Arcula:
A Secure Hierarchical Deterministic Wallet for Multi-asset Blockchains

Adriano Di Luzio*, Danilo Francati, and Giuseppe Ateniese

Stevens Institute of Technology, USA
{adiluzio,dfrancat,gatenies}@stevens.edu

June 13, 2019

Abstract

This work presents Arcula, a new design for hierarchical deterministic wallets that significantly improves the state of the art. Arcula is built on top of provably secure cryptographic primitives. It generates all its cryptographic secrets from a user-provided seed and enables the derivation of new signing public keys without requiring any secret information. Unlike other wallets, it achieves all these properties while being secure against privilege escalation. We prove that an attacker compromising an arbitrary number of users within an Arcula wallet cannot escalate his privileges and compromise users higher in the access hierarchy. Our design works out-of-the-box with any blockchain that enables the verification of signatures on arbitrary messages. We evaluate its usage in a real-world scenario on the Bitcoin Cash network.

Keywords. Hierarchical Deterministic Wallet; Hierarchical Key Assignment; Bitcoin; Blockchain.

1 Introduction

In recent years, the adoption of blockchain-based crypto-systems grew at an exponential rate. At their core, these systems create a free, decentralized, and democratic financial world where users exchange their assets and keep track of their credits and debits with the others without relying on any central authority. With the adoption of these systems, however, we also face a new set of technological and financial challenges that we have never experienced before: How to secure our digital assets, how to protect our financial privacy, and, when we are required to, how to enable an auditor to evaluate our past transactions or to gather the blockchain assets of a financial company.

In this work, we aim at solving these challenges. In particular, we focus on hierarchical deterministic wallets. Similarly to how we keep our coins and our bills in a physical wallet, a blockchain wallet holds all the crypto-currencies of an individual (*e.g.*, her Bitcoins, Ethers, and other coins). In particular, wallets usually hold the cryptographic keys that allow spending these coins. Our goal is to design a wallet that is cryptographically secure, easy to use, and that also guarantees some useful properties, *e.g.*, safeguarding the privacy of users or allowing

*The author is also a PhD Candidate at Sapienza University of Rome, Italy.

the auditing of the transactions of a public company. In addition, we aim at creating a wallet that is hierarchical and deterministic. This means that: 1) The users can organize their keys in a hierarchy that reflects, for example, the division in departments of a company. This way, the financial manager of the company can spend funds on behalf of the departments, but each department can only spend its own funds. 2) They deterministically generate every cryptographic key of the wallet by starting from an initial seed (*e.g.*, a pseudorandom sequence derived from a mnemonic that is easy to remember or to store in a safe). As a result, the users can reliably recover all their keys even in the case they lose the entire wallet (*e.g.*, after a hardware failure or a natural disaster).

To this end, we present Arcula, a hierarchical deterministic wallet named after the small casket where ancient Romans used to store their jewels. We build Arcula on a deterministic variation of hierarchical key assignment schemes, that generates all the cryptographic secrets in a deterministic way, implements arbitrarily complex access hierarchies, and allows for their dynamic modifications. Arcula ties the identities of users to their public signing keys without requiring additional secret information — as identity-based hierarchical signatures but explicitly designed for the existing blockchains. Unlike other wallets, Arcula achieves all these properties while being formally secure against privilege escalation. Besides, it relies on simple cryptographic primitives and does not depend on any particular digital signature scheme. As a consequence, Arcula is compatible with any blockchain that enables the signature verification of arbitrary messages. We show its implementation in Bitcoin Cash, a fork of the original Bitcoin crypto-system.

The rest of this work is organized as follows. In Section 2 we describe in more details the properties of hierarchical deterministic wallets and we discuss the state of the art, BIP32. Section 3 introduces the notation that we use throughout this paper and the required cryptographic primitives. In Section 4 we present a new deterministic key assignment scheme that we leverage at the core of our wallet. Section 5 and Section 6 respectively describe our design of Arcula and its usage in the real world. Sections 7 and 8 detail how we handle dynamic changes to the hierarchy of the wallet and how we incorporate some temporal constraints into its design. Section 9 discusses the related works and Section 10 concludes the paper.

2 Hierarchical Deterministic Wallets

In this section, we provide a more detailed definition of hierarchical deterministic wallets, and we show their intended use cases. We formalize the properties that they require, we introduce the state of the art implementation, and we point out some of its issues and vulnerabilities.

A hierarchical deterministic wallet enables a user to securely generate and store all the cryptographic keys associated with her assets. In general, blockchain-based cryptosystems rely on digital signatures and pairs of private and public signing keys: Users spend their assets by signing the corresponding transaction with the private key; others verify the authenticity of the signature through the public key. Public keys can be typically derived from the corresponding private keys, but not vice-versa. As a result, on a high level, a hierarchical deterministic wallet stores a collection of private signing keys.

The whole wallet should rely on a single seed provided by the user, and the generation of the keys should be deterministic so that users can recover their keys in case of wallet loss. Besides, the private keys of the wallet should be organized under an access hierarchy, *e.g.*, a directed acyclic graph or a tree. Each element of the hierarchy corresponds to a group of users and a pair of signing keys associated with them. The privileges of a group of users depend on their level in the hierarchy. Users with higher privileges (*i.e.*, higher in the hierarchy) should be able to derive the keys of users on lower levels and in turn to sign messages (*i.e.*, transactions) on their behalf. Users on lower levels, however, should not be able to escalate their privileges to the

higher levels of the hierarchy, not even when colluding with others.

Formally, let sk_i be the private signing key of a node v_i of a hierarchy and let pk_i be the corresponding public key. Let sk_j and pk_j be, respectively, the private and public keys associated with a j-th child v_j of v_i. A hierarchical deterministic wallet shall have the following properties:

Property 2.1 (Security to Key Recovery)**.** It is computationally infeasible to recover the private key sk_i by starting from any children private key sk_j.

Property 2.2 (Deterministic Generation)**.** All the sk_j private keys of a subtree are deterministically generated from the private key sk_i of v_i.

Property 2.3 (Public Key Derivation)**.** All the pk_j public keys of a subtree are deterministically generated from the public key pk_i of v_i, without requiring the knowledge of its private key sk_i.

Property 2.1 guarantees the security of the wallet so that any arbitrary number of colluding users lower in the hierarchy cannot escalate their privileges and recover the key of a user higher in the hierarchy. Property 2.2 guarantees that the entire wallet is generated deterministically from an initial seed and, in turn, that users will not lose their assets as long as they remember the seed. Property 2.3 requires every public key to be derivable without relying on any secret information. Such property trades the privacy of the users for some additional capabilities within the blockchain, *e.g.* to allow the auditing of the funds in the wallet or the generation of new public keys in an insecure environment. As an example, an online marketplace could leverage this property to generate a fresh public address for each of its customers in an untrusted web server, while keeping its secret keys in cold storage. An attacker compromising the web server would not compromise the secret keys and would not be able to spend its funds. Similarly, an auditor could leverage the same generation process to inspect all the holdings of the online marketplace. As we will see, BIP32, the state of the art implementation of hierarchical deterministic wallets, fails to satisfy all these properties at the same time. Arcula instead, our design of HDW, achieves them all and also allows its users to dismiss the public derivation property to safeguard their privacy.

In particular, we evaluate a hierarchical deterministic wallet within the following three-levels security model, sorted according to increasing requirements of trust:

Untrusted Environment: The superficial level of the model is entirely untrusted. The web server of an online-based merchant fits into this level. As an example, the merchant aims at associating a different public key to each object in its catalog. When adding a new object to the store, he leverages the public key derivation (Property 2.3) to derive a new public key for the object without storing any sensitive information on the untrusted web-server. Similarly, the merchant could use the public key derivation to generate a fresh address for every incoming payment without storing the corresponding secret keys on the web-server. An auditor that aims at inspecting the transactions and the holdings of the wallet also fits in this environment. By providing her with the public key at the root of the hierarchy, the users enable the auditor to inspect the wallet without revealing any secret information.

Warm Environment: The middle level of the security model is semi-trusted. The different departments of a large company fit into this level. The headquarters of the company might structure the hierarchy of the wallet to reflect the different departments, with the root corresponding to the main office and the children to the departments. Each department autonomously manages its own subtree and its assets. Nonetheless, no department can spend on behalf of the main office or of the other departments. Compromising any node of the warm environment corresponds to compromising its subtree of the hierarchy and nothing more.

Cold Storage: The cold storage is the deepest level of the security model. It is entirely trusted and, usually, it is disconnected from the network, and it also is physically secured. An example of cold storage is a hardware token storing the deterministic seed that generates the entire wallet. As a result, compromising the cold storage implies compromising the entire wallet and any asset associated with its accounts.

