Secure and Practical Cold (and Hot) Staking

Mario Larangeira[1,2][a]
[1]*Institute of Science Tokyo, School of Computing, Ookayama 2-12-1, Meguro-ku, Tokyo, Japan*
[2]*Input Output Global (IOG), Singapore*

Keywords: Delegated Proof of Stake, Stake Delegation, Hardware Wallet, Hot Wallet, Cold Wallet.

Abstract: The stake delegation technique is what turns the general Proof of Stake (PoS) into a practical protocol for a large number of participants, ensuring the security of the distributed system, in what is known as Delegated PoS (DPoS). Karakostas et al. (SCN '20) formalized the delegation method paving the way for a whole industry of stake pools by proposing a formal definition for wallet as a universal composable (UC) functionality and introducing a corresponding protocol. On the other hand, a widely used technique named *hot/cold wallet* was formally studied by Das et al. (CCS '19 and '21), and Groth and Shoup (Eurocrypt '22) for different key derivation methods in the Proof of Work (PoW) setting, but not PoS. Briefly, while hot wallets are exposed to the risks of the network, the cold wallet is kept offline, thus more secure. However this may impair some capabilities given that the cold wallet is kept indefinitely offline. It is straightforward to observe that this "double wallet" design is not naturally portable to the setting where delegation is paramount, i.e., DPoS. This work identifies challenges for PoS Hot/Cold Wallet and proposes a secure and practical protocol.

1 INTRODUCTION

The notion of "addresses" transacting with each other was popularized by Bitcoin. Each address is the hash value of the ECDSA verification key. In comparison with Proof of Work (PoW), typically, Proof of Stake (PoS) systems are more intensive in the use of cryptographic machinery. For example, whereas PoW relies mostly on signature scheme and hash function, generally PoS systems rely also in certificates (for delegation) and Verifiable Random Functions for committee and leader elections. (Karakostas et al., 2020) was the first to propose a universally composable wallet, following the Universally Composability (UC) Framework, for PoS. This development is foundational for Delegated PoS and it has led to the appearance of a stake pool economy with thousands of nodes underpinning the Cardano Blockchain. An example of the wallet design framework with its security supported by thousands of stake pools (Cardano, 2024).

On the other hand, Ethereum has chosen a rather different approach for its migration from PoW to PoS, *i.e.*, Ethereum 2.0. It relies on users allocating their stakes, and blocking them during a period of time, a significant change from (Karakostas et al., 2020), where the user can freely transact their tokens and the systems acknowledges the amount of stake distribution during leader election. The DPoS setting is significantly different from the PoW one as (Karakostas et al., 2020) remarks. It seems the wallet security from one setting could not be trivially adapted to the other. An example is the hot/cold technique formalized in (Das et al., 2019, Das et al., 2021) which, in a nutshell, allows the combination of two wallets: one with network access, hence the *hot wallet*, and the *cold wallet*, isolated and therefore secure. The crucial point is that the hot wallet can generate new cold wallet addresses in an "on demand" fashion, keeping the cold wallet offline via key derivation. How the delegation of the cold wallet, defined for PoW, where most of the funds are stored, works is unclear in DPoS.

This restriction refrains the cold wallet from spending, *or be drained from in the case of an attack*, the stored funds, because the wallet never comes online. However, given the ingenious design of the key derivation (Das et al., 2019, Das et al., 2021), the permanently offline state *does not refrain* the cold wallet of receiving new funds. Furthermore, the cold wallet is often a *hardware wallet* given its typical extra security features. However it is not a mandatory requirement. That is the cold wallet can be a regular (offline) software client.

Such Hot/Cold approach for PoS based wallets,

[a] https://orcid.org/0000-0001-7168-898X

Larangeira, M.
Secure and Practical Cold (and Hot) Staking.
DOI: 10.5220/0013458200003979
In *Proceedings of the 22nd International Conference on Security and Cryptography (SECRYPT 2025)*, pages 339-346
ISBN: 978-989-758-760-3; ISSN: 2184-7711

like those in (Karakostas et al., 2020), still remains unexplored. This literature gap is surprising given the startling design differences between the wallets in (Karakostas et al., 2020) and Bitcoin. In the light of the former, the addresses also encode a richer set of attributes, given is heavy use cryptographic machinery, and this capability does exist for regular ECDSA PoW addresses. Therefore it is far from clear whether the security analysis of (Das et al., 2019, Das et al., 2021), which is based solely for PoW paradigm, preserves the security properties for DPoS setting, as shown in (Karakostas et al., 2020). The very question

"How to perform the hot and cold wallet technique in DPoS?"

remains not fully answered. This works fills this gap.

DPoS Cold/Hot Staking Challenges. In the single wallet setting, *a hot wallet* setting, it is straightforward to receive the rewards and perform delegation and *redelegation* as it is a matter of issuing a new certificate with a new signature. In the case of double wallet setting, forwarding the rewards would be straightforward since it is trivial to assign the rewards target, in the certificate, to the address of the cold wallet, *i.e.*, a cold address. On the other hand, it is unclear how the funds of the cold wallet can be staked since the signature in the certificate corresponds to the hot wallet. The early cited limitations suggest a major rework on the delegation framework proposed in (Karakostas et al., 2020) to accommodate hot/cold wallet setting is necessary. It is also paramount that a novel design does not void the security guarantees offered by that work. Thus, our early outlined research question can be translated to the following questions

- **Cold Stake Delegation.** How the stake of the cold wallet is delegated, if the hot wallet signs the certificate?
- **Rewards Payment.** How rewards are payed to the cold wallet?
- **Cold Stake Redelegation.** How redelegation is done? Does it require the cold wallet to be online?
- **Composable Security.** Does a new hot/cold wallet construction design realizes the functionality in (Karakostas et al., 2020)?

