, which realizes $\mathcal{F}_{\text{CoreWallet}}$. $\pi_{\text{CoreWallet}}$ abstracts the address generation process and is parameterized with a number of helper functions. parsePubAttrs returns an address's public attributes $[\mathsf{vks}, wrt, aux]$. $\mathsf{HKeyGen}$ and $\mathsf{RTagGen}$ produce the child key pair and the recovery tag respectively. We also assume a signature scheme $\Sigma = \langle \mathsf{KeyGen}, \mathsf{Verify}, \mathsf{Sign} \rangle$ (cf. [28]). $\pi_{\text{CoreWallet}}$ interacts with $\mathcal{P}_o$, i.e. the wallet's owner, and maintains the, initially empty, lists PK of payment keys and SK of staking keys. Theorem 1, a core result of this paper, proves $\pi_{\text{CoreWallet}}$'s security w.r.t. $\mathcal{F}_{\text{CoreWallet}}$. For readability purposes, we drop the generic notation δ and instead use names representative of each attribute; the staking and the payment keys are now $(\mathsf{vks}, \mathsf{sks})$ and $(\mathsf{vkp}, \mathsf{skp})$ respectively. The public attributes $d = [\mathsf{vks}, wrt, aux]$ comprise of the public staking key, the recovery tag, and the address's auxiliary information, which identifies its type.

Protocol $\pi_{\text{CoreWallet}}$

Initialization: Upon receiving (INIT, sid) from $\mathcal{P}_o$, set $msk \xleftarrow{\$} \{0,1\}^\lambda$ and return (INITOK, sid) to $\mathcal{P}_o$.

Address Generation: Upon receiving $(\text{GENERATEADDRESS}, sid, aux)$ from $\mathcal{P}_o$, compute the index and "child" attributes as follows: i) pick $i \leftarrow \mathcal{I}$; ii) compute $(\mathsf{vkp}_c, \mathsf{skp}_c) = \mathsf{HKeyGen}(\langle msk, \text{"payment"}, i \rangle)$; iii) compute $wrt = \mathsf{RTagGen}(\mathsf{vkp}_c)$. If $aux = (\text{"base"})$ compute $(\mathsf{vks}_c, \mathsf{sks}_c) = \mathsf{HKeyGen}(\langle msk, \text{"staking"}, i \rangle)$, else if $aux = (\text{"pointer"}, \mathsf{vks})$ then find $(\mathsf{vks}_c, \mathsf{sks}_c) \in K : \mathsf{vks} = \mathsf{vks}_c$, else if $aux = (\text{"exile"})$ set $(\mathsf{vks}_c, \mathsf{sks}_c) = (\perp, \perp)$. Then insert the list $l_\alpha = \langle \mathsf{vks}_c, wrt, aux, \mathsf{vkp}_c, \mathsf{skp}_c, \mathsf{sks}_c \rangle$ to L, generate the new address α as $\alpha = \mathsf{GenAddr}(\langle aux, \mathsf{vks}_c, \mathsf{vkp}_c, wrt \rangle)$, and insert the tuple $\langle \alpha, (\mathsf{vkp}_c, \mathsf{skp}_c) \rangle$ to PK. Then return $(\text{ADDRESS}, sid, \alpha)$ to $\mathcal{P}_o$. If $aux =$ "base" also insert $(\mathsf{vks}_c, \mathsf{sks}_c)$ to SK and send the message $(\text{STAKINGKEY}, sid, \mathsf{vks}_c)$ to $\mathcal{P}o$.

Wallet Recovery: Upon receiving the message $(\text{RECOVERWALLET}, sid, i_{max})$ from $\mathcal{P}_o$, $\forall i \in \mathcal{I} : i < i_{max}$ set $(\mathsf{vkp}_i, \mathsf{skp}_i) = \mathsf{HKeyGen}(\langle msk, \text{"payment"}, i \rangle)$ and return $(\text{TAG}, sid, \mathsf{RTagGen}(\mathsf{vkp}_i))$.

Address Recovery: Upon receiving $(\text{RECOVERADDR}, sid, \alpha, i_{max})$ from $\mathcal{P}_o$, parse the address's attributes $(\mathsf{vks}, wrt, aux) = parsePubAttrs(\alpha)$. If exists $i \in \mathcal{I} : i < i_{max}$, where $(\mathsf{vkp}_i, \mathsf{skp}_i) = \mathsf{HKeyGen}(\langle msk, \text{"payment"}, i \rangle)$ and $\mathsf{RTagGen}(\mathsf{vkp}_i) = wrt$, return $(\text{RECOVEREDADDR}, sid, \alpha)$.

Issue Transaction: Upon receiving $(\text{PAY}, sid, \Theta, \alpha_s, \alpha_r, m)$ from $\mathcal{P}_o$, find an entry $\langle \alpha_s, (\mathsf{vkp}, \mathsf{skp}) \rangle \in PK$ and send $(\text{TRANSACTION}, sid, tx, \mathsf{Sign}(\mathsf{skp}, tx))$ to $\mathcal{P}_o$, such that $tx = (\Theta, \alpha_s, \alpha_r, m)$.

Verify Transaction: Upon receiving a message $(\text{VERIFYPAY}, sid, tx, \sigma)$ from $\mathcal{P}_o$, with

$tx = (\Theta, \alpha_s, \alpha_r, m)$ for some metadata string m, find an entry $\langle \alpha_s, (\mathsf{vkp}, \mathsf{skp}) \rangle$ in PK and return (VERIFIEDPAY, $sid, tx, \sigma, \mathsf{Verify}(tx, \sigma, \mathsf{vkp})$) to $\mathcal{P}_o$.
Issue Staking: Upon receiving (STAKE, sid, stx) from $\mathcal{P}_o$ such that $stx = (\mathsf{vks}, m)$, find $(\mathsf{vks}, \mathsf{sks}) \in SK$ and return (STAKED, $sid, stx, \mathsf{Sign}(\mathsf{sks}, stx)$).
Verify Staking: Upon receiving a message (VERIFYSTAKE, sid, stx, σ) from party $\mathcal{P}_o$, where $stx = (\mathsf{vks}, m)$ for some metadata m, find an entry $(\mathsf{vks}, \mathsf{sks}) \in SK$ and return (VERIFIEDSTAKE, $sid, stx, \sigma, \mathsf{Verify}(stx, \sigma, \mathsf{sks})$).

4.3 Security of the Core-Wallet Protocol

Similar to $\mathcal{F}_{\text{CoreWallet}}$, $\pi_{\text{CoreWallet}}$ accommodates various address schemes and different levels of malleability. Next we present the definitions of the properties needed to prove its security, while Theorem 1 proves that $\pi_{\text{CoreWallet}}$ securely realizes $\mathcal{F}_{\text{CoreWallet}}$ assuming its subprocesses satisfy these definitions. The proof (Appendix B.4) is heavily based on Canetti [10] for the signature's properties, while the address properties (cf. Definitions 1, 2, 3, 4) cover malleability and address generation.

Definition 1 (Collision resistance)**.** *A function H is collision resistant if, given $h \leftarrow \{0,1\}^l$, it should be computationally infeasible for a probabilistic polynomial algorithm to find a value x such that $h = \mathcal{H}(x)$.*

Definition 2 (Address collision resistance)**.** *Analogously to hash functions,* $\mathsf{GenAddr}$ *is collision resistant when it is infeasible to produce two different attribute lists $l_i = (\delta_1^i, \ldots, \delta_g^i)$ for $i \in \{1, 2\}$, i.e. they differ in at least one attribute like $\exists j \in [1, g]$: $\delta_j^1 \neq \delta_j^2$, such that $\mathsf{GenAddr}(l_1) = \mathsf{GenAddr}(l_2)$, after running $\mathsf{GenAddr}(\cdot)$ a polynomial number of times.*

Definition 3 (Hierarchical Key Generation)**.** *Given a key generation function* $\mathsf{HKeyGen}(\cdot)$ *and a signature scheme $\Sigma = \langle \mathsf{KeyGen}, \mathsf{Verify}, \mathsf{Sign} \rangle$, $\mathsf{HKeyGen}(\cdot)$ is hierarchical for Σ if, for all i, the distribution of keys produced by $\mathsf{HKeyGen}(i)$ is computationally indistinguishable from the distribution of keys produced by* KeyGen.

Definition 4 (Non-malleable attribute address generation)**.** *Let $\mathcal{L}$ be a distribution of attribute lists and $l \leftarrow \text{DOM}(\mathcal{L})$ an attribute list, such that $\text{DOM}(\mathcal{L}) = \Delta_1 \times \ldots \times \Delta_g$. Let the first attributes of l $\delta_1, \ldots, \delta_i$ relate to a property over which we define non-malleability. Given an address α, it is infeasible for an adversary $\mathcal{A}$ to produce valid forgeries, i.e. acceptable addresses with the same payment key as α, without access to α's private attributes,* even with access to the address's metadata, *i.e. its semi-public attributes. Concretely, with* Addrs *the list of addresses queried by $\mathcal{A}$ to the oracle $\mathsf{GenAddr}^d(\cdot)$, it holds that:*

$$\Pr\left[\begin{array}{c} l = (\delta_1, \ldots, \delta_g),\ \alpha \leftarrow \mathsf{GenAddr}(l), \\ \mathcal{A}^{\mathsf{GenAddr}^d(\cdot), \mathsf{GenMeta}^d(\cdot)}(\delta_i, \ldots, \delta_g) \rightarrow (\alpha', \delta_i', \ldots, \delta_g') \\ (\mathsf{GenAddr}^d(\delta_i', \ldots, \delta_g') = \alpha') \wedge\ (\alpha' \neq \alpha) \wedge\ (\alpha' \notin \mathsf{Addrs}) \end{array} : \right] \leq negl(\lambda)$$

for probabilities over $\mathsf{GenAddr}$*'s randomness, the PPT adversary $\mathcal{A}$, and l.*

Theorem 1. *Let the generic protocol $\pi_{\text{CoreWallet}}$ be parameterized by a signature scheme $\Sigma = \langle \mathsf{KeyGen}, \mathsf{Verify}, \mathsf{Sign} \rangle$ and the* $\mathsf{RTagGen}$, $\mathsf{HKeyGen}$, *and* $\mathsf{GenAddr}$ *functions. Then $\pi_{\text{CoreWallet}}$ securely realizes the ideal functionality $\mathcal{F}_{\text{CoreWallet}}^{M_{\mathsf{SM}}}$ if and only if Σ is EUF-CMA,* $\mathsf{GenAddr}$ *is collision resistant and attribute non-malleable (cf. Definitions 2 and 4),* $\mathsf{RTagGen}$ *is collision resistant (cf. Definition 1), and* $\mathsf{HKeyGen}$ *is hierarchical for Σ (cf. Definition 3).*

4.4 Address Construction and Wallet Recovery

Following the protocol's definition, we now provide a brief overview of the address generation and recovery. A detailed discussion on address construction, including a sink malleable scheme, which can be used in conjunction with $\pi_{\text{CoreWallet}}$, is available on Appendix C.