2.1 BIP32: Bitcoin hierarchical deterministic wallets

The state of the art implementation of hierarchical deterministic wallets has been defined in Bitcoin Improvement Proposal 32 (BIP-0032) in 2012 [18]. It builds on the same public key cryptography used in Bitcoin, Elliptic Curve Cryptography using the field and curve parameters defined by the standard `secp256k1`. Let g be the generator point of an Elliptic Curve defined over a group a cyclic group $\mathbb{G}$ of order q. A private key sk (of 256 bits in Bitcoin) is associated with its public key $\mathsf{pk} = g^{\mathsf{sk}}$. Let H be a hash function, then the children private keys sk_i are derived as follows:

$$\mathsf{sk}_i = \mathsf{H}(\mathsf{pk} \| i) + \mathsf{sk} \mod q \tag{1}$$

Children public keys pk_i, instead, are derived as follows (we omit the $\mod q$ operators for shortness):

$$\begin{aligned} \mathsf{pk}_i &= g^{\mathsf{sk}_i} \\ \mathsf{pk}_i &= g^{\mathsf{H}(\mathsf{pk}\|i)+\mathsf{sk}} \\ \mathsf{pk}_i &= g^{\mathsf{H}(\mathsf{pk}\|i)} \cdot g^{\mathsf{sk}} \\ \mathsf{pk}_i &= g^{\mathsf{H}(\mathsf{pk}\|i)} \cdot \mathsf{pk} \end{aligned} \tag{2}$$

Equations (1) and (2) respectively satisfy the properties of deterministic generation and public derivation (Properties 2.2 and 2.3). However, BIP32 is not secure against key recovery (Property 2.1), because Equation (1) creates a key recovery vulnerability, where the knowledge of a children private key sk_i and the root public key pk allows recovering the root private key sk. From Equation (1):

$$\begin{aligned} \mathsf{sk}_i &= \mathsf{H}(\mathsf{pk}\|i) + \mathsf{sk} \mod q \\ \mathsf{sk} &= \mathsf{sk}_i - \mathsf{H}(\mathsf{sk}_i\|i) \mod q \end{aligned}$$

This privilege escalation vulnerability has been discussed in the BIP32 specification and also by several security researchers [6, 7, 14]. Essentially, the derivation defined in Equation (1) leverages an invertible operation. For this reason, all the secret keys of a BIP32 wallet rely on a single secret value, sk, shared between all nodes. As a result, when we discuss the security of BIP32 in our model the cold storage and warm environment collapse into a single level: Compromising any individual node immediately leads to compromising the entire wallet.

The BIP32 proposal addresses this issue by designing an additional key derivation method, that generates a so-called *hardened* private key sk_i^h, as follows:

$$\mathsf{sk}_i^h = \mathsf{H}(\mathsf{sk}\|i) + \mathsf{sk} \mod q$$

The hardened key generation process trades the property of public key derivation for security to key recovery (*i.e.*, Property 2.3 for Property 2.1), since generating a new hardened public key pk_i^h now requires knowledge of the root secret key sk:

$$\mathsf{pk}_i^h = g^{\mathsf{sk}_i^h} = g^{\mathsf{H}(\mathsf{sk}\|i)+\mathsf{sk}} = g^{\mathsf{H}(\mathsf{sk}\|i)} \cdot \mathsf{pk}$$

As a result, BIP32 fails to satisfy the three required properties all at the same time.

BIP32 also adds 256 bits of additional entropy to the generation process. This entropy aims at improving the privacy of the wallet so that there is no straightforward connection between the children and parent public keys. More in details, BIP32 defines an extended private key as the pair (sk, c), where c is called the chain code and is an integer of 256 bits (the entropy). The generation process takes as input the parent's extended private key (sk, c) and generates the extended private key (sk_i, c_i) of the i-th children. It replaces the hash function H of the previous equations with a pseudorandom function F_c implemented with HMAC-SHA512. The resulting output is 512 bits long and its most and least 256 bits respectively encode the integers sk_i and c_i. More in details, given the pair (sk, c) the BIP32 generation function outputs a child key (sk_i, c_i) as follows:

$$\begin{aligned} H &= \begin{cases} \mathsf{F}_c(g^{\mathsf{sk}} \| i) & \text{if } i \text{ is not a hardened child} \\ \mathsf{F}_c(\mathsf{sk} \| i) & \text{if } i \text{ is a hardened child} \end{cases} \\ H_L, c_i &= H[:256], H[256:] \\ \mathsf{sk}_i &= H_L + \mathsf{sk} \mod q \end{aligned} \tag{3}$$

The generation of the extended key (sk, c) of the root of the hierarchy starts from a seed provided by the user, S_{-1}, and proceeds to derive the secret key and the chain code by replacing Equation (3) with $H = \mathsf{F}_c(S_{-1})$ where c is the hardcoded string `Bitcoin Seed`.

In short, we formalize the three following scenarios related to the security and privacy of BIP32, differing in whether the chain code c of the wallet is a public parameter and on whether hardened key derivation is enabled. To the best of our knowledge, this is the first formal definition of the security and privacy scenarios of BIP32 and any hierarchical deterministic wallet supporting public-key derivation.

Public Chain Code: We begin by studying the scenario where the chain code c is a public parameter of the wallet, and the hardened key derivation is disabled. In this setting, the public key generation happens in an entirely untrusted environment and, as a result, can be run by any individual of the system (including those outside the wallet). For this reason, there is no pseudo-anonymity associated with the public keys of the wallets. An external, passive observer can correlate together with the public keys of the wallet by repeating the public-key derivation.

Private Chain Code: In this scenario, the hardened key derivation is still disabled, but the chain code c is a private parameter of the wallet and has to be explicitly provided to any party that aims at running the key derivation (*e.g.*, an auditor or a web-server that generates a fresh address for each payment). As long as c is private any pair of public keys of the wallet looks uncorrelated; as such, there is a weak pseudo-anonymity. In particular, the public keys look uncorrelated to any passive observer. Nonetheless, an active adversary can compromise the web-server (that we assume to be insecure), uncover the chain code c, and in turn reveal the correlation between the public keys.

Hardened Derivation: The last scenario considers hardened key derivation. In this case, deriving the public key of any children requires the knowledge of both the chain code (that we assume public) and the private key of its father. For this reason, uncovering a correlation between any pair of public keys is as hard as compromising their corresponding private keys. However, this scenario does not allow public key derivation in an untrusted or semi-trusted environment.

Finally, we point out the following issues of BIP32. First, it constrains the access hierarchy to a tree structure. Equations (1) and (2) do not handle the key derivation of any node with more than one predecessors in the hierarchy. Many use cases, however, require a more complex access hierarchy (*i.e.*, any directed acyclic graph). For example, consider two or more departments of a company that collaborate on a project. They have a shared budget and need to spend from the shared budget without revealing their own secret key. An access hierarchy encoding this context would derive a single node, the budget, as a successor of multiple others (the departments). Besides, we also note that in the **Public Chain Code** model, where we derive the public keys in an untrusted environment, the entropy added by BIP32 serves no purpose. The key derivation of Equation (3) requires indeed the chain code c for the public key generation. The attacker has unrestricted access to the public derivation and, as such, in an untrusted environment, the chain code is equivalent to a public parameter of the wallet. In the real world, we assume the **Private Chain Code** model, where the chain code protects the privacy of the users of the wallet only against a passive attacker. An active attacker, instead, could obtain it in several ways: *E.g.*, by compromising the web-server of an online marketplace, by impersonating an auditor, or by corrupting any user of the wallet. The chain code, in turn, would allow him to uncover the public keys associated with all the nodes of the wallet and to jeopardize the anonymity of its users.

3 Preliminaries

3.1 Notation

We use the notation $[n] = \{1, \ldots, n\}$. Uppercase boldface letters (such as $\mathbf{X}$) are used to denote random variables, lowercase letters (such as x) to denote concrete values, calligraphic letters (such as $\mathcal{X}$) to denote sets, and sans serif letters (such as A) to denote algorithms. All of our algorithms are modeled as (possibly interactive) Turing machines; if algorithm A has access to some oracle O, we often write $\mathcal{Q}_{\mathsf{O}}$ for the set of queries asked by A to O.

For a string $x \in \{0,1\}^*$, we let $|x|$ be its length; $|\mathcal{X}|$ represents the cardinality of the set $\mathcal{X}$. When x is chosen randomly in $\mathcal{X}$, we write $x \leftarrow\!\!\$\, \mathcal{X}$. We write $y = \mathsf{A}(x)$ to denote a run of the algorithm A on input x and output y; if A is randomized, y is a random variable and $\mathsf{A}(x;r)$ denotes a run of A on input x and (uniform) randomness r. We sometimes write $y \leftarrow\!\!\$\, \mathsf{A}(x)$ to denote a run of the randomized algorithm A over the input x and uniform randomness. An algorithm A is *probabilistic polynomial-time* (PPT) if A is randomized and for any input $x, r \in \{0,1\}^*$ the computation of $\mathsf{A}(x;r)$ terminates in a polynomial number of steps (in the input size).