Related Works. As briefly described, (Karakostas et al., 2020) proposed a thorough formalization of the addresses syntax and delegation method for PoS ledgers. In particular, its addresses system supports attributes embedded into the address strings. Moreover, it introduces two types of keys for (1) signing transactions and (2) performing stake delegation. In the PoW setting, the hot/cold wallet was first studied by Das *et al.* (Das et al., 2019) in the setting of deterministic wallets (Maxwell et al., 2014), and their work focused on the key derivation of the ECDSA signature scheme which is widely employed for transactions in blockchain systems and, in particular, their stateful deterministic wallet design. In a follow up work, Das *et al.* (Das et al., 2021) presented a formal security analysis of the BIP32 wallet standard as it is.

This work build upon the security model introduced in (Das et al., 2019). Despite of not mandatory, the cold wallet is often a hardware wallet, *i.e.*, piece of hardware with extra countermeasures against side-channels. Such class of wallets also received a formal treatment in (Arapinis et al., 2019) by Arapinis *et al.* in the UC Framework. Finally a spendable cold wallet was, in fact, proposed with hardware aid (Dowsley et al., 2022), *i.e.*, optical channel, and is based in (Das et al., 2019). Unfortunately it is not designed for the DPoS setting. Nonetheless, the hardware aid could be adapted for the use in DPoS.

Our Contribution. In a nutshell, this work investigates the feasibility of relying on a similar design to the Hot/Cold Wallet in the DPoS setting. As a main requirement in this investigation, it is necessary to identify possible shortcomings in porting the Hot/Cold design from PoW to DPoS in current designs. Given this first step, we proceed in constructing a possible wallet/delegation framework to overcame such shortcomings. Our contribution is three fold

- to identify limitations of the current Hot/Cold Wallet design (Das et al., 2019) from the PoW setting to the DPoS (Karakostas et al., 2020), *i.e.*, perform stake delegation with a hot/cold wallet in PoS;
- to introduce a novel wallet design, *i.e.*, the protocol π_{HC} and the functionality $\mathcal{F}_{\mathrm{HC}}$, in order to support stake delegation, at the same time is offers enhanced security guarantees based on cold storage;
- to present a formal and thorough security analysis in the UC Framework, and prove that our PoS Hot/Cold Wallet Protocol π_{HC} UC realizes the UC Functionality $\mathcal{F}_{\mathrm{HC}}$.

2 PRELIMINARIES

In the next definitions, we consider negl to be a negligible function with respect to the security parameter κ. Moreover, in general we also assume that the adversary is Probabilistic Polynomial Time (PPT) also with respect to κ, and an $\mathsf{EUF\text{-}CMA}$ secure digital signature scheme $\Sigma = \langle \mathsf{Gen}, \mathsf{Verify}, \mathsf{Sign} \rangle$.

2.1 The Stake Delegation in PoS

One of the key points in the of the core wallet design π_{Wallet} of (Karakostas et al., 2020), is that the address system relies on two pairs of keys: one for spending transactions while the other is for the delegation. That is, $(\mathsf{sk}^{pay}, \mathsf{pk}^{pay})$ for payment, *i.e.*, signing regular transactions, and $(\mathsf{sk}^{stk}, \mathsf{pk}^{stk})$ for delegation, *i.e.*, signing of the delegation certificate.

Attributes and Addresses. Along with both pairs of keys, for staking and paying, (Karakostas et al., 2020) relies on collision resistance hash function, and introduces procedures to generate addresses α and assignment attributes to them: $\mathsf{GenAddr}$ and $\mathsf{HKeyGen}$. For completeness, now we review these properties.

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

We now review the properties specific for the framework in (Karakostas et al., 2020) with respect to the address generation. Intuitively, they are designed to capture (and avoid) adversarial attempts to derivate malicious address strings from honest ones.

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)**.** For a key generation function $\mathsf{HKeyGen}(\cdot)$ and signature scheme $\Sigma = \langle \mathsf{Gen}, \mathsf{Verify}, \mathsf{Sign} \rangle$, $\mathsf{HKeyGen}(\cdot)$ is hierarchical for Σ if, for all w, the key distribution produced by $\mathsf{HKeyGen}(w)$ is computationally indistinguishable from the key distribution produced by Gen.

The address generation depends on the key and meta information attributes (Karakostas et al., 2020). For completeness, we recall the following definition.

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

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

for probabilities over the random coins of $\mathsf{GenAddr}$ and the PPT adversary $\mathcal{A}$, and choices of l.

The access to the oracles $\mathsf{GenAddr}^d$ and $\mathsf{GenMeta}^d$ captures the scenario where the adversary has access to a source of well formatted addresses and metadata which may contain useful information.