For the construction of a new, "child" address we utilize a hierarchical address generation scheme (cf. hierarchical deterministic wallets [33]). The child key, which is produced by the master key and an index i, is given to the address generation function GenAddr, which outputs the new address. The wallet produces three types of addresses with different staking information. The stake of a **base** address is controlled by a new staking key. A **pointer** address's staking key is associated with an existing key which is published in a past transaction in the ledger. Finally, an **exile** address is not associated with any staking key, thus cannot perform staking and its assets are automatically removed from the PoS protocol's execution.

Each address also contains the recovery tag wrt, a public parameter which allows the identification of addresses. The tag is created by the function RTagGen and links the address to its attributes without revealing its semi-public attributes, e.g. the payment key. During recovery, a wallet retrieves from the ledger its addresses and their balance, using its master key. Specifically, we assume a well-defined list of domains, each having a finite cardinality. The wallet initially picks indexes from the first domain, constructs the recovery tag for each, and compares it to the ledger's addresses. After identifying all addresses from the indexes of the first domain, it proceeds with the second and so forth. If, for some domain, no index is associated with a ledger's address, recovery halts. The complexity of recovery is $O(n \log(m))$, n being the number of the wallet's addresses and m the number of the ledger's addresses.

5 Integration of the Core-Wallet with PoS Consensus

Combining the core wallet with a Proof-of-Stake ledger we can build a full PoS wallet. We abstract the PoS ledger in the following functions and properties:

i) $F_\theta(\cdot)$: given an address α, F_θ returns the assets that α owns
ii) $F_{\Phi,tx}(\cdot)$: $F_{\Phi,tx}$ outputs the fees Φ of publishing a transaction on the ledger
iii) $F_{otx}(\cdot)$: given an address, F_{otx} outputs its outgoing transactions
iv) Φ_{reg}: the cost of stake pool registration
v) α_{reg}: a special address that pertains to pool registration
vi) $F_{PoS,player}(\cdot)$: given a chain $\mathcal{C}$, $F_{PoS,player}$ outputs the next PoS participant

The above take various forms in distributed ledgers. Given the existence of various flavors of PoS protocols (cf. Section 1.2), we can this model design a wallet that is generic enough; Appendix D also describes the interaction with a specific PoS protocol, Ouroboros Praos [17]. Following, we explore how payments and stake-related actions on a PoS ledger are conducted via the core wallet. Also we evaluate the impact of stake pools on the execution and security assumptions of a typical PoS protocol, various modes of operation for enhanced safety and privacy, as well as two noteworthy attack vectors.

5.1 The PoS Wallet's Actions

A PoS wallet performs two types of actions, *payment* and *staking*. Although payment, i.e. the transfer of assets between accounts, is straightforward, staking depends on the inner-workings of each PoS protocol, so here we cover only delegation and stake pool formation.

Payment: Payment is the transfer of Θ assets from a sender's address α_s to a receiver's address α_r. The wallet creates the transaction $tx = (\Theta, \alpha_s, \alpha_r, m)$, with metadata m like

the amount of fees or the change address. Next it accesses the PAY interface of $\pi_{\text{CoreWallet}}$, retrieves the signed transaction, and publishes it on the ledger.

Stake pool registration: A main staking operation that the wallet enables is the formation of stake pools, which participate in the PoS protocol on behalf of stakeholders. Every pool is identified by a registered staking key. Before registering the key, the wallet first uses the address generation interface of $\pi_{\text{CoreWallet}}$ to produce a new staking key $(\mathsf{vks}, \mathsf{sks})$. It then creates a registration certificate $R = (\mathsf{vks}, m)$, where m is the pool's metadata, e.g. the name of the pool's leader. The certificate is passed to the STAKE interface of the core protocol, in order to sign it. Finally, the signed certificate is published on the ledger as part of the metadata of a payment, thus registering the key vks on behalf of the pool.

Delegation: Delegation enables a stakeholder to commission a stake pool to perform staking on its behalf. It is also based on certificates, like $\Sigma = (\mathsf{vks}_s, \langle \mathsf{vks}_d, m \rangle)$. vks_s is the staking key to which the certificate applies, i.e. the key which controls the stake of an address, vks_d is the delegate's key, i.e. the registered key of a stake pool, and m is the certificate's metadata. Again the wallet gives Σ to the STAKE interface of $\pi_{\text{CoreWallet}}$ and then publishes the signed certificate via a payment transaction's metadata, similar to stake pool registration. When a staking key issues a delegation certificate, *all* addresses which are associated with it are re-delegated accordingly. For instance, if a pointer address points to a key vks_s, then the latest certificate issued by this key takes effect. Finally, a stake pool re-delegates via a lightweight certificate, i.e. a certificate not published via a transaction but included in the block's headers; Appendix E further explores stake re-delegation.

5.2 Participation in the PoS Protocol

PoS participation consists of regularly publishing blocks to extend a chain. A block $\mathcal{B}$ is a tuple (vks, m), where $(\mathsf{vks}, \mathsf{sks})$ is the key signing $\mathcal{B}$ and m are $\mathcal{B}$'s contents. As with the other staking actions, the wallet obtains a block's signature via the staking interface (and a verifier similarly checks it). However, a block's validity, i.e. whether it can extend a given chain, depends on the chain decision rules, which are affected by delegation as shown next.

Given its local chain $\mathcal{C}$, a player retrieves the address which is eligible to produce a block as $\alpha_{PoS} = F_{PoS,player}(\mathcal{C})$. α_{PoS} has staking key vks_{PoS}. A block $\mathcal{B}$ signed by a different staking key $(\mathsf{vks}, \mathsf{sks})$ is valid for $\mathcal{C}$ if i) C contains a certificate that delegates from vks_{PoS} to vks or ii) either vks_{PoS} has not delegated or a certificate that delegates from vks_{PoS} to vks_h is published in $\mathcal{C}$ and $\mathcal{B}$ contains a lightweight certificate delegating from vks_h to vks; otherwise $\mathcal{B}$ is invalid. Additionally, the empty chain $\mathcal{C} = \epsilon$ is valid and, given a chain $\mathcal{C} = \mathcal{C}'||B$, if $\mathcal{C}'$ is valid and B is valid for $\mathcal{C}'$, then $\mathcal{C}$ is valid. A chain is picked between the previously accepted chain $\mathcal{C}$ and the set of *valid* chains $\mathbb{C}$ available on the network: the longest chain from $\mathbb{C} \cup \{\mathcal{C}\}$ is chosen. In case of length tie between two chains $\mathcal{C}_1$ and $\mathcal{C}_2$, where $head(\mathcal{C}_1)$ and $head(\mathcal{C}_2)$ are signed by $(\mathsf{vks}_1, \mathsf{sks}_1)$ and $(\mathsf{vks}_2, \mathsf{sks}_2)$ respectively, the following rules apply:

- if vks_1 and vks_2 are delegated via two heavyweight certificates, then choose the certificate with the higher index
- else if vks_1 is delegated via the heavyweight certificate Σ_1 and vks_2 is delegated via the *lightweight* certificate Σ_2, then $\mathcal{C}_1$ is chosen
- else if vks_1 and vks_2 are delegated via the combination of heavyweight and lightweight certificates $(\Sigma_{1,1}, \Sigma_{1,2})$ and $(\Sigma_{2,1}, \Sigma_{2,2})$ respectively, then:
  - if they have different indexes, choose the one with the higher index
  - else if they have different counters, choose the one with the higher counter
  - else choose the first observed on the network.

5.3 Consensus Security under Stake Pools

To argue about the security of a PoS stake-pooled variant we turn to the underlying protocol's honest stake threshold assumption τ. This parameter identifies the minimum percentage of stake that needs to be honest; typically, τ is set to $1/2 + \epsilon$ or $2/3 + \epsilon$ for some $\epsilon > 0$.

We assume that stake is delegated to P pools, of which P_h back the correct protocol execution and control ρ_h percentage of the total stake. In this setting we need to consider pools, rather than stake itself. Intuitively, when a player delegates, they relinquish their staking rights for block production (though they retain the right to choose their delegate). If an honest party delegates to the adversary, then its becomes adversarial in the stake-pooled setting. Thus, the adversary compromises the stake-pooled security by corrupting enough pools, such that the percentage of stake that honest pools control is less than τ. We stress that this does not necessarily imply that the adversary needs to corrupt a large number of pools. For instance, if a pool controls $\rho_a \geq 1 - \tau$ of the total stake, the adversary can compromise the stake-pooled variant's security by only corrupting this single large pool. Corollary 1 formalizes this argument; its proof following directly from the definition of the chain selection rules of Section 5.2. Also Appendix 5.5 briefly discusses special attacks against stake-pooled PoS, namely sybil and replay attacks.