Throughout the paper, we denote by $\lambda \in \mathbb{N}$ the security parameter and we implicitly assume that every algorithm takes as input the security parameter. A function $\nu : \mathbb{N} \to [0,1]$ is called *negligible* in the security parameter λ if it vanishes faster than the inverse of any polynomial in λ, *i.e.*, $\nu(\lambda) \in \mathcal{O}(1/p(\lambda))$ for all positive polynomials $p(\lambda)$. We sometimes write $\mathsf{negl}(\lambda)$ (resp., $\mathsf{poly}(\lambda)$) to denote an unspecified negligible function (resp., polynomial function) in the security parameter.

3.2 Pseudorandom Function (PRF) Family

Let $\{\mathcal{K}_\lambda, \mathcal{X}_\lambda, \mathcal{Y}_\lambda\}_{\lambda \in \mathbb{N}}$ be a sequence of sets. For $\lambda \in \mathbb{N}$, a PRF family $\{\mathsf{F}_k\}_{k \in \mathcal{K}_\lambda}$ is a set of functions such that $\mathsf{F}_k : \mathcal{X}_\lambda \to \mathcal{Y}_\lambda$ and each function is evaluable by a deterministic polynomial time algorithm F, *i.e.*, $\mathsf{F}(k, \cdot) = \mathsf{F}_k(\cdot)$.

Let $\mathcal{F}_\lambda$ be the set of all functions from $\mathcal{X}_\lambda$ to $\mathcal{Y}_\lambda$. For security, we require that a function randomly sampled from $\{\mathsf{F}_k\}_{k \in \mathcal{K}_\lambda}$ is indistinguishable by a function randomly sampled from $\mathcal{F}_\lambda$.

Definition 3.1 (Pseudorandomness)**.** A PRF family $\{\mathsf{F}_k\}_{k\in\mathcal{K}_\lambda}$ is pseudorandom if for every PPT adversary A we have:

$$\left|\Pr\left[\mathbf{G}_{\mathsf{F},\mathsf{A}}^{\mathsf{prf}-0}(\lambda)=0\right]-\Pr\left[\mathbf{G}_{\mathsf{F},\mathsf{A}}^{\mathsf{prf}-1}(\lambda)=1\right]\right|\leq \mathsf{negl}(\lambda)\,,$$

where the two experiments $\mathbf{G}_{\mathsf{F},\mathsf{A}}^{\mathsf{prf}-0}(\lambda)$ and $\mathbf{G}_{\mathsf{F},\mathsf{A}}^{\mathsf{prf}-1}(\lambda)$ are defined in the following way:

$\mathbf{G}_{\mathsf{F},\mathsf{A}}^{\mathsf{prf}-0}(\lambda)$	$\mathbf{G}_{\mathsf{F},\mathsf{A}}^{\mathsf{prf}-1}(\lambda)$
$k \leftarrow\!\!\$\, \mathcal{K}_\lambda$	$f \leftarrow\!\!\$\, \mathcal{F}_\lambda$
$d \leftarrow\!\!\$\, \mathsf{A}^{\mathsf{F}_k(\cdot)}(1^\lambda)$	$d \leftarrow\!\!\$\, \mathsf{A}^{f(\cdot)}(1^\lambda)$
return d	**return** d

In this paper, we are interested in PRF families such that $\mathcal{K}_\lambda = \mathcal{Y}_\lambda = \{0,1\}^\lambda$.

3.3 Symmetric Encryption Scheme

We follow the definition of symmetric encryption scheme provided by Atallah *et al.* [3]. A symmetric-key encryption scheme $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ with message space $\mathcal{M}$ is a triple of polynomial-time algorithm defined in the following way:

$\mathsf{Gen}(1^\lambda)$**:** The randomized key generation algorithm takes as input a security parameter 1^λ and outputs a secret key sk.

$\mathsf{Enc}(\mathsf{sk}, m)$**:** The deterministic (possibly randomized) encryption algorithm takes as input a secret key sk, a message $m \in \mathcal{M}$, and outputs a ciphertext c.

$\mathsf{Dec}(\mathsf{sk}, c)$**:** The deterministic decryption algorithm takes as input a secret key sk, a ciphertext c, and outputs a message m.

For correctness, we require that honestly generated ciphertexts must decrypt correctly.

Definition 3.2 (Correctness of symmetric encryption)**.** A symmetric encryption scheme $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ with message space $\mathcal{M}$ is correct if $\forall\lambda \in \mathbb{N}, \forall m \in \mathcal{M}$:

$$\Pr\left[\mathsf{Dec}(\mathsf{sk}, \mathsf{Enc}(\mathsf{sk}, m)) = m \;\middle|\; \mathsf{sk} \leftarrow\!\!\$\, \mathsf{Gen}(1^\lambda)\right] = 1$$

For security, we are interested in semantic security: It must be infeasible to distinguish between an encryption of a message m from one of a random message.

Definition 3.3 (Semantic Security)**.** A symmetric encryption scheme $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ with message space $\mathcal{M}$ is semantically secure if for every PPT adversary A we have:

$$\left|\Pr\left[\mathbf{G}_{\Pi,\mathsf{A}}^{\mathsf{sem}}(\lambda)=1\right]-\frac{1}{2}\right|\leq \mathsf{negl}(\lambda)\,,$$

where $\mathbf{G}_{\Pi,\mathsf{A}}^{\mathsf{sem}}(\lambda)$ is defined in the following way:

Setup: The challenger runs $\mathsf{sk} \leftarrow\!\!\$\, \mathsf{Gen}(1^\lambda)$.

Challenge: The adversary specifies a message $m_0 \in \mathcal{M}$. The challenger picks a random bit $b^* \in \{0,1\}$. If $b^* = 0$, then it computes $c^* \leftarrow\!\!\$\, \mathsf{Enc}(\mathsf{sk}, m_0)$; otherwise, it sets $c^* \leftarrow\!\!\$\, \mathsf{Enc}_{\mathsf{sk}}(m_1)$, where $m_1 \leftarrow\!\!\$\, \mathcal{M}$. The challenger returns c^* to A.

Guess: The adversary outputs a bit $b \in \{0,1\}$. If $b = b^*$ return 1; otherwise return 0.

3.4 Signature Scheme

A signature scheme with message space $\mathcal{M}$ is made of the following polynomial-time algorithms.

$\mathsf{KGen}(1^\lambda)$**:** The randomized key generation algorithm takes the security parameter and outputs a secret and a public key $(\mathsf{sk}, \mathsf{pk})$.

$\mathsf{Sign}(\mathsf{sk}, m)$**:** The randomized signing algorithm takes as input the secret key sk and a message $m \in \mathcal{M}$, and produces a signature σ.

$\mathsf{Vrfy}(\mathsf{pk}, m, \sigma)$**:** The deterministic verification algorithm takes as input the public key pk, a message m, and a signature σ, and it returns a decision bit.

A signature scheme is correct if honestly generated signatures always verify correctly.

Definition 3.4 (Correctness of signatures)**.** A signature scheme $\Pi = (\mathsf{KGen}, \mathsf{Sign}, \mathsf{Vrfy})$ with message space $\mathcal{M}$ is correct if $\forall \lambda \in \mathbb{N}$ and $\forall m \in \mathcal{M}$, the following holds:

$$\Pr\left[\mathsf{Vrfy}(\mathsf{pk}, m, \mathsf{Sign}(\mathsf{sk}, m)) = 1 \;\middle|\; (\mathsf{sk}, \mathsf{pk}) \leftarrow_\$ \mathsf{KGen}(1^\lambda)\right] = 1.$$

As for security we are interested in existential unforgeability, *i.e,* it must be infeasible to forge a valid signature on a new fresh message.

Definition 3.5 (Unforgeability of signatures)**.** A signature scheme $\Pi = (\mathsf{KGen}, \mathsf{Sign}, \mathsf{Vrfy})$ is existentially unforgeable under chosen-message attacks if for all PPT adversaries A:

$$\Pr\left[\mathbf{G}^{\mathsf{euf}}_{\Pi,\mathsf{A}}(\lambda) = 1\right] \leq \mathsf{negl}(\lambda)\,,$$

where $\mathbf{G}^{\mathsf{euf}}_{\Pi,\mathsf{A}}(\lambda)$ is the following experiment:

Setup: The challenger runs $(\mathsf{sk}, \mathsf{pk}) \leftarrow_\$ \mathsf{KGen}(1^\lambda)$ and gives pk to A.

Query: The adversary has access to a signing oracle $\mathsf{O}_{\mathsf{Sign}}(\cdot)$. On input m, the challenger computes and returns $\sigma \leftarrow_\$ \mathsf{Sign}(\mathsf{sk}, m)$. Let $\mathcal{Q}_{\mathsf{Sign}}$ denote the the messages queried to the signing oracle.

Forgery: The adversary outputs (m, σ). If $m \notin \mathcal{Q}_{\mathsf{Sign}}$, and $\mathsf{Vrfy}(\mathsf{pk}, m, \sigma) = 1$, output 1, else output 0.