PoS Wallet Interface and Stake Pool. As mentioned earlier, the PoS design requires more cryptography machinery than the PoW approach, therefore the wallet π_{Wallet}, as devised in (Karakostas et al., 2020), introduces separate actions for *payment*, *delegation* (by putting forward PAY and STAKE interfaces), and also *stake pool registration*. For the latter, their design introduces space for metadata *meta* and makes use of a regular payment transaction whose syntax we introduce next for completeness.

Payment. It is the transfer of Θ assets from a sender's to receiver's addresses α_s and α_r via transaction $tx = (\Theta, \alpha_s, \alpha_r, meta)$, with metadata *meta*, *i.e.*, fee amount. Next it accesses the PAY interface of π_{Wallet}, retrieves the signed (by the payment secret key sk^{pay}) tx, and publishes it on the ledger.

Stake Pool Registration. Every pool is identified by a registered staking key. That is, the pool operator relies on π_{Wallet} to produce a new staking key pair $(\mathsf{sk}^{stk}_{pool}, \mathsf{pk}^{stk}_{pool})$, which the operator receives pk^{stk}_{pool}. It, then, creates a registration certificate $\mathcal{C}_{reg} = (\mathsf{pk}^{stk}_{pool}, meta)$, where *meta* is the pool's metadata, *e.g.*, the name of the pool's operator. The certificate is signed by π_{Wallet} with sk^{stk}_{pool}, and the signature is returned. The certificate $\mathcal{C}_{reg}$ and the received signature are published on the ledger (in a payment transaction), thus registering pk^{stk}_{pool} on behalf of the pool.

Delegation/Redelegation. The stake delegation enables a wallet to assign a stake pool to perform staking on its behalf. It is based on delegation certificates like $\mathcal{C}_{del} = (\mathsf{pk}^{stk}_{user}, \langle \mathsf{pk}^{stk}_{pool}, meta \rangle)$. Namely, pk^{stk}_{user} is the staking key to which the certificate applies, *i.e.*, the key which controls the stake of an address, pk^{stk}_{pool} is the delegate's key, *i.e.*, the registered key of a stake pool, and *meta* is the certificate's metadata. The wallet gives $\mathcal{C}_{del}$ to π_{Wallet} to be signed with sk^{stk}_{user}, then it publishes $\mathcal{C}_{del}$ and the signature via a payment transaction (as in the stake pool registration).

2.2 The Stateful Hot/Cold Wallet

For completeness, we review (Das et al., 2019). The following definition is important because we rely on it in order to keep the key in the cold storage synchronized with the hot wallet via key derivation.

Definition 5 (Stateful Hot/Cold Wallet Scheme (Das et al., 2019))**.** For a security parameter κ, a Stateful Hot/Cold Wallet is defined by the algorithms $\Sigma_{HC} = (\mathsf{MGen}, \mathsf{SKDer}, \mathsf{PKDer}, \mathsf{WSign}, \mathsf{WVer})$ such that

- $\mathsf{MGen}(1^\kappa, \rho) \rightarrow (\mathtt{st}_0, \mathtt{msk}, \mathtt{mpk})$:The probabilistic master key generation algorithm MGen takes as input the security parameter 1^κ and randomness ρ, then outputs an initial state $\mathtt{st}_0$ that is given to both hot and cold wallets, a master public key mpk that is given to the hot wallet, and a master secret key msk that is given to the cold wallet;
- $\mathsf{SKDer}(\mathtt{msk}, \mathtt{st}, \mathsf{id}) \rightarrow (\mathsf{sk}_{\mathsf{id}}, \mathtt{st}')$: The deterministic secret key derivation algorithm SKDer is run by the cold wallet to derive a session secret key. It takes as input the master secret key msk, the state st, identifier id, and outputs a session secret key $\mathsf{sk}_{\mathsf{id}}$ and the new state $\mathtt{st}'$;
- $\mathsf{PKDer}(\mathtt{mpk}, \mathtt{st}, \mathsf{id}) \rightarrow (\mathsf{pk}_{\mathsf{id}}, \mathtt{st}')$: The deterministic public key derivation algorithm PKDer is run by the hot wallet to derive a session public key. It takes the master public key mpk, the state st and an identifier id, and outputs a session public key $\mathsf{pk}_{\mathsf{id}}$ and the new state $\mathtt{st}'$;
- $\mathsf{WSign}(\mathtt{m}, \mathsf{sk}) \rightarrow \sigma$: The probabilistic signing algorithm WSign is executed by the cold wallet. It takes as input a message m and a session secret key sk and outputs a signature σ;
- $\mathsf{WVer}(\mathtt{m}, \sigma, \mathsf{pk}) \rightarrow \{0,1\}$: The deterministic verification algorithm WVer is executed by any party that wants to verify the validity of a signature. It takes as input a session public key pk, a message m and a signature σ, and outputs 0 or 1.

We stretch the EUF-CMA security definition for signature to Definition 5 by saying that Σ_{HC} is EUF-CMA if it shows similar properties.

Correctness of a Stateful Hot/Cold Wallet Scheme. The correctness requirement guarantees that if the cold/hot wallets derive session key pairs on the same set of identifiers $\mathsf{id}_1, \ldots, \mathsf{id}_n$ and in the same order (departing from the initial state $\mathtt{st}_0$), then any signature created by the cold wallet using one of the session secret keys $\mathsf{sk}_{\mathsf{id}_i}$ should be accepted when the verification algorithm is executed with the corresponding session public key $\mathsf{pk}_{\mathsf{id}_i}$. In other words, all the derived session secret and public keys should match.