Corollary 1. *Assume a PoS protocol π, the execution of which is secure if at least τ stake is honest. The execution of π and the core-wallet protocol $\pi_{\text{CoreWallet}}$ under the chain selection rules of Section 5.2 is secure if $\rho_h \geq \tau$, where ρ_h is the percentage of stake controlled by pools that back the correct protocol execution.*

5.4 The Wallet Modes of Execution

- **Regular Wallet:** A regular wallet is bootstrapped with a *base* address α_0 and its stake is managed by a key $(\mathsf{vks}, \mathsf{sks})$. After α_0 receives its first assets, the wallet performs staking actions using $(\mathsf{vks}, \mathsf{sks})$. In order to stake on its own, the wallet publishes a delegation certificate Σ to its own key $(\mathsf{vks}, \mathsf{sks})$. Subsequent addresses are pointer addresses to Σ, hence all addresses are managed by the same staking key. The user can delegate to a staking pool with key vks_P via a certificate Σ_d that delegates from vks to vks_P.
- **Cold Staking:** This wallet is offline (e.g. on paper) and rarely issues payments, but performs staking actions. It is bootstrapped as above and, with its payment keys stored offline, the staking keys are managed as follows: i) **basic security:** the staking key $(\mathsf{vks}, \mathsf{sks})$ that manages all addresses is online; in case sks is compromised, the user moves the funds to new addresses with a new staking key; ii) **enhanced security:** the wallet creates a certificate Σ' that delegates from vks to a "hot" key vks_h; next the user stores $(\mathsf{vks}, \mathsf{sks})$ offline and uses $(\mathsf{vks}_h, \mathsf{sks}_h)$ such that, if $(\mathsf{vks}, \mathsf{sks})_h$ is compromised, $(\mathsf{vks}, \mathsf{sks})$ re-delegate to a new "hot" key without the need to move the funds.
- **Enhanced Unlinkability of Addresses:** Aiming at better privacy, each address is managed by separate staking keys. To re-delegate the wallet issues a certificate for each staking key, i.e. for each address. Different security levels are also available: i) **online:** the wallet creates pointer addresses directly to the stake pool's registration certificate; in order to re-delegate, it moves the funds to new addresses; ii) **offline:** the payment keys are stored offline, so the wallet creates base addresses, managed by different staking keys which are online; in order to re-delegate, it publishes a new certificate for each key.
- **Stake Pool Wallet:** A stake pool's wallet performs staking with a key $(\mathsf{vks}_P, \mathsf{sks}_P)$ as such: i) **basic security:** $(\mathsf{vks}_P, \mathsf{sks}_P)$ is online; in case of compromise, the wallet creates

a new staking key, while an alert mechanism should notify the users to re-delegate to the new key; ii) **enhanced security:** the wallet creates a lightweight certificate Σ_l, which delegates to a "hot" key vks_{Ph}, and then stores $(\mathsf{vks}_P, \mathsf{sks}_P)$ offline, while using $(\mathsf{vks}, \mathsf{sks})_{Ph}$ and Σ_l for staking; if $(\mathsf{vks}, \mathsf{sks})_{Ph}$ is compromised, the wallet creates a new hot key $(\mathsf{vks}, \mathsf{sks})'_{Ph}$ and re-delegates to it using a higher counter compared to Σ_l.

5.5 Attacks against Stake Pooled PoS

Sybil Attacks. Using stake pools for the PoS protocol's execution, rather than the stakeholders themselves, introduces the possibility of **Sybil Attacks** [21]. Specifically, suppose that the adversary creates a large number of stake pools. Since it is hard for honest players to identify an adversarial pool, they could appear legitimate and users might be convinced to delegate to them, thus increasing the adversarial stake ratio. This is an inherent problem, since no form of external identification exists and an adversary can create a large number of staking keys and registration certificates. A potential countermeasure is to have pool leaders commit (some of) their own stake to their pool. In our setting, this method is facilitated by introducing an extra field in the delegation certificate's metadata, which identifies the leader's addresses and funds committed to the pool. As long as the funds are locked in these addresses, the leader cannot use them for multiple pool commitments. While this does not directly prevent a Sybil attack, it does bound the attacker's identity production capability.

Replay Attacks. An important consideration is **replay protection**. Replay attacks are prominent in account-based ledgers, where an adversary may re-publish an old transaction. For instance, suppose *Alice* sends x assets from her account A to *Bob*. After the payment is published, A controls $y = z - x$ assets, where z are the funds A held before the payment. In a replay scenario, the adversary re-publishes this payment, thus a further amount x of funds is sent from A to Bob. The same vulnerability exists against certificates, e.g. an attacker can re-publish a past certificate in order to forcefully change a user's delegation choice. Our solution is based on an *address whitelist*. Specifically, the certificate defines the addresses which are allowed to publish it; naturally, this scheme assumes that, upon creating the certificate, the wallet knows the possible addresses that can publish it. In order to verify a certificate, a node checks whether it is published in a transaction issued by a whitelisted address. In order to replay the certificate, the adversary would need to infiltrate one of the whitelisted addresses. Notably, no state needs to be maintained by the verifiers. Indeed, the information which counters a replay attack, i.e. the address whitelist, is in the certificate, without the need to parse the entire ledger or maintain extra local state, as in the case of the counter-based mechanisms used in Ethereum, cf. [22]. Alternative, a deadline-based approach could be followed. Similar to a whitelist, a certificate also includes a block limit, i.e. the latest block in which the transaction can be published. The block limit should be carefully chosen, in order to allow the block producers enough time to include it in a block. On the one hand, if the limit is too low, then the block producers might not receive it on time or might not prioritize it, and thus not publish it at all after the limit has expired. On the other hand, if it is too high, then the user needs to wait a large amount of time before being able to re-delegate their stake, which hurts the usability of the system.

6 Conclusion

Our work explores digital asset management on Proof-of-Stake (PoS) ledgers. Motivated by the lack of formal treatment of PoS wallets, as well as the low level of decentralization in existing systems, we provide two core results. First, we identify address malleability, a serious

threat against any system which combines multiple attributes in a single object; in our case two keys for payment and staking combined into a single address. Second, we formally define a wallet's core, i.e. the module responsible for actions like signing transactions and staking actions. Combining the core wallet with a PoS ledger, we implement delegation and stake pool in a PoS system, while investigating how these actions affect security. We also pose a number of questions. For instance, our functionality does not cover the privacy and address unlinkability requirements, as well as forward secure solutions. Finally, we highlight the need for computationally efficient non malleable schemes without leaking the public payment key.

References

[1] Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, L.: On the malleability of bitcoin transactions. In: Brenner, M., Christin, N., Johnson, B., Rohloff, K. (eds.) FC 2015 Workshops. LNCS, vol. 8976, pp. 1–18. Springer, Heidelberg (Jan 2015). https://doi.org/10.1007/978-3-662-48051-9_1

[2] Arapinis, M., Gkaniatsou, A., Karakostas, D., Kiayias, A.: A formal treatment of hardware wallets. In: Goldberg, I., Moore, T. (eds.) FC 2019. LNCS, vol. 11598, pp. 426–445. Springer, Heidelberg (Feb 2019). https://doi.org/10.1007/978-3-030-32101-7_26

[3] Badertscher, C., Gaži, P., Kiayias, A., Russell, A., Zikas, V.: Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 913–930. CCS '18, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3243734.3243848, http://doi.acm.org/10.1145/3243734.3243848

[4] Bellare, M., Boldyreva, A., O'Neill, A.: Deterministic and efficiently searchable encryption. In: Annual International Cryptology Conference. pp. 535–552. Springer (2007)

[5] Bellare, M., Miner, S.K.: A forward-secure digital signature scheme. In: Wiener, M.J. (ed.) CRYPTO'99. LNCS, vol. 1666, pp. 431–448. Springer, Heidelberg (Aug 1999). https://doi.org/10.1007/3-540-48405-1_28

[6] Bentov, I., Pass, R., Shi, E.: Snow white: Provably secure proofs of stake. Cryptology ePrint Archive, Report 2016/919 (2016), http://eprint.iacr.org/2016/919

[7] Bruenjes, L., Kiayias, A., Koutsoupias, E., Stouka, A.P.: Reward sharing schemes for stake pools. Computer Science and Game Theory (cs.GT) arXiv:1807.11218 (2018)

[8] Buterin, V., Griffith, V.: Casper the friendly finality gadget. CoRR **abs/1710.09437** (2017), http://arxiv.org/abs/1710.09437

[9] Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: 42nd FOCS. pp. 136–145. IEEE Computer Society Press (Oct 2001). https://doi.org/10.1109/SFCS.2001.959888

[10] Canetti, R.: Universally composable signatures, certification and authentication. Cryptology ePrint Archive, Report 2003/239 (2003), http://eprint.iacr.org/2003/239

[11] Chakravarty, M.M.T., Coretti, S., Fitzi, M., Gazi, P., Kant, P., Kiayias, A., Russell, A.: Hydra: Fast isomorphic state channels. Cryptology ePrint Archive, Report 2020/299 (2020), https://eprint.iacr.org/2020/299

[12] Chen, J., Gorbunov, S., Micali, S., Vlachos, G.: ALGORAND AGREEMENT: Super fast and partition resilient byzantine agreement. Cryptology ePrint Archive, Report 2018/377 (2018), https://eprint.iacr.org/2018/377

[13] Community, E.: Eos.io technical white paper v2 (2018), https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md

[14] Courtois, N.T., Emirdag, P., Valsorda, F.: Private key recovery combination attacks: On extreme fragility of popular bitcoin key management, wallet and cold storage solutions in presence of poor RNG events. Cryptology ePrint Archive, Report 2014/848 (2014), http://eprint.iacr.org/2014/848

[15] Damgård, I.: Collision free hash functions and public key signature schemes. In: Chaum, D., Price, W.L. (eds.) EUROCRYPT'87. LNCS, vol. 304, pp. 203–216. Springer, Heidelberg (Apr 1988). https://doi.org/10.1007/3-540-39118-5_19

[16] David, B., Gaži, P., Kiayias, A., Russell, A.: Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake protocol. Cryptology ePrint Archive, Report 2017/573 (2017), http://eprint.iacr.org/2017/573

[17] David, B., Gazi, P., Kiayias, A., Russell, A.: Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018, Part II. LNCS, vol. 10821, pp. 66–98. Springer, Heidelberg (Apr / May 2018). https://doi.org/10.1007/978-3-319-78375-8_3

[18] Decker, C., Wattenhofer, R.: Bitcoin transaction malleability and MtGox. In: Kutylowski, M., Vaidya, J. (eds.) ESORICS 2014, Part II. LNCS, vol. 8713, pp. 313–326. Springer, Heidelberg (Sep 2014). https://doi.org/10.1007/978-3-319-11212-1_18

[19] decred.org: Decred—an autonomous digital currency (2019), https://decred.org