4 Deterministic Hierarchical Key Assignment

In this section we first introduce hierarchical key assignment (HKA) schemes [3] and then we define a new HKA scheme, entirely deterministic, that we use as a building block in our construction of Arcula. A hierarchical key assignment scheme is a mechanism designed to assign a set of cryptographic keys to a set of users (or, equivalently, to a set of classes of users). It starts from an access hierarchy (*e.g.*, a directed acyclic graph) representing the access rights: A path from a node v_i to a node v_j implies that the user of class v_i, higher in the hierarchy, can assume the same access rights of v_j. Equivalently, the access graph defines a partial order over the users of the hierarchy, where $v_i \geq v_j$ indicates that v_i has higher access privileges than v_j. The goal of a hierarchical key assignment scheme is to enforce the access hierarchy while minimizing the number of keys distributed to the users. HKA schemes share some similarities with hierarchical deterministic wallets. Both mechanisms assign a set of cryptographic keys to the users of a hierarchy; users of an HKA can derive the cryptographic keys of other users lower

in the hierarchy, while users of a hierarchical deterministic wallet aim at signing transactions on their behalf. Despite their similarities, HKA schemes have never been leveraged before to build hierarchical deterministic wallets. To this end, HKA schemes provide several advantages: They have been studied in depth in prior work, are provably secure, and efficient. They can enforce arbitrarily complex access hierarchies (in comparison to the tree hierarchy of BIP32), and consider temporal constraints. For these reasons, a contribution of this work is defining a hierarchical key assignment scheme that is suitable for building hierarchical deterministic wallets.

Typically, HKA schemes sample the cryptographic secrets of the hierarchy at random instead of generating them deterministically as required by Property 2.2. In this work, instead, we design a deterministic hierarchical key assignment (DHKA) scheme where every cryptographic secret is derived from a seed provided by the user. To do so, we propose a deterministic version of the efficient HKA developed by Atallah *et al.* [3] that is secure under key indistinguishability (and as a consequence guarantees Property 2.1 by design). Section 5 shows how we leverage the DHKA scheme as a building block of Arcula.

4.1 Security model

A deterministic hierarchical key assignment scheme is defined as follows (we adhere to the definitions of HKA of Atallah *et al.*[3]). Let $G = (V, E)$ be a directed acyclic graph (DAG) representing an access hierarchy. We define the set of ancestors $Anc(v_i, G) = \{v_j \mid v_j \rightsquigarrow_w v_i\}$ of a node v_i to be the set of nodes v_j such that there exists a direct path w from v_j to v_i in G. Analogously, the set of descendants of node v_j as $Desc(v_i, G) = \{v_j \mid v_i \rightsquigarrow_w v_j\}$ includes any node v_j such that there exists a path w from v_i to v_j in G. When the graph G is clear in a particular context we omit it from the parameters of the functions $Anc(v_i)$ and $Desc(v_i)$.

A Deterministic Hierarchical Key Assignment (DHKA) scheme with seed space $\mathcal{S}$ is composed of the following polynomial-time algorithms:

$\mathsf{Set}(1^\lambda, G, S_{-1})$**:** The deterministic setup algorithm takes as input the security parameter, a DAG $G = (V, E)$, and an initial seed $S_{-1} \in \mathcal{S}$, and outputs two mappings: 1) a public mapping $\mathsf{Pub} : V \cup E \to \{0,1\}^*$, associating a public label l_i to each node v_i in G and a public information y_{ij} to each edge $(v_i, v_j) \in E$; 2) a secret mapping $\mathsf{Sec} : V \to \{0,1\}^\lambda \times \{0,1\}^\lambda$, associating a secret information S_i and a cryptographic key x_i to each node v_i in G. (No secret information is associated to the edges).

$\mathsf{Derive}(G, \mathsf{Pub}, v_i, v_j, S_i)$**:** The deterministic derivation algorithm takes as input the access graph G, the public information Pub, a source node v_i, a target node v_j, and the secret information S_i of node v_i. It outputs the cryptographic key x_j associated to node v_j if $v_j \in Desc(v_i)$.

The correctness of a DHKA scheme requires that any user v_i should be able to derive, correctly, the secret key x_j of any user $v_j \in Desc(v_j)$ lower in the hierarchy.

Definition 4.1 (Correctness of DHKA)**.** A DHKA $\Pi = (\mathsf{Set}, \mathsf{Derive})$ with seed space $\mathcal{S}$ is correct if for every DAG $G = (V, E)$, $\forall \lambda \in \mathbb{N}$, $\forall v_i \in V$, $\forall v_j \in Desc(v_i)$, $\forall S_{-1} \in \mathcal{S}$:

$$\Pr[x_j = \mathsf{Derive}(G, \mathsf{Pub}, v_i, v_j, S_i)] = 1,$$

where $(\mathsf{Pub}, \mathsf{Sec}) = \mathsf{Set}(1^\lambda, G, S_{-1})$, $(S_i, x_i) = \mathsf{Sec}(v_i)$, and $(S_j, x_j) = \mathsf{Sec}(v_j)$.

We now formalize the security level of the scheme. We adapt the security definition originally defined by Atallah *et al.* [3] to account for the determinism in our scheme.

Definition 4.2 (Key Indistinguishability of DHKA)**.** A DHKA $\Pi = (\mathsf{Set}, \mathsf{Derive})$ with seed space $\mathcal{S}$ is key indistinguishable if for every PPT adversary A and every DAG $G = (V, E)$:

$$\left| \Pr\left[\mathbf{G}^{\mathsf{sk-ind}}_{\Pi,\mathsf{A}}(\lambda, G) = 1 \right] - \frac{1}{2} \right| \leq \mathsf{negl}(\lambda)\,,$$

where $\mathbf{G}^{\mathsf{sk-ind}}_{\Pi,\mathsf{A}}(\lambda, G)$ is defined in the following way:

Setup: The challenger receives a challenge node $v^* \in V$ from the adversary A. The challenger samples $S_{-1} \leftarrow\!\!\$\, \mathcal{S}$, then runs $\mathsf{Set}(1^\lambda, G, S_{-1})$, and gives the resulting public information Pub to the adversary A. The challenger samples a random bit $b^* \leftarrow\!\!\$\, \{0,1\}$: If $b^* = 0$, it returns to A the cryptographic key x_{v^*} associated to node v^*; otherwise, it returns a random key $\bar{x}_{v^*}$ of the corresponding length.

Query: The adversary has access to a corrupt oracle $\mathsf{O}_{\mathsf{Corr}}(\cdot)$. On input $v_i \notin Anc(v^*)$, the challenger retrieves $(S_i, x_i) = \mathsf{Sec}(v_i)$ and sends S_i to A.

Guess: The adversary outputs a bit $b \in \{0,1\}$. If $b = b^*$ return 1; otherwise return 0.

Remark. *We note that the adversary* A *depicted in* $\mathbf{G}^{\mathsf{sk-ind}}_{\Pi,\mathsf{A}}(\lambda, G)$ *is a static adversary who chooses the challenge node* v^* *before the experiment begins. Ateniese* et al. *[5, Theorem 1], however, prove that any hierarchical key assignment scheme secure (in the sense of* $\mathbf{G}^{\mathsf{sk-ind}}_{\Pi,\mathsf{A}}(\lambda, G)$*) against a static attacker is also secure against an adaptive attacker,* i.e.*, against an adversary that adaptively chooses the challenge node* v^**. The authors prove that the two security models are polynomially equivalent since there exists a reduction between the static and the adaptive adversaries. The static adversary can simply guess the challenge node* v^* *of the adaptive adversary and abort the simulation if the guess is incorrect. For these reasons, we discuss the security of any DHKA scheme only in the setting of a static attacker.*

4.2 The DHKA scheme

This section describes the implementation of our deterministic hierarchical key assignment scheme over any DAG G encoding an access hierarchy.

We assume, without loss of generality, that: 1) There exists a unique *root* node $v_0 \in V$ of G, *i.e.* the most-privileged node of the hierarchy encoded by G that can derive the keys of any other node. For any DAG G, it is always possible to elect a root node v_0. Since G is a DAG, v_0 shall be one of the minimal nodes in a topological ordering of G and, equivalently, v_0 shall have no ancestors. If two or more nodes v_j have no ancestors, then it is always possible to construct a new graph $G' = (V \cup \{v_0\}, E \cup \{(v_0, v_j) \mid v_j \text{ has no ancestors}\})$ such that the access hierarchy encoded by G' is equivalent to the one of G, where the new node v_0 in G' is the root of the graph (and has no associated users). 2) That every node v_j has a fixed *parent* node in the hierarchy, *i.e.* a node v_i such that the edge $(v_i, v_j) \in E$. As an example, we fix the parent node v_j of v_i to be the first ancestor of v_i in any ordering of the nodes of the graph G (*e.g.*, obtained with a depth-first-search of the graph) such that $(v_i, v_j) \in E$.