Security of the Stateful Hot/Cold Wallet Scheme. As described by Das *et al.* (Das et al., 2019), the Stateful Hot/Cold Wallet Scheme should satisfy two security properties: *Unlinkability* and *Unforgeability*. Intuitively, the former protects the privacy of the transaction receiver and captures the guarantee that it should be infeasible to link transactions sending funds to different session public keys that were derived from the same master public key.

It is required that an adversary that is given the master public key cannot distinguish between session public keys derived from that master public key, and session public keys that are generated from a fresh (*i.e.*, independently and randomly chosen) master public key. As highlighted in (Das et al., 2019), such security property cannot be achieved for keys which the adversary knows the state used to derive them (the adversary can learn such state if there is a breach in the hot wallet, for example).

The Unforgeability property captures the feature that once funds are transferred to the cold wallet, they remain secure if: (1) the hot wallet is breached; and (2) the adversary observes transactions signed by the cold wallet. Even such adversary cannot generate new valid transactions spending cold wallet funds.

3 PoS HOT/COLD LIMITATIONS

The current design, outlined in Section 2, presented features that can be used to perform delegation in the hot/cold wallet setting. On the other hand, some features are missing and others are not adequate. We thoroughly discuss each one of them next.

- **Cold Stake Delegation.** As pointed in Section 2.1, the delegation is crucially dependent of the publication of the delegation certificate $\mathcal{C}_{del} = (\mathsf{pk}^{stk}_{user}, \langle \mathsf{pk}^{stk}_{pool}, meta \rangle)$ and σ_{del} generated by sk^{stk}_{user}. The construction to be presented has to set the *cold wallet* key pairs, *i.e.*, for staking and payment, and manage them accordingly;
- **Rewards Payment.** The delegation certificate, *i.e.*, $\mathcal{C}_{del} = (\mathsf{pk}^{stk}_{user}, \langle \mathsf{pk}^{stk}_{pool}, meta \rangle)$, and $meta$ can contain the (see next *Cold staking redelegation*) cold wallet address to where the rewards can be moved. The pool operator uses it to deposit the rewards;
- **Cold Stake Redelegation.** Every new delegation, *i.e.*, for redelegation, a new address can be issued under the control of the cold wallet, *i.e.*, controlled by sk^{stk}_{user};
- **Composable Security.** The protocol π_{Wallet} from (Karakostas et al., 2020) seems to be incompatible to the Stateful Hot/Wold Wallet, *i.e.*, Definition 5, in the current form, which suggests a re-

design of π_{Wallet} in order to realize the security definition of (Karakostas et al., 2020);
- **Hot Wallet Funds.** Given that the delegation is performed by certificate publication in the ledger via a regular (payment) transaction, the hot wallet, *i.e.*, sk^{pay}, must to have some funds in order to pay (and also stake) regular fees for publication.

In summary, given the earlier discussion, it is not clear that it is possible to port the current stateful hot/cold wallet design into DPoS.

4 WALLET PROTOCOL π_{HC}

The main motivation of our next definition is to fill the gaps discussed in Section 3 by introducing our protocol π_{HC}. We start from the basic intuition.

4.1 Intuition: Four Pairs of Keys

We rely on four pairs of keys: $(\mathsf{sk}^{stk}_{hot}, \mathsf{pk}^{stk}_{hot})$, $(\mathsf{sk}^{pay}_{hot}, \mathsf{pk}^{pay}_{hot})$, and $(\mathsf{sk}^{stk}_{cold}, \mathsf{pk}^{stk}_{cold})$ are key pairs from a secure regular (ECDSA) signature, whereas $(\mathtt{msk}^{pay}_{cold}, \mathtt{mpk}^{pay}_{cold})$ is a Stateful Hot/Cold Wallet key pair in the sense of Definition 5. Thus the latter is generated as outlined in Section 2.2:
- ECDSA key pair $\mathtt{msk}^{pay}_{cold} = x$ and $\mathtt{mpk}^{pay}_{cold} = x \cdot G$, for uniformly random x;
- Derivation information generation: $w \leftarrow \mathcal{H}(\mathsf{id}, ch)$, for a "chaincode" ch and index id;
- Keys derivation: $\mathsf{pk}^{pay}_{c,\mathsf{id}} \leftarrow \mathtt{mpk}^{pay}_{cold} + w \cdot G$ and $\mathsf{sk}^{pay}_{c,\mathsf{id}} \leftarrow \mathtt{msk}^{pay}_{cold} + w$, for a group generator G;
- Existence of a deterministic function $I : \mathbb{N} \rightarrow \{0,1\}^*$, such that the cold wallet can check the equality $I(i) = \mathsf{id}_i = \mathsf{id}^*$ for sequential values i.

4.2 Key/Address Basic Procedures

As expected, our protocol is based on the one from (Karakostas et al., 2020). The main difference is that here we have to capture two wallets, namely, the hot and the cold one. Hence, we extend the protocol from (Karakostas et al., 2020), with the option of distinguishing the option between the wallets.