[20] Dolev, D., Dwork, C., Naor, M.: Nonmalleable cryptography. SIAM review **45**(4), 727–784 (2003)

[21] Douceur, J.R.: The sybil attack. In: International workshop on peer-to-peer systems. pp. 251–260. Springer (2002)

[22] Ethereum: Glossary: Account nonce (2018), https://github.com/ethereum/wiki/wiki/Glossary

[23] Ethereum: Proof of stake faqs (2018), https://github.com/ethereum/wiki/wiki/Proof-of-Stake-FAQs

[24] Fanti, G.C., Kogan, L., Oh, S., Ruan, K., Viswanath, P., Wang, G.: Compounding of wealth in proof-of-stake cryptocurrencies. In: Goldberg, I., Moore, T. (eds.) FC 2019. LNCS, vol. 11598, pp. 42–61. Springer, Heidelberg (Feb 2019). https://doi.org/10.1007/978-3-030-32101-7_3

[25] Garay, J.A., Kiayias, A., Leonardos, N.: The bitcoin backbone protocol: Analysis and applications. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015, Part II. LNCS, vol. 9057, pp. 281–310. Springer, Heidelberg (Apr 2015). https://doi.org/10.1007/978-3-662-46803-6_10

[26] Garay, J.A., Kiayias, A., Leonardos, N.: The bitcoin backbone protocol with chains of variable difficulty. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017, Part I. LNCS, vol. 10401, pp. 291–323. Springer, Heidelberg (Aug 2017). https://doi.org/10.1007/978-3-319-63688-7_10

[27] Gilad, Y., Hemo, R., Micali, S., Vlachos, G., Zeldovich, N.: Algorand: Scaling byzantine agreements for cryptocurrencies. Cryptology ePrint Archive, Report 2017/454 (2017), http://eprint.iacr.org/2017/454

[28] Goldwasser, S., Micali, S., Rivest, R.L.: A "paradoxical" solution to the signature problem (abstract) (impromptu talk). In: Blakley, G.R., Chaum, D. (eds.) CRYPTO'84. LNCS, vol. 196, p. 467. Springer, Heidelberg (Aug 1984)

[29] Goodman, L.: Tezos—a self-amending crypto-ledger white paper (2014)

[30] Gutoski, G., Stebila, D.: Hierarchical deterministic bitcoin wallets that tolerate key leakage. In: Böhme, R., Okamoto, T. (eds.) FC 2015. LNCS, vol. 8975, pp. 497–504. Springer, Heidelberg (Jan 2015). https://doi.org/10.1007/978-3-662-47854-7_31

[31] Kerber, T., Kiayias, A., Kohlweiss, M., Zikas, V.: Ouroboros crypsinous: Privacy-preserving proof-of-stake. In: 2019 IEEE Symposium on Security and Privacy. pp. 157–174. IEEE Computer Society Press (May 2019). https://doi.org/10.1109/SP.2019.00063

[32] Kiayias, A., Russell, A., David, B., Oliynykov, R.: Ouroboros: A provably secure proof-of-stake blockchain protocol. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017, Part I. LNCS, vol. 10401, pp. 357–388. Springer, Heidelberg (Aug 2017). https://doi.org/10.1007/978-3-319-63688-7_12

[33] Maxwell, G., et al.: Deterministic wallets (2014)

[34] Micali, S., Rabin, M.O., Vadhan, S.P.: Verifiable random functions. In: 40th FOCS. pp. 120–130. IEEE Computer Society Press (Oct 1999). https://doi.org/10.1109/SFFCS.1999.814584

[35] Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008)

[36] Pass, R., Seeman, L., shelat, a.: Analysis of the blockchain protocol in asynchronous networks. In: Coron, J., Nielsen, J.B. (eds.) EUROCRYPT 2017, Part II. LNCS, vol. 10211, pp. 643–673. Springer, Heidelberg (Apr / May 2017). https://doi.org/10.1007/978-3-319-56614-6_22

[37] Pass, R., Shi, E.: The sleepy model of consensus. In: Takagi, T., Peyrin, T. (eds.) ASIACRYPT 2017, Part II. LNCS, vol. 10625, pp. 380–409. Springer, Heidelberg (Dec 2017). https://doi.org/10.1007/978-3-319-70697-9_14

[38] Reed, D., Sporny, M., Longley, D., Allen, C., Grant, R., Sabadello, M.: Decentralized identifiers (dids) v0. 11. W3C, Draft Community Group Report **9** (2018)

[39] Steem: Steem whitepaper (2018), https://steem.com/steem-whitepaper.pdf

[40] Van Saberhagen, N.: Cryptonote v 2.0 (2013)

[41] Wood, G.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper **151**, 1–32 (2014)

[42] Wuille, P.: Hierarchical Deterministic Wallets. https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki (2017), [Online; January 2020]

A The Malleability Predicates

In this section we describe specific predicates for various cases of malleability. The first parameter of the predicate is the set L_P of all created addresses and their attributes for a party P in the system. This parameter is necessary, so that the predicate compares the given address with the honestly generated ones. The second parameter is the auxiliary information aux_M, which takes the values "recover", "issue", or "verify". The auxiliary information allows the predicate to behave differently, depending on the action, thus making it more adjustable for various use cases. The third parameter is the address, for which the predicate may identify a malleability attack.

A.1 The Full Malleability Predicate

The lowest level of security against malleability attacks is provided by the full malleability predicate. A *fully malleable* construction, instantiated $M^{\mathbb{L},\mathbb{T},P}$ with $M^{\mathbb{L},\mathbb{T},P}_{\mathsf{FM}}$ described in Algorithm 2, enables an adversary to produce forged addresses with access only to a single honestly-produced address.

Algorithm 2 The *fully malleable* predicate.

```
function M^{L,T,P}_FM(aux, α)
    switch "aux" do
        case "issue"
            return 1
        case "verify" OR "recover"
            d = parsePubAttrs(α)
            for δ ∈ d do
                if ∀α' ∈ L_P, d' = parsePubAttrs(α'): δ ∉ d' then
                    return 0                          ▷ δ not registered detected
                end if
            end for
    return 1
end function
```

A.2 The Ex Post Malleability Predicate

In an *ex post malleable* construction, the malleability predicate $M^{\mathbb{L},\mathbb{T},P}$ is instantiated with $M^{\mathbb{L},\mathbb{T},P}_{\mathsf{PM}}(aux, \alpha)$. In this case, the predicate first identifies the list d of public attributes for the address α. Then, for each attribute $\delta \in d$, it checks: i) if there exists an issued transaction $tx = (\Theta, \alpha_s, \alpha_r, m)$, such that the list of public attributes that pertain to the sender's address includes δ, and ii) if there exists an address that the wallet has created which used δ in its public attributes. If both checks fail, then the predicate returns 0, otherwise 1.

Intuitively, this construction allows malleability to occur only for addresses whose payment key has been previously used in a transaction and for which all public attributes have been used by the wallet in other addresses. Therefore, as long as the payment key of the address has not been used, the scheme provides non-malleability.

The *ex post malleable* construction, instantiated with the predicate $M_{\mathsf{PM}}^{\mathbb{L},\mathbb{T},P}$, is described in Algorithm 3. By parsePubAttrs we denote the parsing of the list of public attributes given an address.

Algorithm 3 The *ex post* malleability predicate.

```
function M_PM^{L,T,P}(aux, α)
    switch "aux" do
        case "issue"
            return 1
        case "verify" OR "recover"
            d = parsePubAttrs(α)
            for δ ∈ d do
                if ∀α' ∈ L_P, d' = parsePubAttrs(α'): δ ∉ d' then
                    return 0
                end if
                if ∀(Θ, α_s, α_r, m) ∈ T, d_s = parsePubAttrs(α_s): δ ∉ d_s then
                    return 0
                end if
            end for
            return 1
    return 0
end function
```

A.3 The "Sink" Malleability Predicate

"Sink" malleability schemes are instantiated with simple cryptographic primitives and protect against attacks from both network and targeting adversaries, such as the stake pool leader described above. Nevertheless, our design is (on purpose) generic enough, in order to allow the description and usage of predicates for all malleability levels. Intuitively, a "sink" malleable algorithm requires that only the owner of the honest wallet can create addresses for payment keys of the wallet, even though it is possible to send funds to a forgery. This is expressed by differentiating the behavior of the predicate depending on the auxiliary information. If aux pertains to the issuing of transactions then the predicate returns 1, i.e. accepts all addresses to which the wallet tries to send funds, whereas for all other cases it requires that the address is honestly generated. For completeness, the *"sink" malleable* predicate M_{SM} is defined in Algorithm 4.

B Security of the Generic Core-Wallet Protocol

The rationale on our security analysis is based on the Universal Composable Framework by Canetti [9], which we review in this section. Also we briefly revisit the definition of secure digital signatures, a main component of our definitions. For a more complete discussion, we refer the reader to [10, 28]. Finally, we describe the properties of a cryptographic hash function following the definitions of Damgård [15].

Algorithm 4 The *"sink" malleable* predicate.

```
function M_SM(L_P, aux_M, α)
    switch aux_M do
        case "issue"
            return 1
        case "verify" OR "recover"
            if ∃ l_α : (α, l_α) ∈ L_P then
                return 1                                  ▷ α is registered
            end if
    return 0                                              ▷ No α is registered
end function
```

B.1 The Universally Composable Framework

In our analysis, we rely on a simulation-based definition for capturing the security properties of our system and, more specifically, the Universal Composability (UC) Framework by Canetti [9].

As a preparation for presenting the framework, consider two ensembles $X = \{X_{\lambda,z}\}_{\lambda\in\mathbb{N},z\in\{0,1\}^*}$ and $Y = \{Y_{\lambda,z}\}_{\lambda\in\mathbb{N},z\in\{0,1\}^*}$ of binary random variables. X and Y are said to be *computationally indistinguishable*, denoted by $X \approx_c Y$, if for all z it holds that $\mid \Pr[\mathcal{D}(X_{\lambda,z}) = 1] - \Pr[\mathcal{D}(Y_{\lambda,z}) = 1] \mid$ is negligible[7] in λ, i.e. $negl(\lambda)$, for every probabilistically polynomial-time (PPT) distinguishing algorithm $\mathcal{D}$.