At a high level, we build on the randomized hierarchical key assignment scheme of Atallah *et al.* [3] where each node v_i of the hierarchy is identified by a random label l_i and holds a random secret information S_i, that it will use to generate its own cryptographic key and to derive the keys of the nodes lower in the hierarchy. In our scheme, we modify the original design so that both the label l_i and the secret information S_i are deterministic. We label each node through its index[1] (*i.e.*, $l_i = v_i$) and we derive its secret information S_i deterministically

[1] We will extend the node labels with a version number when handling dynamic changes to the hierarchy of the DHKA. We refer the reader to Section 7 for more details.

(through a pseudorandom function) from the secret information of its parent. We formally define our implementation of DHKA as follows.

Construction 1. Let $\{\mathsf{F}_k\}_{k\in\mathcal{K}_\lambda}$ and $\mathcal{E} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be respectively a family of pseudorandom functions and a symmetric key encryption scheme. Let $G = (V, E)$ be a directed acyclic graph representing an access hierarchy. We build a DHKA scheme in the following way:

$\mathsf{Set}(1^\lambda, G, S_{-1})$**:** On input the security parameter, a directed acyclic graph $G = (V, E)$, and an initial seed S_{-1}, the algorithm proceeds as follows:

1. Compute $S_0 = \mathsf{F}_{S_{-1}}(11\|l_0)$ for $v_0 \in V$, where $l_0 = v_0$ and v_0 is the *root* of the directed acyclic graph G.
2. For each vertex $v_i \in V$ and $v_j \in V$ such that v_j is the *parent* of v_i, compute $S_i = \mathsf{F}_{S_j}(11\|l_i)$ where $l_i = v_i$.
3. For each vertex $v_i \in V$ compute $t_i = \mathsf{F}_{S_i}(00\|l_i)$ and $x_i = \mathsf{F}_{S_i}(01\|l_i)$.
4. For each edge $(v_i, v_j) \in E$, compute $r_{ij} = \mathsf{F}_{t_i}(10\|l_j)$ and $y_{ij} \leftarrow\!\!\$\ \mathsf{Enc}_{r_{ij}}(t_j\|x_j)$.[2]

Finally, the algorithm returns the public mapping $\mathsf{Pub} : V \cup E \to \{0,1\}^*$ and the secret mapping $\mathsf{Sec} : V \to \{0,1\}^\lambda \times \{0,1\}^\lambda$, defined as:

$$\mathsf{Pub} : v_i \mapsto l_i \qquad \mathsf{Pub} : (v_i, v_j) \mapsto y_{ij}$$
$$\mathsf{Sec} : v_i \mapsto (S_i, x_i)$$

$\mathsf{Derive}(G, \mathsf{Pub}, v_i, v_j, S_i)$**:** On input a directed acyclic graph $G = (V, E)$, a public mapping Pub, two nodes $v_i, v_j \in V$, and a seed S_i, the algorithm proceeds as follows:

1. If there is no path from v_i to v_j in G, return $\perp$;
2. If $i = j$, retrieve l_i from Pub and return $x_j = \mathsf{F}_{S_i}(01\|l_i)$;
3. Otherwise, compute $t_i = \mathsf{F}_{S_i}(00\|l_i)$ and set $\bar{i} = i$ and $t_{\bar{i}} = t_i$; then
   (a) Let $\bar{j}$ be the successor of $\bar{i}$ in the path from v_i to v_j.
   (b) Retrieve $l_{\bar{j}}$ and $y_{\bar{i}\bar{j}}$ from Pub
   (c) Compute $r_{\bar{i}\bar{j}} = \mathsf{F}_{t_{\bar{i}}}(10\|l_{\bar{j}})$ and $t_{\bar{j}}\|x_{\bar{j}} = \mathsf{Dec}_{r_{\bar{i}\bar{j}}}(y_{\bar{i}\bar{j}})$.
   (d) Set $\bar{i} = \bar{j}$ and $t_{\bar{i}} = t_{\bar{j}}$.
   (e) If $\bar{j} = j$ then return x_j; otherwise repeat from Item 3a.

The proposed key assignment scheme is entirely deterministic. In particular, it differs from the design of Atallah *et al.* at the Items 1 and 2 of the Set algorithm in Construction 1. The original key assignment scheme draws the values l_i and S_i (respectively, l_0 and S_0) at random. In our case, instead, we deterministically derive them from the identifier v_i of the node and from the secret information S_j of its *parent* v_j (respectively, from the seed S_{-1}).

Computation and space complexity The efficiency of the scheme is linear in time and space, respectively, to the key derivation distance and the size of the graph. Let w be the shortest path between v_i and $v_j \in Desc(v_i)$: Deriving x_j by starting from S_i requires $|w|$ invocations F and $|w|$ invocations of Dec. For space complexity, each node v_i in V is required to store a single secret S_i—the private storage required by each node is proportional to the size λ of the security parameter. On the other hand, the public information holds the mapping between nodes and

[2]We implicitly assume that the PRF output space and the symmetric encryption key space have the same distribution. In alternative, r_{ij} can be used as randomness of key generation algorithm Gen.

labels and the encrypted information associated with each edge. As such, the overall space required is linear to $\lambda|V| + \lambda|E|$. That said, we note that in our case the mapping between nodes and labels is the identity function and that we can further reduce the storage requirements by leveraging the deterministic derivation: Any parent node v_j can directly derive the secret information S_i of its descendant v_i and, for this reason, we can avoid storing any encrypted information on the edge that connects them. As a result, we can reduce the size of the encrypted information on the edges and only store them for any node v_i such that there exists an edge $(v_i, v_j) \in E$ and v_i is not the parent of v_j and as such cannot deterministically derive the secret value S_j by starting from its own secret value S_i. With this optimization in place, our scheme is comparable to a tree-based hierarchical key assignment scheme [8] where we store the additional derivation keys as encrypted information on the edges instead of storing them as secrets within each node that requires them. Finally, if the key generation and derivation processes happen on the fly (*i.e.*, when the entire process starts from the seed), then the only private storage required is proportional to the length of the initial seed S_{-1}, *i.e.*, to the length of the security parameter λ.

Remark. *At first glance, it might seem that fixing the randomness of the* Set *algorithm of the HKA by Atallah* et al. *[3] is sufficient to enforce its determinism. We remark here that such solution, alone, does not guarantee this result. When we fix the randomness of the* Set *algorithm of the HKA we are implicitly fixing an ordering on the sampling of the secret values S_i of each node v_i: Sampling at random S_i before S_j, as opposed to sampling S_j before S_i, will result in different secret values assigned to each node. For this reason, the HKA with fixed randomness would also require additional public information about the ordering of the nodes of the hierarchy. Our DHKA, instead, deterministically generates the secret values according to the structure of the hierarchy and not to any ordering of its nodes. This approach allows us to design a deterministic scheme that does not require any additional public information and that, furthermore, can take the determinism into account to reduce the amount of encrypted information stored on the edges of the hierarchy.*

We conclude this section by establishing the following result, that we prove in Appendix A.1.

Theorem 4.1. *Let* $\{\mathsf{F}_k\}_{k\in\mathcal{K}_\lambda}$ *and* $\mathcal{E} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *be respectively a pseudorandom function family and a symmetric encryption scheme. If* $\{\mathsf{F}_k\}_{k\in\mathcal{K}_\lambda}$ *is pseudorandom (Definition 3.1) and* $\mathcal{E}$ *is semantically secure (Definition 3.3), then the DKHA scheme* Π *from Construction 1 is key indistinguishable.*

5 Arcula: Secure Hierarchical Deterministic Wallets

The goal of a hierarchical deterministic wallet (HDW) is to assign a set of signing keys to the users of a hierarchy so that each node can sign transactions on behalf of its children, but can not escalate its privileges to sign transactions on behalf of any node higher in the hierarchy. Also, an HDW requires the public key of a child node to depend exclusively on the public key of its parent and do not require any additional secret information. Combining all these properties at once has proved in the past to be a challenging task. As it happens in BIP32, enabling public key derivation seems to expose the wallet to privilege escalation inevitably.

Arcula, our new design of hierarchical deterministic wallet, solves this issue while satisfying the three properties all at once. It is secure to key recovery, it deterministically derives the private and public information from an initial seed, and it enables public key derivation (*e.g.*, to generate new public keys in an untrusted environment or to allow the optional auditing of the wallet funds). On a high level, Arcula is defined over 5 algorithms $(\mathsf{Set}, \mathsf{DPub}, \mathsf{DPriv}, \mathsf{Sign}, \mathsf{Vrfy})$

that setup the wallet parameters, enable the derivation of the private and public signing keys of the nodes of the hierarchy and implement the creation and the verification of signatures. More in details: 1) Set deterministically instantiates the wallet by generating a single master public key mpk and a set of master secret keys msk_i, one for each node of the hierarchy. 2) DPriv and DPub are responsible for the secret and public key derivation. DPriv derives the secret signing key sk_i of a node v_j descendent of a node v_i by using the master secret key associated to v_i, msk_i (Property 2.2) whereas DPub derives the public key pk_i of a node v_i by starting from the master public key mpk (Property 2.3). 3) Sign and Vrfy are the standard signing and verification algorithms. The following sections first provide a formal definition of hierarchical deterministic wallets and then detail our implementation of Arcula.