The main idea is that a user $\mathcal{U}$ has access to π_{HC} for the basic procedures for a wallet: initialize, pay (signing transaction), verify transaction, staking, generating address, recovery, etc.

As a preparation to introduce our full construction for π_{HC}, we present the concrete parameters for the protocol, which were originally proposed in (Karakostas et al., 2020).

- GenAddr (Algorithm 1): It is the malleable address generation function [1] which generates the address strings. Concretely it is given by Algorithm 1, and it supports three types of addresses [2]: "base", "pointer" and "exile";

Algorithm 1: (GenAddr). The malleable address generation function, parameterized by $\mathcal{H}(\cdot)$ and the Stateful Hot/Cold Wallet, *i.e.*, Definition 5. The input is a tuple l_α of the auxiliary information *aux* and attributes for address α.

function $\mathsf{GenAddr}(l_\alpha)$
  $(aux, st, \mathsf{sk}^{pay}, \mathsf{pk}^{pay}, wallet) = \mathsf{parse}(l_\alpha)$
  switch("aux''") **do**
    case("$base''$") : $\beta = \mathcal{H}(st)$
    case("$pointer''$") : $\beta = \mathsf{getPointer}(st)$
    case("$exile''$") : $\beta = st$
  $\alpha = \mathcal{H}(\mathsf{pk}^{pay}_{wallet} || \mathsf{pk}^{stk}_{wallet} || \mathcal{H}(\mathsf{pk}^{pay}_{wallet})) || \beta || \mathcal{H}(\mathsf{pk}^{pay}_{wallet})$
  return α

It is not hard to verify the following lemma.

Lemma 1. GenAddr is collision resistant if $\mathcal{H}$ is collision resistant.

- HKeyGen (Algorithm 2): It is the procedure which is used in combination of the GenAddr. For every new address generation, the generation relies on a derivation of a signature public/secret key pair. In our construction, we use this procedure in addition to the Stateful Hot/Cold Wallet, *i.e.*, Definition 5.

Algorithm 2: (HKeyGen). Hierarchical Key Generation, relies on a Pseudorandom Function (PRF), and Pseudorandom Number Generation (PRG). It is designed for $label \in \{$"pay","stk"$\}$ and $wallet \in \{$"hot","$cold$"$\}$.

function
$\mathsf{HKeyGen}(\kappa, r, \langle label, wallet, index\rangle, (mpk, st))$
  if label="pay" $\wedge$ wallet ="cold" **do**
    return
      $\mathsf{PKDer}(mpk, st, index) = (\mathsf{pk}^{pay}_{c,index}, st')$
  else
    $v \leftarrow PRF(r || label || wallet || index)$
    $\rho \leftarrow PRNG(v)$
    return$\mathsf{Gen}(1^\kappa, \rho) = (\mathsf{sk}^{label}_{wallet}, \mathsf{pk}^{label}_{wallet})$

[1] In (Karakostas et al., 2020), this function is referred as Ex Post Malleability type of address, and it is one of various degrees of malleability supported by their framework. Later the functionality receives an equivalente malleability predicative.

[2] The extra types of addresses are to support different entities like regular user, *i.e.*, "base", stake pools, *i.e.*, "pointer", not participants of PoS consensus, *i.e.*, "exile".

- RTagGen: It is the Recovery *Tag* Generation algorithm used in the recovery procedure of the wallet. The tags are generated by hashing the input with the collision resistant hash function $\mathcal{H}$.

4.3 Cold Storage and the π_{HC} Wallet

The cold storage and the wallet π_{HC} work in pair. While the former keeps the cold key $\mathtt{msk}_{cold}$, and therefore sk^{pay}_{cold}, the latter keeps remaining keys, and can perform the hierarchical key generation for address issuing. The initial cold key state, along with the $\mathtt{mpk}_{cold}$ and session key id, is passed to π_{HC}.

Concretely, $\mathcal{U}$ executes $\mathsf{MGen}(1^{\kappa},\rho) \rightarrow (\mathtt{st}_0, \mathtt{msk}^{pay}_{cold}, \mathtt{mpk}^{pay}_{cold})$ for a randomness ρ, pick a random id, and send $(\text{INIT}, sid, \mathtt{st}_0, \mathtt{mpk}^{pay}_{cold}, \mathsf{id})$ to π_{HC} as it can be seen in the *Initialization Interface* of π_{HC} next. The address generation starts by picking "child attributes" by the jargon of (Karakostas et al., 2020).

Protocol π_{HC}

Initialization:
Upon receiving $(\text{INIT}, sid, \mathtt{st}_0, \mathtt{mpk}^{pay}_{cold}, \mathsf{id})$ from $\mathcal{U}$, set $r \xleftarrow{\$} \{0,1\}^{\kappa}$ and return (INITOK, sid).