What follows is a brief description of the framework. We refer the interested reader to [9] for a formal presentation.

The main idea of security proofs under the UC framework relies on the comparison between the execution of a concrete protocol, say π, and a security definition, named the *ideal functionality*. These two executions are, respectively, the *real world* and the *ideal world*. Both are controlled by an entity called the *environment*, denoted by $\mathcal{Z}$, which can submit actions and observe outputs from the executions. The environment controls the execution of π, through choosing the inputs of its participants, and also the actions of the adversary $\mathcal{A}$ in the real world. It also controls the inputs of the ideal functionality, $\mathcal{F}$, and the actions of the ideal adversary $\mathcal{S}$ (or *simulator*). The adversary $\mathcal{A}$ is expected to read the messages exchanged between the protocol players and even delay them. Moreover, it is allowed to corrupt players, in which case the player's secret state is compromised and is available to the adversary.

More formally, every entity is modeled as a PPT Interactive Turing Machine (ITM), and the real world and ideal executions are respectively represented by the ensembles

$$\mathsf{REAL}_{\pi,\mathcal{A},\mathcal{Z}} = \{\mathsf{REAL}_{\pi,\mathcal{A},\mathcal{Z}}(\lambda, z, r)\}_{\lambda\in\mathbb{N},z\in\{0,1\}^*}$$

and

$$\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}} = \{\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(\lambda, z, r)\}_{\lambda\in\mathbb{N},z\in\{0,1\}^*}$$

and uniform randomly chosen value r. We use $\mathsf{REAL}_{\pi,\mathcal{A},\mathcal{Z}}(\lambda, z, r)$ to denote the output of the environment $\mathcal{Z}$ in the real-world execution of a protocol π and the adversary $\mathcal{A}$ under security parameter λ, input z and randomness r. Analogously, we denote by $\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(\lambda, z, r)$ the output of the environment in the ideal interaction between the simulator $\mathcal{S}$ and the ideal

[7] We say that a function f is negligible in k if for every c and d in $\mathbb{N}$, there is a $k_0 \in \mathbb{N}$ such that for all $k > k_0$ and $x \in \{0,1\}^{k^d}$ it holds that $f(x,k) < k^{-c}$.

functionality $\mathcal{F}$ under security parameter λ, input z and randomness r. It is said that *the protocol π securely realizes the functionality $\mathcal{F}$* when the environment cannot distinguish between the two worlds, i.e. for every $\mathcal{A}$ exists a simulator $\mathcal{S}$ such that for every PPT $\mathcal{Z}$ we have that $\mathsf{REAL}_{\pi,\mathcal{A},\mathcal{Z}} \approx_c \mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$.

B.2 The Secure Digital Signature Scheme

A digital signature scheme Σ, in the sense of Canetti [10] and Goldwasser *et al.* [28], is a triple of algorithms $\langle \mathsf{KeyGen}, \mathsf{Verify}, \mathsf{Sign} \rangle$. Σ is said to be resistant to Existential Unforgeable under Adaptive Chosen Message Attacks ($\mathsf{EUF\text{-}CMA}$) if it presents the following properties w.r.t. the security parameter λ:

Completeness: For any message m, it holds:

$$\Pr[(\mathsf{vk},\mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda), \sigma \leftarrow \mathsf{Sign}(\mathsf{sk}, m) : 0 \leftarrow \mathsf{Verify}(m, \sigma, \mathsf{vk})] \leq negl(\lambda)$$

where all the probabilities are computed over the random coins of the generation and sign algorithms.

Non-repudiation: For any message m, the probability that two independent executions of $\mathsf{Verify}(m, \sigma, \mathsf{sk})$ for a key pair $(\mathsf{vk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$, output two different outcomes is smaller than $negl(\lambda)$;

Unforgeability: For any PPT algorithm $\mathcal{A}_{forger}$, which can query the signature oracle $\mathsf{Sign}(\mathsf{sk}, \cdot)$ for signatures on a polynomial number of messages m_i, it holds:

$$\Pr[(\mathsf{vk},\mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda) : (m, \sigma) \leftarrow \mathcal{A}_{forger}^{\mathsf{Sign}(\mathsf{sk},\cdot)} \wedge m \neq m_i] < negl(\lambda)$$

where all the probabilities are computed over the random coins of the adversary, generation algorithm and the sign oracle.

B.3 Cryptographic Hash Functions

A cryptographic hash function $\mathcal{H} : \{0,1\}^* \rightarrow \{0,1\}^l$, for some $l \in \mathbb{N}$ which is the length of the hash values, is a function that presents the following properties:

Definition 1 (Collision resistance). *Given $h \leftarrow \{0,1\}^l$ it should be computationally infeasible for a probabilistic polynomial algorithm to find a value x such that $h = \mathcal{H}(x)$.*

Definition 5 (Pre-image resistance)**.** *It should be computationally infeasible for a probabilistic polynomial algorithm to find two values x, y where $x \neq y$ such that $\mathcal{H}(x) = \mathcal{H}(y)$.*

Definition 6 (Second pre-image resistance)**.** *Given a value x, it should be computationally infeasible for a probabilistic polynomial algorithm to find a value $y \neq x$ such that $\mathcal{H}(x) = \mathcal{H}(y)$.*

B.4 Security of $\pi_{\text{CoreWallet}}$ in the "sink" malleable setting

The security of $\pi_{\text{CoreWallet}}$ is given w.r.t. $\mathcal{F}^M_{\text{CoreWallet}}$, the signature scheme's Existential Unforgeability under Adaptive Chosen Message Attacks ($\mathsf{EUF\text{-}CMA}$) property, and a number of properties of our custom algorithms.

The complete proof for the Theorem 1 presented in Section 4.2 can be found below. Here, the ideal functionality, parameterized with the "sink" malleability predicate M_{SM}, is realized by the protocol that employs the "sink" malleable address generation function. It also uses tag and hierarchical key construction functions which present the above necessary properties. For completeness, we define the "sink" malleable address generation function in Algorithm 5.

Algorithm 5 The "sink" malleable address generation function, parameterized by $\mathcal{H}(\cdot)$ and $\Sigma = \langle \mathsf{KeyGen}, \mathsf{Verify}, \mathsf{Sign} \rangle$. The input is a tuple $l_{\alpha,Gen}$, consisting of the auxiliary information aux and the attributes.

```
function SinkGenAddr(l_{α,Gen})
    (aux, st, vkp, skp) = parse(l_{α,Gen})
    switch aux do
        case "base"
            β = H(st)
        case "pointer"
            β = getPointer(st)
        case "exile"
            β = st
    α = H(vkp)||β||Sign(skp, β)
    return α
end function
```

Theorem 1. *Let the generic protocol* $\pi_{\text{CoreWallet}}$ *be parameterized by a signature scheme* $\Sigma = \langle \mathsf{KeyGen}, \mathsf{Verify}, \mathsf{Sign} \rangle$ *and the* RTagGen, HKeyGen, *and* GenAddr *functions. Then* $\pi_{\text{CoreWallet}}$ *securely realizes the ideal functionality* $\mathcal{F}_{\text{CoreWallet}}^{M_{\mathsf{SM}}}$ *if and only if* Σ *is EUF-CMA,* GenAddr *is collision resistant and attribute non-malleable (cf. Definitions 2 and 4),* RTagGen *is collision resistant (cf. Definition 1), and* HKeyGen *is hierarchical for* Σ *(cf. Definition 3).*

Proof. The proof is constructed in the UC Framework, therefore it is a simulation based proof. As such, we will show that the environment $\mathcal{Z}$ cannot efficiently distinguish between two executions, the ideal and the real. Here, the simulator $\mathcal{S}$ interacts with the ideal functionality $\mathcal{F}_{\text{CoreWallet}}^{M_{\mathsf{SM}}}$ in the ideal execution, whereas $\mathcal{A}$ interacts with $\pi_{\text{CoreWallet}}$ in the real execution.

Proof overview. We divide the proof in the "if" and "only if" parts. First, the "if" part shows that if $\pi_{\text{CoreWallet}}$ does securely realize the ideal functionality $\mathcal{F}_{\text{CoreWallet}}^{M_{\mathsf{SM}}}$, when instantiated with a EUF-CMA signature scheme Σ, a collision resistant and non-malleable address generation scheme GenAddr, and suitable RTagGen, HKeyGen functions at least one of the conditions is violated. The "only if" part shows that, if either of the functions' properties does not hold, e.g. if Σ is not EUF-CMA or GenAddr is either not collision resistant or non-malleable, then $\pi_{\text{CoreWallet}}$ does not securely realize the functionality $\mathcal{F}_{\text{CoreWallet}}^{M_{\mathsf{SM}}}$, i.e. the environment is able to distinguish between the two executions.

Let us now provide the construction for the simulator $\mathcal{S}$, which will be useful in the "if" part of the proof.

The simulator $\mathcal{S}$. The simulator $\mathcal{S}$ runs internally a copy of the adversary $\mathcal{A}$, and keeps a table TABLE of tuples $(\cdot, \cdot, \cdot, \cdot)$ of respectively addresses, attributes, and staking key pairs. Also it performs as follows:

- Any inputs received from the environment $\mathcal{Z}$, forward them to the internal copy of $\mathcal{A}$. Moreover, forward any output from $\mathcal{A}$ to $\mathcal{Z}$;
- **Initialization:** Upon receiving the message $(\textsc{Initialise}, sid)$ from the functionality $\mathcal{F}_{\text{CoreWallet}}$, compute a dummy master key $msk \xleftarrow{\$} \{0,1\}^\lambda$ and return $(\textsc{InitialiseOk}, sid)$;
- **Address Generation:** Upon receiving $(\textsc{GenerateAddress}, sid, aux)$ from $\mathcal{F}_{\text{CoreWallet}}$, do similarly to protocol $\pi_{\text{CoreWallet}}$, that is:
  - set $i \leftarrow \mathcal{I}$,
  - set the key pair $(\mathsf{vkp}_c, \mathsf{skp}_c) = \mathsf{HKeyGen}(\langle msk, \text{“payment”}, i\rangle)$,
  - set the tag $wrt = \mathsf{RTagGen}(\langle msk, i\rangle)$,

  and do the following:
  - if $aux = (\text{“base”})$ compute the key pair $(\mathsf{vks}_c, \mathsf{sks}_c) = \mathsf{HKeyGen}(\langle msk, \text{“staking”}, i\rangle)$ and set $\beta = \mathsf{vks}_c$;
  - if $aux = (\text{“pointer”}, \mathsf{vks})$, set $\beta = \mathsf{vks}$;
  - if $aux = (\text{“exile”})$, set $\beta = \perp$.