5.1 Security Model

A hierarchical deterministic wallet (HDW) $\Pi = (\mathsf{Set}, \mathsf{DPub}, \mathsf{DPriv}, \mathsf{Sign}, \mathsf{Vrfy})$, defined over a seed space $\mathcal{S}$ and message space $\mathcal{M}$ is defined in the following way:

$\mathsf{Set}(1^\lambda, G, S_{-1})$**:** The deterministic setup algorithm takes as input a security parameter, an access graph $G = (V, E)$, and an initial seed $S_{-1} \in \mathcal{S}$, and outputs a set of master secret keys $\{\mathsf{msk}_i\}_{v_i \in V}$, a master public key mpk.

$\mathsf{DPub}(\mathsf{mpk}, v_i)$**:** The deterministic public derivation algorithm takes as input the master public key mpk, a target node v_i, and outputs the public key pk_i associated to node v_i.

$\mathsf{DPriv}(\mathsf{mpk}, \mathsf{msk}_i, v_i, v_j)$**:** The deterministic private derivation algorithm takes as input the master public key mpk, master secret key msk_i of node v_i, and a target node $v_j \in Desc(v_i)$, and outputs the secret key sk_j associated to node v_j.

$\mathsf{Sign}(\mathsf{sk}_i, m)$**:** The randomized signing algorithm takes as input a message $m \in \mathcal{M}$, and a secret key sk_i, and outputs a signature σ.

$\mathsf{Vrfy}(\mathsf{pk}_i, m, \sigma)$**:** The deterministic verification algorithm takes as input a public key pk_i, a message m, and a signature σ, and outputs a decisional bit b.

A hierarchical deterministic wallet is correct if any user can derive the private and public key of its descendants and create a valid signature on behalf of them. More in details, we say that a HDW is correct if any node v_i can derive the signing key sk_j of any node $v_j \in Desc(v_i)$ and produce, in turn, a valid signature σ on behalf of v_j (*i.e.*, that passes the verification process against the public key pk_j obtained through public key derivation).

Definition 5.1 (Correctness of HDW)**.** A hierarchical deterministic wallet $\Pi = (\mathsf{Set}, \mathsf{DPub}, \mathsf{DPriv}, \mathsf{Sign}, \mathsf{Vrfy})$, with seed space $\mathcal{S}$ and message space $\mathcal{M}$, is correct if for every DAG $G = (V, E)$, $\forall v_i \in V$, $\forall v_j \in Desc(v_i)$, $\forall S_{-1} \in \mathcal{S}, \forall m \in \mathcal{M}$ the following conditions holds:

$$\Pr\left[\mathsf{Vrfy}(\mathsf{pk}_j, m, \mathsf{Sign}(\mathsf{sk}_j, m)) = 1\right] \geq 1 - \mathsf{negl}(\lambda)\,,$$

where $(\{\mathsf{msk}_i\}_{v_i \in V}, \mathsf{mpk}) = \mathsf{Set}(1^\lambda, G, S_{-1})$, $\mathsf{sk}_j = \mathsf{DPriv}(\mathsf{mpk}, \mathsf{msk}_i, v_i, v_j)$, and $\mathsf{pk}_j = \mathsf{DPub}(\mathsf{mpk}, v_j)$.

Remark. *Arcula shares a significant number of similarities with hierarchical identity-based cryptography. Its public derivation, indeed, takes as input the identity of a target node (*e.g.*, v_i) and the master public key* mpk *to output the public key* pk_i*. Nonetheless, we point out some fundamental differences. Most hierarchical identity-based signature schemes leverage a number of public parameters and rely on bilinear mappings. For this reason, these solutions are unpractical*

in the existing blockchains. The public parameters of each instantiation of the scheme should be stored on the blockchain itself and, in addition, the underlying protocol should handle the bilinear mappings efficiently. Our design of Arcula, on the other hand, explicitly takes the blockchain into consideration. We do not rely on bilinear mappings, and we only store a portion of the master public key mpk *(typically a single group element) on the blockchain.*

From the security point of view, we take inspiration from existentially unforgeable signatures. We allow an attacker to corrupt an arbitrary number of nodes in the hierarchy and we provide him with a signing oracle to obtain signatures on arbitrary messages from any uncorrupted node (*i.e.,* every node that is not a descendant of a corrupt one). Then, we challenge the attacker to forge a signature for a fresh message on behalf of an uncorrupted node.

Definition 5.2 (Hierarchical existential unforgeability of HDW)**.** A hierarchical deterministic wallet is hierarchically existentially unforgeable under chosen-message attacks if for every DAG $G = (V, E)$ and PPT adversary A the following condition holds:

$$\Pr\left[\mathbf{G}^{\mathsf{heuf}}_{\Pi,\mathsf{A}}(\lambda, G) = 1\right] \leq \mathsf{negl}(\lambda)\,,$$

where experiment $\mathbf{G}^{\mathsf{heuf}}_{\Pi,\mathsf{A}}(\lambda, G)$is defined in the following way:

Setup: The challenger samples a random $S_{-1} \leftarrow\!\!\$\ \mathcal{S}$ and executes $(\{\mathsf{msk}_i\}_{v_i \in V}, \mathsf{mpk}) = \mathsf{Set}(1^\lambda, G, S_{-1})$. It gives the master public key mpk to A.

Query: The adversary A has access to the following oracles:

- $\mathsf{O}_{\mathsf{Corr}}(\cdot)$**:** On input $v_i \in V$, the challenger answers by giving msk_i to A. Let $\mathcal{Q}_{\mathsf{Corr}}$ denote the set of nodes v_i that A corrupted, including their descendants $Desc(v_i)$.
- $\mathsf{O}_{\mathsf{Sign}}(\cdot,\cdot)$**:** On input $(m, v_i) \in \mathcal{M} \times V$, the challenger returns $\sigma \leftarrow\!\!\$\ \mathsf{Sign}_{\mathsf{sk}_i}(m)$ where $\mathsf{sk}_i = \mathsf{DPriv}(\mathsf{mpk}, \mathsf{msk}_0, v_0, v_i)$. Let $\mathcal{Q}_{\mathsf{Sign}}$ denote the pairs (m, v_i) for which A queried the oracle $\mathsf{O}_{\mathsf{Sign}}$.

Forgery: A outputs a forgery (v_i, m, σ). If $\mathsf{Vrfy}_{\mathsf{pk}_i}(m, \sigma) = 1$ where $\mathsf{pk}_i = \mathsf{DPub}(\mathsf{mpk}, v_i)$ and $v_i \notin \mathcal{Q}_{\mathsf{Corr}}$, $(m, v_i) \notin \mathcal{Q}_{\mathsf{Sign}}$, return 1; otherwise return 0.

5.2 Arcula from DHKA and signatures

This section describes our implementation of hierarchical deterministic wallet from a DHKA and a signature scheme. We start from an initial seed to deterministically generate a global pair of *cold storage* keys $(\mathsf{wsk}, \mathsf{wpk})$ and a pair of *signing* keys $(\mathsf{sk}'_i, \mathsf{pk}'_i)$ associated to each node v_i of the wallet. The cold storage secret key wsk is designed to be stored in a safe environment (*e.g.*, offline). The signing keys, instead, are generated through a key indistinguishable DHKA scheme and enable nodes to sign transactions on behalf of their descendants in the hierarchy while guaranteeing security against key recovery attacks. We identify the nodes of the wallet v_i according to the cold storage public key wpk and to their public label l_i and, as a result, we achieve public key derivation by design. Then, we bind the identity of an Arcula node to their the signing key pair $(\mathsf{sk}'_i, \mathsf{pk}'_i)$ through a certificate, signed by the cold storage secret key wsk, that explicitly authorizes the signing key sk'_i to spend the coins destined to the node v_i.

In practice, in a blockchain-based crypto-system, we leverage the cold storage public key wpk and the public label l_i of a node as an address for receiving payments. Users spend their funds by signing transactions through their signing key sk'_i and by presenting, at the same time, the certificate provided by the cold storage that authorizes them to spend funds. This approach has

several advantages: Receiving funds does not require the certificate—all we need is the address of the destination node, *i.e.* the cold storage public key wpk and the public node label l_i; a user only requires her certificate when spending funds for the first time, *i.e.*, when she signs a transaction through her secret key sk'_i; the creation of the certificate that authorizes the signing key sk'_i to spend the coins of node v_i can happen entirely offline (*i.e.*, in cold storage). On the other hand, if a user does not require the public derivation property, she can also leverage her signing public key pk'_i directly as a pseudonym address on which to receive funds and then spend them through the corresponding private key sk'_i. These signing keys, generated by the DHKA, are indistinguishable and, as a consequence, this approach guarantees the unlinkability of the users within the wallet and disables the public address derivation (Property 2.3).