Address Generation:
-Upon receiving the message $(\text{GENERATEADDRESS}, sid, aux, wallet)$ from $\mathcal{U}$, compute the index and "child" attributes as follows: i) pick $i \leftarrow I$; ii) if wallet="cold", set $store = (\mathtt{mpk}^{pay}_{cold}, \mathtt{st}_0)$ and $index = \mathsf{id}$, else set $store = \perp$ and $index = i$ iii) set $(\mathsf{pk}^{chd,pay}_{wallet}, \mathsf{sk}^{chd,pay}_{wallet}) = \mathsf{HKeyGen}(\kappa, r, \langle$ "pay", $wallet, index\rangle, store)$; iv) set $wrt = \mathsf{RTagGen}(\mathsf{pk}^{chd,pay}_{wallet})$.
-If $aux =$ "base", set $(\mathsf{pk}^{chd,stk}, \mathsf{sk}^{chd,stk}) = \mathsf{HKeyGen}(\kappa, r, \langle$"stk", $wallet, i\rangle, \perp)$; else, if $aux = (\text{"pointer"}, \mathsf{pk}^{stk})$,find $(\mathsf{pk}^{chd,stk}, \mathsf{sk}^{chd,stk}) \in K : \mathsf{pk}^{stk} = \mathsf{pk}^{chd,stk}$; else if $aux =$ "exile", set $(\mathsf{pk}^{chd,stk}, \mathsf{sk}^{chd,stk}) = (\perp, \perp)$.
-Then insert $l_{\alpha} = \langle \mathsf{pk}^{chd,stk}, wrt, aux, \mathsf{pk}^{chd,pay}_{wallet}, \mathsf{sk}^{chd,pay}_{wallet}, \mathsf{sk}^{chd,stk}_{wallet}, wallet\rangle$, generate new address α as $\alpha = \mathsf{GenAddr}(\langle aux, \mathsf{pk}^{stk}_c, \mathsf{sk}^{pay}_c, wrt\rangle)$, and insert the tuple $\langle \alpha, (\mathsf{pk}^{pay}_c, \mathsf{sk}^{pay}_c)\rangle$ to P^{wallet}_{key}.
-Return $(\text{ADDRESS}, sid, \alpha)$ to $\mathcal{U}$. If $aux =$ "base" also insert $(\mathsf{pk}^{chd,stk}_{wallet}, \mathsf{sk}^{chd,stk}_{wallet})$ to S^{wallet}_{key} and send the message $(\text{STAKINGKEY}, sid, \mathsf{pk}^{chd,stk}_{wallet})$ to $\mathcal{U}$.

Wallet Recovery: *// For both cold and hot*
Upon receiving the message $(\text{RECOVERWALLET}, sid, wallet, i_{max})$ from $\mathcal{U}$, $\forall i \in [0, i_{max}]$ set $(\mathsf{pk}^{pay}_i, \mathsf{sk}^{pay}_i) = \mathsf{HKeyGen}(\kappa, r, \langle$"pay", $wallet, i\rangle, store)$, $store = \{(\mathtt{mpk}^{pay}_{cold}, \mathtt{st}_0), \perp\}$, return the message $(\text{TAG}, sid, \mathsf{RTagGen}(\mathsf{pk}^{pay}_i))$ to $\mathcal{U}$.

Address Recovery: *// For both cold and hot*
Upon receiving the message $(\text{RECOVERADDR}, sid, wallet, \alpha, i_{max})$ from $\mathcal{U}$, parse the address's attributes $(\mathsf{pk}^{stk}, wrt, aux) = parsePubAttrs(\alpha)$. If exists $i \in I : i < i_{max}$, where $(\mathsf{pk}^{pay}_i, \mathsf{sk}^{pay}_i) = \mathsf{HKeyGen}(\kappa, r, \langle$"pay", $wallet, i\rangle, store)$, for $store = \{(\mathtt{mpk}^{pay}_{cold}, \mathtt{st}_0), \perp\}$ and $\mathsf{RTagGen}(\mathsf{pk}^{pay}_i) = wrt$, and then return $(\text{RECOVEREDADDR}, sid, \alpha)$.

Issue Hot Transaction: *// Only hot payment*
Upon receiving the message $(\text{PAY}, sid, tx = (\Theta, \alpha_s, \alpha_r, m))$ from $\mathcal{U}$, find $\langle \alpha_s, (\mathsf{pk}^{pay}, \mathsf{sk}^{pay})\rangle \in P^{hot}_{key}$ and return the message $(\text{TRANSACTION}, sid, tx, \mathsf{Sign}(\mathsf{sk}^{pay}, tx))$.

Issue Cold Transaction: *There is no interface because sk^{pay}_{cold} is disconnected in cold storage.*

Verify Transaction: *// For both cold and hot*
Upon receiving $(\text{VERIFYPAY}, sid, tx, \sigma)$ from $\mathcal{U}$, with $tx = (\Theta, \alpha_s, \alpha_r, meta)$ for some metadata *meta*, find an entry $\langle \alpha_s, (\mathsf{pk}^{pay}, \mathsf{sk}^{pay})\rangle$ in P^{wallet}_{key}, for either $wallet \in$ {"*hot*","*cold*"}, and return o $\mathcal{U}$ the message $(\text{VERIFIEDPAY}, sid, tx, \sigma, \mathsf{WVer}(tx, \sigma, \mathsf{pk}^{pay}))$.

Issue Staking: *// Staking secret key is accessible*
Upon receiving $(\text{STAKE}, sid, stx, wallet)$ from $\mathcal{U}$ such that $stx = (\mathsf{pk}^{stk}, meta)$, for some metadata *meta* and find $(\mathsf{pk}^{stk}, \mathsf{sk}^{stk}) \in S^{wallet}_{key}$ for either $wallet \in \{hot, cold\}$, then return $(\text{STAKED}, sid, stx, \mathsf{Sign}(\mathsf{sk}^{stk}, stx))$.