  Then compute the address $\alpha = \mathsf{GenAddr}(\langle aux, \beta, \mathsf{vkp}_c, wrt\rangle)$ and set the address $l_\alpha = \langle \mathsf{vks}_c, wrt, aux, \mathsf{vkp}_c, \mathsf{skp}_c, \mathsf{sks}_c\rangle$. Then record $(\alpha, l_\alpha, \beta, \mathsf{skp}_c)$ to TABLE. Finally, hand to $\mathcal{F}_{\text{CoreWallet}}$ the message $(\textsc{Address}, sid, \alpha, l_\alpha)$;
- **Issue Transaction:** Upon receiving $(\textsc{Pay}, sid, \Theta, \alpha_s, \alpha_r, m)$ find a record l_α on TABLE that contains the sender's address α_s as the first item. Then generate the signature σ for the transaction tx, such that $tx = (\Theta, \alpha_s, \alpha_r, m)$, using Sign and the payment key of α_s, and hand $(\textsc{Transaction}, sid, tx, \sigma)$ to the functionality $\mathcal{F}_{\text{CoreWallet}}$. Note that with the attribute list l_α, $\mathcal{S}$ can properly generate σ. Moreover such record is expected to be in TABLE, since the functionality allows the issuing of transactions by properly generated addresses by checking on the list L_P before sending to $\mathcal{S}$;
- **Verify Transaction:** Upon receiving $(\textsc{VerifyPayment}, sid, tx, \sigma)$ from $\mathcal{F}_{\text{CoreWallet}}$, find the recorded verification key vkp_c for the sender's address α_s in $tx = (\Theta, \alpha_s, \alpha_r, m)$ by looking up l_α for α_s in TABLE, and use Verify to retrieve the verification bit ϕ. Then return to $\mathcal{F}_{\text{CoreWallet}}$ the message $(\textsc{VerifiedPayment}, sid, tx, \sigma, \phi)$;
- **Issue Staking:** Similarly to issuing a payment transaction, upon receiving the message $(\textsc{Stake}, sid, stx)$, such that $stx = (\mathsf{vkp}, m)$, find the correspondent staking key skp and use Sign to generate σ, then hand $(\textsc{Staked}, sid, stx, \sigma)$ to $\mathcal{F}_{\text{CoreWallet}}$;
- **Verify Staking:** As before, upon receiving $(\textsc{VerifyStaking}, sid, stx, \sigma)$, find the staking key that pertains to stx, use Verify to retrieve the verification bit ϕ, and send the message $(\textsc{VerifiedPayment}, sid, stx, \sigma, \phi)$ to $\mathcal{F}_{\text{CoreWallet}}$. Similarly to *Issue Transaction* interface, note that $\mathcal{S}$ knows skp_c via TABLE;
- **Party Corruption:** Whenever the adversary $\mathcal{A}$ corrupts a party P, $\mathcal{S}$ corrupts it in the ideal process and hands to $\mathcal{A}$ the corresponding entries in TABLE.

The “if” part. Assume, for the sake of the argument, that the environment $\mathcal{Z}$ can distinguish between the ideal and the real execution with non-negligible probability for any simulator construction, including the earlier $\mathcal{S}$ construction. In that case, it suffices to show that, if $\pi_{\text{CoreWallet}}$ does not securely realize $\mathcal{F}_{\text{CoreWallet}}$, and given the $\mathcal{S}$ construction, then either of the following holds when “bad” events (say E) occur: the signature scheme is not $\mathsf{EUF\text{-}CMA}$, $\mathsf{GenAddr}$ is not collision resistant or attribute non-malleable, the function $\mathsf{RTagGen}$ is not collision resistant, or the hierarchical property of $\mathsf{HKeyGen}$ does not hold.

Unforgeability: We assume that *collision* resistance and non-malleability properties of the algorithm $\mathsf{GenAddr}$ hold, likewise the collision resistance and hierarchical properties for $\mathsf{RTagGen}$ and $\mathsf{HKeyGen}$ respectively, along with the signature scheme properties *completeness*

and *non-repudiation*, but unforgeability does not hold (otherwise the theorem completes). Note that, by hypothesis, $\mathcal{Z}$ distinguishes between the two worlds for any construction of $\mathcal{S}$, including the earlier $\mathcal{S}$ construction.

Now we construct a forger G for the unforgeability game per the $\mathsf{EUF\text{-}CMA}$ definition, which initially receives $(\mathsf{vkp}, \mathsf{skp})$ as a challenge (we focus on the payment keys and interfaces, but we note that the case for staking is analogous). G simulates the earlier described $\mathcal{S}$ in the interaction with $\mathcal{Z}$. It then issues signature queries to its game, when requested by its simulation of $\mathcal{S}$ and $\mathcal{F}_{\text{CoreWallet}}$ for signatures on vkp.

In addition, when receiving messages like $(\text{VERIFYPAYMENT}, sid, tx, \sigma)$ for other addresses (possibly from other verification keys), G uses its internal simulation, specifically the *Transaction Verification* interface, i.e. its internally generated keys, to properly simulate the execution to $\mathcal{Z}$. In particular it checks if (tx, σ) has been queried in the security game. If it is not listed as queried and $\mathsf{Verify}(tx, \sigma, \mathsf{vkp}) = 1$, then it outputs (tx, σ) and wins the game. Otherwise, it continues with the simulation.

Given that the environment $\mathcal{Z}$ distinguishes between the two executions by hypothesis and all other properties of the functions hold, we are guaranteed that if $\pi_{\text{CoreWallet}}$ does not securely realize $\mathcal{F}_{\text{CoreWallet}}$, then the unforgeability property does not hold.

Note that the earlier unforgeability reasoning is valid for Σ with key generation relying on KeyGen, however $\pi_{\text{CoreWallet}}$ relies on $\mathsf{HKeyGen}$. Therefore, consider the following argument.

$\mathsf{HKeyGen}$ **is hierarchical for** Σ **and every index** i **is used only once:** Assume the two following protocols: i) a protocol which is similar to $\pi_{\text{CoreWallet}}$, except the key generation function KeyGen of Σ is used instead of $\mathsf{HKeyGen}$, and ii) the protocol $\pi_{\text{CoreWallet}}$. Now, it is evident that the first protocol securely realizes the ideal functionality (since all other properties needed as per the Theorem hold). Therefore, the execution of the first protocol is indistinguishable from the execution of the ideal functionality, as proved in the earlier reasoning.

Next, consider a PPT algorithm D who tries to distinguish between the executions of the two protocols above. Specifically, assume that there exists a "special" index i, for which the usage of $\mathsf{HKeyGen}$ in the signature scheme is insecure, i.e. a forgery can be computed by the adversary. For the sake of argument, consider the probability that the scheme breaks for this index i by p. It is clear that, for this index i, D is successful, by observing the violation of the properties of the signature scheme with $\mathsf{HKeyGen}$. Note also that the number of produced keys, i.e. the number of used indexes in both executions, is bounded by a polynomial $P(\lambda)$. Therefore, the overall probability that D is successful is equal to $\frac{p}{P(\lambda)}$.

However, by definition, $\mathsf{HKeyGen}$ is hierarchical for Σ. Thus, the executions of the two protocols are indistinguishable and, as a result, the probability that D is successful, i.e. $\frac{p}{P(\lambda)}$, is negligible. Consequently, the probability p, i.e. that the signature scheme which uses $\mathsf{HKeyGen}$ breaks, is also negligible. Therefore, the execution of $\pi_{\text{CoreWallet}}$ is indistinguishable from the execution of the ideal functionality as well, thereby $\pi_{\text{CoreWallet}}$ securely realizes the ideal functionality.

The "only if" part. Here we show that, if a single property does not hold, then the environment $\mathcal{Z}$ can distinguish between the real and ideal executions with non-negligible probability. In other words, there is no simulator construction that prevents $\mathcal{Z}$ from distinguishing both executions.

Success probability of $\mathcal{Z}$ **under weakened assumptions.** We now assume that some property of the functions used by the protocol is broken. We will then show that $\pi_{\text{CoreWallet}}$ does not securely realize $\mathcal{F}_{\text{CoreWallet}}$. Specifically, we can create an environment $\mathcal{Z}$ and an

adversary $\mathcal{A}$ such that, for *any* simulator $\mathcal{S}$, $\mathcal{Z}$ distinguishes between the real execution of $\mathcal{A}$ with $\pi_{\text{CoreWallet}}$ and the ideal execution of $\mathcal{S}$ with $\mathcal{F}_{\text{CoreWallet}}$.