More in details, the Set algorithm deterministically generates a master public key mpk, that we use for the public derivation, and that holds the public cold storage key wpk and the certificates authorizing the signing keys sk'_i, and a master secret key msk_i associated to each node v_i, that we use to derive their signing keys sk'_i. The DPriv and DPub algorithms respectively derive the signing secret keys (through the DHKA scheme) and the corresponding public keys (by combining the public cold storage key wpk and the node identifiers). Finally, the Sign and the Vrfy algorithms handle the creation and the verification of digital signatures. Any node v_i runs the Sign algorithm with its signing key sk'_i to create a signature and the Vrfy algorithm checks that there exists a certificate authorizing sk'_i to spend funds on behalf of the node v_i (identified by l_i) under the cold storage key wsk of the wallet. Formally:

Construction 2. Let $\Gamma = (\mathsf{Set}_\Gamma, \mathsf{Derive}_\Gamma)$ and $\Sigma = (\mathsf{KGen}_\Sigma, \mathsf{Sign}_\Sigma, \mathsf{Vrfy}_\Sigma)$ be respectively a DHKA and a signatures signature scheme. We build Arcula in the following way:

$\mathsf{Set}(1^\lambda, G, S_{-1})$**:** On input the security parameter, a DAG $G = (V, E)$, and a seed $S_{-1} \in \mathcal{S}$ the algorithm proceeds as follows:

1. Let $S^0_{-1}, S^1_{-1} \in \{0,1\}^{|S_{-1}|/2}$ be respectively the most significant and least significant part of S_{-1}. Compute $(\mathsf{Pub}, \mathsf{Sec}) = \mathsf{Set}_\Gamma(1^\lambda, G, S^0_{-1})$ and $(\mathsf{wsk}, \mathsf{wpk}) = \mathsf{KGen}_\Sigma(1^\lambda; S^1_{-1})$.
2. For each node $v_i \in V$:
   (a) Let $(S_i, x_i) = \mathsf{Sec}(v_i)$ and set $\mathsf{msk}_i = S_i$.
   (b) $(\mathsf{sk}'_i, \mathsf{pk}'_i) = \mathsf{KGen}_\Sigma(1^\lambda; x_i)$.
   (c) Compute $\hat{\sigma}_i \leftarrow\!\!\$\, \mathsf{Sign}_\Sigma(\mathsf{wsk}, (\mathsf{pk}'_i, l_i))$ where $l_i = \mathsf{Pub}(v_i)$.
3. Output $\{\mathsf{msk}_i\}_{v_i \in V}$ and $\mathsf{mpk} = (G, \mathsf{Pub}, \{\hat{\sigma}_i\}_{v_i \in V}, \mathsf{wpk})$.

$\mathsf{DPub}(\mathsf{mpk}, v_j)$**:** On input the master public key $\mathsf{mpk} = (G, \mathsf{Pub}, \{\hat{\sigma}_i\}_{v_i \in V}, \mathsf{wpk})$ and a node $v_j \in V$, the algorithm returns $\mathsf{pk}_j = (\mathsf{wpk}, l_j)$ where $l_j = \mathsf{Pub}(v_j)$.

$\mathsf{DPriv}(\mathsf{mpk}, \mathsf{msk}_i, v_i, v_j)$**:** On input the master public key $\mathsf{mpk} = (G, \mathsf{Pub}, \{\hat{\sigma}_i\}_{v_i \in V}, \mathsf{wpk})$, the master secret key $\mathsf{msk}_i = S_i$, two nodes $v_i \in V, v_j \in \mathit{Desc}(v_i)$, the algorithm runs $x_j = \mathsf{Derive}_\Gamma(G, \mathsf{Pub}, v_i, v_j, S_i)$ and $(\mathsf{sk}'_j, \mathsf{pk}'_j) = \mathsf{KGen}_\Sigma(1^\lambda; x_j)$. Finally, it returns $\mathsf{sk}_j = (\mathsf{sk}'_j, \mathsf{pk}'_j, \hat{\sigma}_j)$.

$\mathsf{Sign}(\mathsf{sk}_i, m)$**:** On input a signing key $\mathsf{sk}_i = (\mathsf{sk}'_i, \mathsf{pk}'_i, \hat{\sigma}_i)$ and a message m, the algorithms returns $\sigma = (\mathsf{pk}'_i, \sigma', \hat{\sigma}_i)$ where $\sigma' \leftarrow\!\!\$\, \mathsf{Sign}_\Sigma(\mathsf{sk}'_i, m)$.

$\mathsf{Vrfy}(\mathsf{pk}_i, m, \sigma)$**:** On input a public key $\mathsf{pk}_i = (\mathsf{wpk}, l_i)$, a message m, and a signature $\sigma = (\mathsf{pk}'_i, \sigma', \hat{\sigma}_i)$, the algorithms returns 1 if $\mathsf{Vrfy}_\Sigma(\mathsf{wpk}, (\mathsf{pk}'_i, l_i), \hat{\sigma}_i) = 1$ and $\mathsf{Vrfy}_\Sigma(\mathsf{pk}'_i, m, \sigma') = 1$; otherwise it returns 0.

The correctness of the scheme comes directly from the correctness of the underlying primitives. As for security, we establish the following result whose proof appears in Appendix A.2.

Theorem 5.1. *Let* $\Gamma = (\mathsf{Set}_\Gamma, \mathsf{Derive}_\Gamma)$ *and* $\Sigma = (\mathsf{KGen}_\Sigma, \mathsf{Sign}_\Sigma, \mathsf{Vrfy}_\Sigma)$ *be respectively a deterministic hierarchical key assignment and a signature scheme. If* Γ *is key indistinguishable (Definition 4.2) and* Σ *is existentially unforgeable (Definition 3.5), then the HDW* Π *from Construction 2 is hierarchically existentially unforgeable (Definition 5.2).*

In the context of the three-levels security models defined in Section 2, the public derivation of Arcula does not assume any level of trust and, as such, belongs to the **Untrusted Environment**. An Arcula public key is, indeed, the concatenation of two public values: The cold storage key wpk and the identifier l_i of node v_i. Redeeming the coins destined to an Arcula node v_i assumes instead the **Warm Environment**. The certificate $\hat{\sigma}_i$ that authorizes it to spend the funds through the public key pk'_i is a public parameter of the wallet, but we require the node's private signing key sk'_i to sign a new transaction. Compromising the secrets of node v_i in this environment leads to compromising all its descendants, but none of the other nodes. Finally, the secret key wsk must be safely stored within the context of the **Cold Storage**. We leverage this key wsk only during the setup of the wallet, *i.e.* when we sign and release the authorization certificate $\hat{\sigma}_i$ associated to a identifier l_i of a node v_i (*e.g.*, when we run the Set algorithm). Such secret key is critical to the security of the wallet, as an attacker can use it to forge a certificate that associates any pair of keys to any target node and spend the coins of the entire wallet.

To summarize, Arcula defines a hierarchical deterministic wallet that benefits from the following properties:

1. Is secure against key recovery attacks (Property 2.1).
2. Generates every cryptographic key from an initial seed (Property 2.2).
3. Enables public-key derivation so that nodes can derive the public keys of their children without accessing their own private keys (see Property 2.3).
4. Enables secret-key derivation so that nodes can sign a transaction on behalf of their descendants.
5. Does not rely on any particular digital signature scheme.
6. The DHKA at the core of Arcula is secure under key indistinguishability and handles any directed acyclic graph encoding a partially ordered hierarchy. In addition, it allows to dynamically modify the hierarchy (*i.e.*, by adding or removing nodes, as we detail in Section 7) and to control the assignment according to some temporal constraints (Section 8).

6 Arcula in the real world

With Arcula, we aim at achieving two goals at the same time. First, to design a hierarchical deterministic wallet that is secure against privilege escalation, and that is suitable for any modern blockchain-based crypto-system. Also, to create an HDW that can be used in practice and that supports the most widely used crypto-systems of today. To this end, we constrain our design with as few cryptographic assumptions as possible. Arcula works with any existentially unforgeable signature scheme and only requires the verification of a signature on an arbitrary message (*i.e.*, the certificate provided by the cold storage that authorizes the signing key of a user). This design makes it immediately compatible with the Ethereum blockchain, that implements a Turing-complete language, and with all the forks based on Bitcoin that allow the signature verification of arbitrary messages (*e.g.*, Bitcoin Cash). The original Bitcoin implementation,

instead, does not allow such operation (in fact, it goes as far as disabling the operations of string concatenation and integer multiplication that, originally, it allowed). We first focus on how to spend and receive funds, out of the box, on the Bitcoin Cash blockchain; then, we discuss the modifications that would make it compatible with the original Bitcoin protocol; finally, we compare the properties that Arcula achieves against those of BIP32.