Verify Staking: *// For both cold and hot*
Upon receiving $(\text{VERIFYSTAKE}, sid, stx, \sigma)$ from party $\mathcal{U}$, where $stx = (\mathsf{pk}, meta)$ for some metadata *meta*, then find $(\mathsf{pk}^{stk}, \mathsf{sk}^{stk}) \in S^{wallet}_{key}$, for $wallet \in \{hot, cold\}$, return $(\text{VERIFIEDSTAKE}, sid, stx, \sigma, \mathsf{WVer}(stx, \sigma, \mathsf{sk}^{stk}))$.

Remark. The trick is that the cold storage only stores the cold secret key for payment. When sk^{pay}_{cold} is stored, it is in fact storing the new state $\mathtt{st}'$ to be used in the new key derivation. In order to perform the online redelegation, we only need the cold secret key for *staking*. That is, we do not need the key for signing a payment transaction (from the cold wallet). Thus the double key structure, for payment and staking, comes handy. In order words we can freely do (re)delegation even with the cold wallet offline permanently.

5 SECURITY ANALYSIS

In (Karakostas et al., 2020), the main security definition is the wallet functionality $\mathcal{F}_{\text{Wallet}}^{M}$, which receives, as a parameter, the malleability predicative[3] M, in order to propose a flexible definition for different degrees of address malleability. That is, how much the adversary can reuse the same address with different staking keys; the so called malleability issue identified in (Karakostas et al., 2020).

Unfortunately, $\mathcal{F}_{\text{Wallet}}^{M}$ does not support hot/cold design, therefore from now we proposed a redesign of $\mathcal{F}_{\text{Wallet}}^{M}$ into $\mathcal{F}_{\text{HC}}^{M}$ in order to analyze the previously presented wallet protocol π_{HC}.

5.1 The Functionality $\mathcal{F}_{\text{HC}}$

Before presenting our functionality $\mathcal{F}_{\text{HC}}$, for completeness, we recall the malleability predicate M which is a parameter for $\mathcal{F}_{\text{HC}}$.

Algorithm 3: The malleability predicate M keeps list of generated addresses and transactions, $\mathbb{L}$ and $\mathbb{T}$, and verifies for rightfully generated addresses.

function $M^{\mathbb{L},\mathbb{T},\mathcal{U}}(aux,\alpha)$
  switch("aux")
    case("issue") : **return** 1
    case("verify" *OR* "recover") : $d =$ parsePubAttrs(α)
      for $\delta \in d$ **do**
        if $\forall \alpha' \in L_{\mathcal{U}} \in ,d' =$ parsePubAttrs(α'): $\delta \notin d'$ **then return** 0
        if $\forall(\Theta,\alpha_s,\alpha_r,m) \in \mathbb{T}, d_s =$ parsePubAttrs(α_s): $\delta \notin d_s$ **then return** 0
      return 1
  return 0

The early predicate M is used as a parameter in the next functionality. Its role is to specify the level of address malleability we consider in the functionality.

Now, we present the adapted functionality $\mathcal{F}_{\text{HC}}^{M}$.

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

Initialization: On receiving the message $(\text{INIT}, sid, \mathtt{st}_0, \mathtt{mpk}_{cold}^{pay}, \mathsf{id})$ from $P \in \mathbb{P}$, forward it to $\mathcal{S}$ and wait for (INITOK, sid). Then initialize the empty lists L_P of addresses and attribute lists and K_P of staking keys, return (INITOK, sid).

Address Generation: On receiving the message $(\text{GENERATEADDRESS}, sid, aux, wallet)$ from $P \in \mathbb{P}$, forward it to $\mathcal{S}$. On $(\text{ADDRESS}, sid, \alpha, l_\alpha)$ 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", pk^{stk}), check that $\delta_1 = \mathsf{pk}^{stk}$,
- 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 $(\text{ADDRESS}, sid, \alpha)$ to P. If $aux =$ ("base") also insert δ_1 to K_P and return the message $(\text{STAKINGKEY}, sid, \delta_1)$ to P.

Wallet Recovery: Upon receiving the message $(\text{RECOVERWALLET}, sid, wallet, i)$ from $P \in \mathbb{P}$, for the first i elements in L_P return $(\text{TAG}, sid, \delta_2)$.

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

Issue Hot Transaction: Upon receiving $(\text{PAY}, sid, tx = (\Theta, \alpha_s, \alpha_r, m))$ from $P \in \mathbb{P}$, if $\exists l_\alpha : (\alpha_s, l_\alpha) \in L_P$ forward it to $\mathcal{S}$. Upon receiving $(\text{TRANSACTION}, sid, tx, \sigma)$ from $\mathcal{S}$, 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 $\mathcal{T}$, return $(\text{TRANSACTION}, sid, tx, \sigma)$.

Verify Transaction: Upon receiving $(\text{VERIFYPAY}, sid, tx, \sigma)$ from $P \in \mathbb{P}$, with $tx = (\Theta, \alpha_s, \alpha_r, m)$ for a metadata string m, forward it to $\mathcal{S}$ and wait for $(\text{VERIFIEDPAY}, sid, tx, \sigma, \phi)$. 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 $(\text{VERIFIEDPAY}, sid, tx, \sigma, f)$ to P.