Initially, the environment sends (INITIALISE, sid) for some party P. For each property required by the theorem, we show that the environment can distinguish between the two executions as follows:

- **Completeness:** We assume that Σ is not complete. The environment now initializes the wallet for a second party P'. Then it sends two types of message to generate address (GENERATEADDRESS, sid, aux) for some arbitrary value of auxiliary information aux and obtains two addresses α_s and α_r for the parties P and P' respectively. Next, it creates a transaction object $tx = (\Theta, \alpha_s, \alpha_r, m)$, for arbitrary values of assets Θ and metadata m, and obtains a signature σ for the transaction tx by sending the message (PAY, sid, tx). Finally, it calls the verification interface of the wallet by sending the message (VERIFYPAYMENT, sid, tx, σ). In the ideal execution the output is always (VERIFIEDPAYMENT, $sid, tx, \sigma, 1$), whereas in the real execution the probability that the output of the interface is (VERIFIEDPAYMENT, $sid, tx, \sigma, 0$) is non-negligible. The environment could also succeed in distinguishing the executions by accessing the **Staking** and **Staking Verification** interfaces, issuing staking acts and checking the verification bit similarly as with payment transactions.
- **Non-repudiation:** We assume that Σ does not offer non-repudiation. The environment now acts like in the case of **completeness**, obtaining a signed transaction (tx, σ). However, it now calls the verification interface twice. In the ideal execution, the verification bit of the response will both times be equal to 1, whereas in the real execution the probability that the verification bit is 0 is non-negligible. Again the environment could access the staking issuing and verification interfaces similarly.
- **Unforgeability:** We assume that Σ is forgeable, so there exists a forger G for Σ. The environment now runs an internal copy of G. When G wishes to obtain a signature from its oracle for some transaction tx, $\mathcal{Z}$ accesses the **Issue Transaction** interface by sending the message (PAY, sid, tx) and obtains a signature, which it forwards to G. When G outputs a signed transaction (tx, σ), $\mathcal{Z}$ proceeds as follows. If tx has been previously signed, i.e. if σ has been creating by accessing the **Issue Transaction** before, then it halts. Otherwise, it accesses the verification interface by sending the message (VERIFYPAYMENT, sid, tx, σ). Now, in the ideal execution the verification bit in the response from the verification interface is always 0, whereas in the real world it is 1 with non-negligible probability.
- **Collision resistance:** We assume that $\mathsf{GenAddr}$ is not collision resistant. The environment obtains two addresses by calling the address generation interface twice, i.e. sending two messages (GENERATEADDRESS, sid, aux) for the same auxiliary information aux. In the ideal execution the attribute lists in the address responses will always be different, whereas in the real execution the probability that two equal addresses for different attribute lists are returned is non-negligible.
- **Attribute non-malleable:** We assume that $\mathsf{GenAddr}$ is not attribute non-malleable. Then the environment $\mathcal{Z}$, which may retrieve correctly generated addresses by accessing the *Address Generation* interface, can generate a forged address α^*. Assume, without loss of generality, that α^* has been the receiving address for some past transaction, therefore α^* owns some assets. Assume also that $\mathcal{Z}$ issues a transaction from α^* to a legitimate address α_r, i.e. created via the address generation interface of $\mathcal{F}_{\text{CoreWallet}}$. Then, upon submitting (VERIFYPAYMENT, $sid, \Theta, \alpha^*, \alpha_r, m, \sigma$), for some assets Θ and metadata m, $\mathcal{Z}$ will receive (VERIFIEDPAYMENT, $sid, \Theta, \alpha^*, \alpha_r, m, \sigma, 0$), since the check of the predicate M_{SM} within $\mathcal{F}^{M_{\mathsf{SM}}}_{\text{CoreWallet}}$ outputs 0. On the other hand,

in the real world interaction with $\pi_{\text{CoreWallet}}$, the environment $\mathcal{Z}$ will receive the message $(\text{VERIFIEDPAYMENT}, sid, \Theta, \alpha^*, \alpha_r, m, \sigma, 1)$. Therefore, $\mathcal{Z}$ is able to distinguish between the executions.

- $\mathsf{RTagGen}$ **collision resistance and every index i is used only once:** Let us now assume that either $\mathsf{RTagGen}$ is not collision resistant or that an index i is used more than once. Then $\mathcal{Z}$ can generate several address requests $(\text{GENERATEADDDRESS}, sid, aux)$ and observe the generated tags wrt within each address α. If $\mathsf{RTagGen}$ is not collision resistant, then $\mathcal{Z}$ will observe a bias in the distribution of the output of $\mathsf{RTagGen}$ in the real world, i.e. it will observe the same recovery tag for two different addresses.
- $\mathsf{HKeyGen}$ **hierarchical property and every index i is used only once:** Assume now that $\mathsf{HKeyGen}$ is not hierarchical for Σ. Then, following the reasoning above, the execution of the protocol which uses KeyGen is indistinguishable from the ideal execution. However, by assumption the execution of $\pi_{\text{CoreWallet}}$ is not indistinguishable from the execution of that protocol anymore. Therefore, it is not indistinguishable from the execution of the ideal functionality as well.

Note that, in all cases, there is no mention of the simulator $\mathcal{S}$, therefore the reasoning applies for any construction of $\mathcal{S}$.

In conclusion, we have shown that if either of the properties is broken, then the environment can distinguish between the two executions, thus a protocol that uses a scheme that does not provide one of the properties does not securely realize the ideal functionality $\mathcal{F}_{\text{CoreWallet}}$. □

C Construction and Recovery of Addresses

The final step in fully realizing the core wallet is to implement the functions used by the protocol $\pi_{\text{CoreWallet}}$, more importantly the address generation function. In the following paragraphs, we outline the three types of addresses in our framework, i.e. the *base*, *pointer*, and *exile* addresses. We concretely describe an address's attributes and how an index are chosen to generate a "child" address.

C.1 Address Types and their Attributes

An address α is typically associated with an attribute list $l_{\alpha,Gen}$. As shown in the previous sections, at least two attributes are required, the staking $(\mathsf{vks}, \mathsf{sks})$ and the payment $(\mathsf{vkp}, \mathsf{skp})$ key pairs. The signing keys, sks and skp, of these pairs are *private* attributes, whereas the verification keys of each pair, vkp of the payment pair and vks of the staking key pair, are *semi-public* and *public* attributes, respectively. We remind that $(\mathsf{vkp}, \mathsf{skp})$ is used in proving ownership of the assets and issuing payments, whereas $(\mathsf{vks}, \mathsf{sks})$ is used to perform staking actions on behalf of the assets.

The first step in computing a "child" address and its attributes is the choice of an index i from the set $\mathcal{I}$. An index is an identifier that is used to generate a "child" key. Our design defines a list of domains $[\mathcal{I}_1, \mathcal{I}_2, \ldots,]$, where each $\mathcal{I}_i$ has a finite, relatively small, cardinality. During address generation, the wallet initially picks indexes from $\mathcal{I}_1$. After all indexes in $\mathcal{I}_1$ have been used, it uses $\mathcal{I}_2$ and so on. It is required that at least one address for an index in $\mathcal{I}_j$ is published on the ledger, i.e. is on the sending or receiving end of a transaction, before indexes from $\mathcal{I}_{j+1}$ are used. During recovery, the wallet sets $j = 1$ and generates all indexes in $\mathcal{I}_j$. It then constructs the recovery tag for each index and compares it with each address in the blockchain. If at least one index has been used in a published address, then the wallet sets $j = j + 1$ and repeats for $\mathcal{I}_{j+1}$. When, for some j no index corresponds to any published

address, recovery is complete.

We discuss now the complexity of the recovery procedure. We note that, given that the cardinality of $\mathcal{I}$ is small, the probability that the same index is chosen twice, even for a random choice, is not negligible. Therefore, two devices that maintain the same wallet core would need to share the state of the indexes that have been used by each. The number of indexes and addresses that the wallet can generate is not restricted, since the set of domains is infinite. In terms of complexity, this naive scheme is linear to the number of addresses in the ledger. We can improve on it by using an index of published addresses indexed by their recovery tag. Using such index, the recovery complexity is now linear to the cardinality of $\bigcup_j \mathcal{I}_j$, i.e. the number of addresses that the wallet owns and has published. We note that, since the recovery tags are public, such index can be constructed and circulated on the network by everybody.

A hierarchical key, which is derived from the wallet's master key msk and is linked to a "child" address, is created by $\mathsf{HKeyGen}$. This function takes the master key, a label $\mathtt{lbl} \in \{$"payment", "staking"$\}$ and an index i and passes them to a Pseudo Random Function, which outputs a pseudo-random value passes it to a Pseudo-Random Number Generator that outputs $P(\lambda)$ bits ρ for some suitable polynomial P. These bits are passed as random coins to the key generation function $\mathsf{KeyGen}(1^\lambda; \rho)$. Therefore, in order to generate a "child" payment key, the protocol runs $\mathsf{HKeyGen}(\langle msk,$ "payment", $i\rangle)$, while similarly $\mathtt{lbl} =$ "staking" is used to issue a new staking key.

As mentioned above, the wallet produces three types of addresses, which are differentiated by the staking object denoted by β. In order to output a **base** address, the wallet computes a staking key vks and sets β as the hash of it. In the case of a **pointer** address, the address's staking key is set indirectly. Specifically, the staking object β is a delegation pointer ptr. The pointer is a string that identifies a published certificate, i.e. the representation of a staking action on the ledger. We briefly mention here that, if ptr points to a valid delegation certificate τ_{del}, then the address's staking key is the delegate's key in τ_{del}, whereas, if ptr points to a registration certificate, then the delegate's key is the key of the stake pool defined in that certificate. In the case of an **exile** address, the staking object is a fixed value ϵ, which is equivalent to $\perp$ in the ideal functionality. Since ϵ does not identify a staking key, the owner of an exile address cannot perform staking actions or delegate the staking rights. Therefore, all assets owned by such addresses are effectively removed from the PoS protocol.

Each address also contains the recovery tag wrt, a public parameter which allows the identification of addresses. The tag is created by the function $\mathsf{RTagGen}$ and links the address to the attribute list without revealing the semi-public attributes, e.g. the payment key. We remind that recovery is a process that relies only on the master key of the wallet, so the wallet should be able to recover an address by *only knowing its payment key*. During recovery in the simplest setting, the wallet computes the keys for its indexes and hashes them to compute the recovery tags. Therefore, $\mathsf{RTagGen}$ is the hash function $\mathcal{H}$ and, by definition, is also collision resistant as needed.

Searchable recovery. This design builds on the premise of searchable encryption [4]. Specifically, a user who possesses the wallet's master key is able to use it in order to search all addresses that appear in the ledger and recognize the ones that belong to the wallet. We assume an instance of a semantically secure symmetric encryption scheme that is comprised of the algorithms $\langle \mathsf{Enc}, \mathsf{Dec}\rangle$. The hierarchical tag generation function $\mathsf{SearchableTagGen}$ in this scheme computes the output of Enc using msk as the encryption key and the index i as the plaintext: $sht = \mathsf{Enc}(msk, i)$

During the recovery phase, the wallet parses all addresses in the blockchain and decrypts

the tag sht' in each as $i' = \mathsf{Dec}(msk, sht')$. If the output is a well formed index $i' \in \mathcal{I}$, then the address belongs in the wallet and its hierarchical index is i'. Also, since the output of the encryption function Enc is by definition random in the domain of Enc, then the recovery tag generation is a PRF.

C.2 Non Malleable Address Generation

Here we analyze the security of the non malleable address scheme, i.e. the most secure scheme we can hope to achieve in this setting. The core idea is to certify the staking object with the payment key, while also revealing the public payment key, such that anybody can verify this certification. A "sink" malleable address is constructed as follows: $\alpha = \mathsf{vkp}||\beta||\mathsf{Sign}(\mathsf{skp}, \beta)$ We note that the payment key vkp also serves the recovery tag wrt. The staking object β for the three address types is: i) **Base Address:** $\beta = \mathcal{H}(\mathsf{vks})$; ii) **Pointer Address:** $\beta = \mathsf{getPointer}(\mathsf{vks})$; iii) **Exile Address:** $\beta = \epsilon$, where $\mathsf{getPointer}$ computes a pointer to the staking key vks. Algorithm 6 defines the non malleable construction and Lemmas 1 and 2 prove that our scheme is collision resistant and non-malleable.

Algorithm 6 The non malleable address generation function, parameterized by $\mathcal{H}(\cdot)$ and $\Sigma = \langle \mathsf{KeyGen}, \mathsf{Verify}, \mathsf{Sign} \rangle$. The input is a tuple $l_{\alpha,Gen}$, consisting of the auxiliary information aux and the address's attributes.

```
function NMGenAddr(l_{α,Gen})
    (aux, st, vkp, skp) = parse(l_{α,Gen})
    switch aux do
        case "base"
            β = H(st)
        case "pointer"
            β = getPointer(st)
        case "exile"
            β = st
    α = vkp||β||Sign(skp, β)
    return α
end function
```

Lemma 1. $\mathsf{NMGenAddr}$ *is collision resistant if* $\mathcal{H}$ *is collision resistant and* Σ *is EUF-CMA.*

Proof. Suppose two different attribute lists $l_1 = (aux_1, st_1, \mathsf{vkp}_1, \mathsf{skp}_1)$, $l_2 = (aux_2, st_2, \mathsf{vkp}_2, \mathsf{skp}_2)$, such that $l_1 \neq l_2$, which correspond to the addresses $\alpha_1 = \mathsf{NMGenAddr}(l_1) = \mathsf{vkp}_1||\beta_1||\mathsf{Sign}(\mathsf{skp}_1, \beta_1)$ and $\alpha_2 = \mathsf{NMGenAddr}(l_2) = \mathsf{vkp}_2||\beta_2||\mathsf{Sign}(\mathsf{skp}_2, \beta_2)$. If $\alpha_1 = \alpha_2$ then either $st_1 \neq st_2$ or $\mathsf{Sign}(\mathsf{skp}_1, \beta_1) = \mathsf{Sign}(\mathsf{skp}_2, \beta_2)$ (otherwise $\mathsf{skp}_1 \neq \mathsf{skp}_2$ which is impossible). If the latter holds, then an adversary is able to produce the same signature for two different pairs of key/message $(\mathsf{skp}_1, \beta_1)$ and $(\mathsf{skp}_2, \beta_2)$, which should be impossible. If the former holds, then the addresses are either base or pointer addresses. If they are base addresses, then $\mathcal{H}(st_1) = \mathcal{H}(st_2)$, i.e. a collision has been found. If they are pointer addresses, then they point to the same position on the ledger, i.e. $st_1 = st_2$ which is a contradiction. □ □

Lemma 2. $\mathsf{NMGenAddr}$*, parameterized with a signature scheme* Σ*, is attribute non-malleable if* Σ *is EUF-CMA.*

Proof. Assume the key pair $(\mathsf{vkp}, \mathsf{skp})$ and the address $\alpha = \mathsf{vkp}||\beta||\mathsf{Sign}(\mathsf{skp}, \beta)$, for staking object β. Also assume the existence of an adversary $\mathcal{A}$ who breaks the attribute non-malleability property of $\mathsf{NMGenAddr}$. We will construct a forger F for the signature scheme, which simulates the security game for $\mathcal{A}$. The forger works as follows: F receives a key vkp and has access to the signing oracle. It sets the attribute list $l = (\mathsf{vkp}, aux, \beta, wrt)$ and initializes $\mathcal{A}$ with public attributes (aux, β, wrt). Note that F answers generation address queries by using its own signature oracle. That is, upon receiving (aux_i, β_i, wrt_i), it issues a signature query on β_i and generates a new address α_i. Moreover, F may receive a metadata query on issued addresses α_i and answer by revealing vkp. By hypothesis, the adversary $\mathcal{A}$ outputs a list $(\alpha^*, aux^*, \beta^*, wrt^*)$, following the definition of the attribute non-malleability game, for which it holds that $\mathsf{NMGenAddr}(\mathsf{skp}, \mathsf{vkp}, aux^*, \beta^*, wrt^*) \rightarrow \alpha^*$, such that α^* was not queried during the game and $\alpha^* \neq \alpha$, where α is the original address of the challenge $\mathsf{NMGenAddr}(\mathsf{skp}, \mathsf{vkp}, aux, \beta, wrt) \rightarrow \alpha$. Since $\mathcal{A}$ is successful, the signature holds for both α and α^*, so F uses $\alpha^* = \mathsf{vkp}||\beta^*||\sigma^*$ to output (α^*, σ^*) as its pair of forged message and signature. □ □

D Integration of Core-Wallet with Ouroboros Praos

Section 5 provided a generic model of a PoS ledger. Now we further explore our wallet construction in conjunction with an existing PoS protocol. In particular, given its decentralized nature and support for delegation, we find Ouroboros Praos [17] a suitable study case for our framework. Furthermore, our framework and Ouroboros Praos enable both stake pool registration and the cold/hot wallet techniques detailed earlier.

The execution of Ouroboros Praos is divided in time-slots, each associated with a single block generated by the parties that run the protocol. A fixed number of time-slots is called an *epoch*. In a nutshell, for each time-slot within the epoch a particular token is selected via the consensus protocol. The owner of the token or, in our setting the player who has the delegation rights over it, is expected to generate the block. In our ledger abstraction, this selection mechanism is described by the function $F_{PoS,player}(\cdot)$.

A key component of Ouroboros Praos is the Verifiable Random Function (VRF) introduced in [34]. Its use is intrinsically related to the choice of the slot leader, i.e. the issuer of the block for a particular time-slot. Briefly, a VRF allows a player to evaluate a value x, given a secret key sk, and compute a pseudorandom value y and a proof π of the computation. Other players can then verify the generation of y using π, the initial input x, and the corresponding verification key vk. Following our abstraction, the function $F_{PoS,player}(\cdot)$ outputs the pair (y, π) based on the earlier epochs.

Pool Registration with Previously Delegated Stake. First, the pool operator publishes a registration certificate $\Sigma = (r, \sigma)$, where $r = (\mathsf{vks}_P, m)$. This is achieved via a transaction clearly marked for pool registration and therefore with a fixed fee Φ_{reg}. The metadata m includes information regarding the stake pool, e.g. real-world information regarding its operator. Following, the pool controls its initial and its delegated stake; the delegation is achieved via the heavyweight certificates described in Section 5. Regarding the initial stake, we require m to store additional information:

- a list verification keys $(\mathsf{vks}_1, \mathsf{vks}_2, \dots)$, corresponding to the players who pledge their stake to the pool prior to its registration;
- the VRF verification key vk used to authenticate the issuing of each block.

Since the registration certificate describes the pool's initial stake, it requires the signature of all $(\mathsf{vks}_1, \mathsf{vks}_2, \dots)$. For extra security, the pool's key $(\mathsf{vks}_P, \mathsf{sks}_P)$ has to be a *cold key*,

otherwise the sks_P has to be accessed every time a block is issued. Finally, the registration of the pool should contain the address α_{reg}, which is controlled by the pool's operator and is used to deposit the rewards for the participation in the protocol.

Hot Evolving and Cold Regular Key Pairs. In the hot/cold wallet mechanism, the pair $(\mathsf{vks}_P, \mathsf{sks}_P)$ is a *cold key*, i.e. is kept offline, whereas the pair $(\mathsf{vks}_{Ph}, \mathsf{sks}_{Ph})$ is exposed online and is used for signing blocks. This mechanism raises the need of generating a lightweight certificate to authenticate the creation of the block by the proof operator. Now, the pair $(\mathsf{vks}_{Ph}, \mathsf{sks}_{Ph})$ can be instantiated using a *Key Evolving Signature* scheme (KES) in the spirit of [5], as employed in [17]. Briefly, the secret key sks_{Ph} of a KES scheme can be continuously evolved, i.e. each instance $\mathsf{sks}_{Ph,i}$ is related to the time period i. Therefore, each new block contains a lightweight certificate signed by the corresponding instance $\mathsf{sks}_{Ph,j}$ for the correct time j.

E Delegation Chains

Chain delegation is the ability of a staking key to re-delegate stake that has been delegated to it. As described in Section 5.1, the metadata section of a delegation certificate defines the rules that pertain to the certificate. One such rule relates to *chain delegation*, the metadata entry for which is the boolean value "allowChain", which identifies whether chain delegation is allowed to occur for the stake on which the certificate applies. For example, suppose two certificates Σ_0 and Σ_1, such that Σ_0 delegates from the key vks_0 to a key vks_2 and states "allowChain" $= true$, whereas Σ_1 delegates from vks_1 to the same delegate key vks_2 and states "allowChain" $= false$. Suppose now that a third certificate Σ_3 is published, with source key vks_2 and delegate key vks_3. In this case, although vks_3 can stake for all addresses associated with vks_0, it cannot stake for those associated with vks_1, since the key that can stake for them, according to the delegation chain rules, is vks_2. More concretely, a *delegation chain* is a list of certificates $[\Sigma_1, \ldots, \Sigma_i]$ such that, for each certificate $\Sigma_j, 1 < j \leq i$, it holds that $\Sigma_{j-1}[\mathsf{vks}_{delegate}] = \Sigma_j[\mathsf{vks}_s]$. We say that vks *is delegated via a chain* $[\Sigma_1, \ldots, \Sigma]$ if $\Sigma[\mathsf{vks}_{delegate}] = \mathsf{vks}$.