Issue Staking: Upon receiving the message $(\text{STAKE}, sid, stx, wallet)$ from P, such that $stx = (\mathsf{pk}^{stk}, m)$ for a metadata string m, forward the message to $\mathcal{S}$. On $(\text{STAKED}, sid, stx, \sigma)$ from $\mathcal{S}$, if $\forall(stx',\sigma',b') \in S : \sigma' \neq \sigma$, $(stx,\sigma,0) \notin S$, and $\mathsf{pk}^{stk} \in K_P$, add $(stx,\sigma,1)$ to S and return the message $(\text{STAKED}, sid, stx, \sigma)$ to P.

Verify Staking: Upon receiving

[3](Karakostas et al., 2020) proposed various predicative parameters for different malleability degrees. We will consider here in this work the *"sink" malleable* predicate, although we drop the term "*sink malleable*" for readability.

$(\text{VERIFYSTAKE}, sid, stx, \sigma)$ from $P \in \mathbb{P}$, forward it to $\mathcal{S}$ and wait for the message $(\text{VERIFIEDSTAKE}, sid, stx, \sigma, \phi)$, with $stx = (\mathsf{pk}^{stk}, m)$. Then find P_s such that $\mathsf{pk}^{stk} \in K_{P_s}$. Then
- 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 are finally ready to present our main theorem.

5.2 Main Theorem

Theorem 1. Let the protocol π_{HC} be parameterized by a Stateful Hot/Cold Wallet Σ_{HC} (Definition 5) and the HKeyGen, GenAddr (Algorithms 2 and 1), and RTagGen Functions. Then π_{HC} securely realizes the ideal functionality $\mathcal{F}_{\text{HC}}^{M}$ if and only if Σ_{HC} is EUF-CMA, GenAddr is collision resistant and attribute non-malleable (Definitions 2 and 4), RTagGen is collision resistant (Definition 1), and HKeyGen is hierarchical for Σ_{HC} (Definition 3).

We refer the reader the full version of the paper for the complete proof.

5.3 Addressing Cold/Hot Limitations

Our proof directly shows that it is possible to fulfill all the items from Section 3. In particular, three items, namely 1) **Composable security**, 2) **Cold stake redelegation** and 3) **Cold stake delegation** which seemed more challenging. To that purpose, our proposed protocol π_{HC} crucially exploits the design of the delegation framework: keys for staking and payment. The address issuing and delegation certificate signing require only the cold wallet staking secret key, which can be managed by the hot wallet.

6 FINAL REMARKS

We introduced the first Hold/Cold DPoS wallet protocol π_{HC} and the respective security definition, *i.e.*, Functionality $\mathcal{F}_{\text{HC}}$. We further showed that π_{HC} is secure under this security definition in the UC Framework. Our protocol addresses the limitations we have found, however it relies on less restrictive malleability level (as in (Karakostas et al., 2020)). For more restrictive levels, note that our protocol supports hardware based solutions in the spirit of (Dowsley et al., 2022). Our protocol was shown to allow **Cold stake delegation** and **Rewards payment**, in addition to **Composable security** and the trivially achievable **Hot wallet funds** criteria from Section 3.

Even more crucially, our protocol is shown to perform **Cold stake redelegation** even with a (permanently) off-line wallet, *i.e.*, the cold wallet. It is worth to highlight that our technique exploits the double key design of (Karakostas et al., 2020): a key pair for staking alone and the other for payment transactions for each wallet, with a total of four key pairs.

We leave for future work extending this study to the framework which models hardware wallets, *i.e.*, (Arapinis et al., 2019), to more restrictive levels of address malleability, *i.e.*, (Karakostas et al., 2020), and designs with hardware aid to access the cold wallet (Dowsley et al., 2022).

ACKNOWLEDGEMENTS

This work was supported by JSPS KAKENHI under Grant JP21K11882.

REFERENCES

Arapinis, M., Gkaniatsou, A., Karakostas, D., and Kiayias, A. (2019). A formal treatment of hardware wallets. In Goldberg, I. and Moore, T., editors, *FC 2019*, volume 11598 of *LNCS*, pages 426–445. Springer, Cham.

Cardano (2024). Cardano explorer. https://https://cexplorer.io/.

Das, P., Erwig, A., Faust, S., Loss, J., and Riahi, S. (2021). The exact security of BIP32 wallets. In Vigna, G. and Shi, E., editors, *ACM CCS 2021*, pages 1020–1042. ACM Press.

Das, P., Faust, S., and Loss, J. (2019). A formal treatment of deterministic wallets. In Cavallaro, L., Kinder, J., Wang, X., and Katz, J., editors, *ACM CCS 2019*, pages 651–668. ACM Press.

Dowsley, R. B., Farias, M. C., Larangeira, M., Nascimento, A. C., and Virdee, J. (2022). A spendable cold wallet from qr video. In *International Conference on Security and Cryptography 2022*, pages 283–290. Scitepress.

Karakostas, D., Kiayias, A., and Larangeira, M. (2020). Account management in proof of stake ledgers. In Galdi, C. and Kolesnikov, V., editors, *SCN 20*, volume 12238 of *LNCS*, pages 3–23. Springer, Cham.

Maxwell, G. et al. (2014). Deterministic wallets.