6.1 Technical Implementation

Our open-source implementation of Arcula is available online[3]. We define the hash function $\mathsf{H}(x)$ to be SHA3-256(x) and we implement the pseudorandom function $\mathsf{F}_k(x) = \mathsf{H}(k\|x)$. The symmetric encryption scheme $\mathcal{E}$ required by our Deterministic Hierarchical Key Assignment scheme Γ is the authenticated encryption scheme AES256 with Galois/Counter Mode (GCM). We generate a hierarchal deterministic wallet based on the tree structure defined in BIP43 and BIP44 [15, 16], where the keys to different crypto-coins correspond to different subtrees, and each branch of the subtrees is a chain associated to a single account that contains multiple receiving addresses. We obtain an initial seed S_{-1} of 512 bits by following the specification of BIP39 [17] that generates a seed from a random mnemonic sequence. We generate the wallet that we use in our tests by fixing the randomness of the mnemonic generation process to the result of the operation H(`correct horse battery staple`).

6.2 Bitcoin Cash

This section shows how to send and receive funds while using Arcula in Bitcoin Cash or in any Bitcoin-based blockchain that implements the verification of signatures on arbitrary messages.

Bitcoin Scripts A Bitcoin transaction is a cryptographically signed statement that transfers some coins from a sender to a receiver. The sender of the coins signs the transaction through her secret key to spend, in turn, the coins destined to the corresponding public key. Every transaction specifies a locking and an unlocking script. These scripts respectively state the necessary conditions to spend, in a future transaction, the coins being transferred (*i.e.*, their locking condition) and provide the information required to redeem them (*i.e.*, to unlock them as a result of a past transaction). Both scripts are written through a stack-based language that allows simple mathematical operations, stack manipulations, and enables simple cryptographic primitives (*i.e.* computing the result of a hash function and verifying a signature).

A typical Bitcoin locking script specifies the address of the receiver (usually through the hash of its public key) and requires him to provide a valid signature in order to redeem the coins being transferred. More in details, the locking and unlocking scripts of a standard Bitcoin transaction are defined as follows. Uppercase monospace words indicate operations of the Bitcoin scripting language, while angular brackets enclose variable inputs.

Locking: `OP_DUP OP_HASH160` <$\mathsf{H}(\mathsf{pk})$> `OP_EQUALVERIFY OP_CHECKSIG`

Unlocking: <σ> <pk>

Together, these scripts ensure that the public key pk provided in the unlocking script is the pre-image of the hash $\mathsf{H}(\mathsf{pk})$ (the Bitcoin address) contained in the locking script; then, verify the validity of the transaction signature σ under the public key pk.

In Arcula, instead, we identify the nodes of our wallet v_i according to the cold storage public key wpk and to their public label l_i. For this reason, an Arcula address is simply the

[3] Available at `https://github.com/aldur/Arcula`.

Address type	Locking Script	Unlocking Script	Total
Standard	24	106	130
Arcula	43	179	222

Table 1: The script bytes sizes of a transaction to a standard Bitcoin address and to an Arcula address.

concatenation of the byte representations of these values, that we encode in the locking script. The unlocking script, on the other hand, contains the certificate $\hat{\sigma}_i = \mathsf{Sign}_\Sigma(\mathsf{wsk}, (\mathsf{pk}'_i, l_i))$, signed by the cold storage secret key wsk and associating the signing public key pk'_i to the node v_i with label l_i, and a signature σ of the transaction under the public signing key pk'_i. More in details, with Arcula the locking and the unlocking scripts respectively become:

Locking: `OP_DUP OP_TOALTSTACK` <l_i> `OP_CAT` <wpk> `OP_CHECKDATASIGVERIFY OP_FROMALTSTACK OP_CHECKSIG`

Unlocking: <σ> <$\hat{\sigma}_i$> <pk'_i>

The two scripts: 1) Verify that the certificate $\hat{\sigma}_i$ is a valid signature of the message (pk'_i, l_i) under the wallet public key wpk; 2) verify the validity of the transaction signature σ under the signing public key pk'_i. In particular, the locking script checks the validity of the certificate through the operation `OP_CHECKDATASIGVERIFY`, which allows the stack-based scripting language to validate a signature of an arbitrary message (the concatenation of pk'_i and l_i obtained through the operation `OP_CAT`). The scripting language of the original Bitcoin does not implement such operation yet. Nonetheless, a significant portion of the Bitcoin community believes that its adoption would provide substantial benefits to the entire system, *e.g.*, by enabling third-parties to store and verify independent messages on the blockchain. For this reason, many Bitcoin forks (Bitcoin Cash, Bitcoin Ultimate, and Blockstream to name a few), that aim at modernizing the protocol and at improving the stack-based language used in scripts, now implement this operation.

In our experiments, we focus, as an example, on Bitcoin Cash—the sixth crypto-currency by market capitalization at the time of writing—and we evaluate Arcula on its test blockchain. We first create a transaction[4] that locks 0.5 BCH (the Bitcoin Cash crypto-coin) to a node of our wallet of Section 6.1, identified through the cold storage public key wpk (also in the locking script) and the integer label 3. Next, we redeem the coins through a second transaction that provides the transaction signature σ of the secret key sk_i, an appropriate certificate $\hat{\sigma}_i$ signed by the cold storage key wsk, and the public signing key pk'_i. We create both the signature and the certificate through the ECDSA signatures scheme on the `secp256k1` elliptic curve used in Bitcoin, and we encode the integer label of the node v_i with 4 bytes.

Transaction Costs To study the costs of Bitcoin transactions to an Arcula address, we analyze the amount of storage that they require on the blockchain. Every Bitcoin transaction devolves a small amount of fees to the system to incentive its inclusion in the next block of the chain. Fees are usually measured in coins per byte and, for this reason, the size of a transaction on the Bitcoin wire protocol is directly related to the amount of fees that it should pay to be included in the blockchain. In particular, the length of the locking and unlocking scripts influences directly the final transaction cost. Table 1 compares the sizes, in bytes, of the locking

[4]The transcripts of the transactions are available, respectively, at `https://bit.ly/2UI62tt` and `https://bit.ly/2UoQNGI`.

and unlocking scripts of standard Bitcoin transactions and to an address of our wallet. Every operation of the stack-based scripting language is encoded with a single byte; a standard Bitcoin address is the result of a hash function that outputs 20 bytes; the ECDSA signature and the public key in the unlocking script require, respectively, 73 and 33 bytes. By summing these values up, we find that the locking script of a transaction to a standard Bitcoin address is 24 bytes long (4 script operations plus the receiver address) while the unlocking scripts take 106 bytes (the ECDSA signature and its associated public key). In Arcula, on the other hand, the locking script encodes 6 operations, the identifier of a node (that we encode with 4 bytes), and the cold storage public key (33 bytes, as opposed to its 20 bytes hash), for a total of 43 bytes. The unlocking script, instead, contains two ECDSA signatures (one for the transaction and one for the certificate) and the signing public key; as a result, it is 179 bytes long. Overall, the size of the locking and unlocking scripts for a transaction to an Arcula address is 222 bytes, 70% longer than the standard address counterparts.

In particular, the Bitcoin users aim at minimizing the size of the locking script, as its associated fees will be paid by the sender of the transaction, *e.g.*, the customer of an online service, and the service providers usually aim at minimizing these costs. Bitcoin solves this issue through the pay to script hash mechanism, proposed in BIP16 [1], that reduces the size of any locking script to a constant at the cost of longer unlocking scripts. The intuition is that instead of specifying the full locking script, the users can constrain the coins of a transaction by locking them to the hash of the original script; then, in the unlocking script, they can provide both the pre-image of the hash, *i.e.*, the full locking script, and its required inputs. This approach brings several advantages. First, any locking script can be expressed with a constant byte size that results in a fixed cost for the sender. Second, it hides the details of the locking script until the users reveal the pre-image of the hash in an unlocking script, *i.e.* when they redeem the coins sent by the transaction. Finally, the Bitcoin protocol proposes a way to encode the pay to script hash locking scripts into standard Bitcoin addresses, so that exchanging transactions of this kind is entirely transparent to the software used by the sender. By using the pay to script hash mechanism, any user can send a transaction to an Arcula address through her favorite Bitcoin wallet, in a transparent way that does not require any specific software modification to it. More in details, an Arcula pay to script hash transaction is defined as follows, where the script that we input to the hash function is the locking script of a transaction to an Arcula address that we have seen before:

Script: `OP_DUP OP_TOALTSTACK` <l_i> `OP_CAT` <wpk> `OP_CHECKDATASIGVERIFY OP_FROMALTSTACK OP_CHECKSIG`

Locking: `OP_HASH160` <H(Script)> `OP_EQUAL`

Unlocking: <σ> <$\hat{\sigma}_i$> <pk'_i> <Script>

The pay to script hash mechanism reduces to 22 bytes (2 operations and a 20 bytes hash) the size of the locking script and, equivalently, the amount of fees that users have to spend to send funds to an Arcula address. The size of the unlocking script, on the other hand, affects the fees that the users of Arcula need to pay when spending their coins. In particular, when using pay to script hash, this amount of fees is slightly larger than the one required for a traditional Bitcoin transaction. In many cases, however, the benefits that arise with Arcula justify the increase in the transaction cost. We take as an example an online marketplace that handles incoming and outgoing payments through the blockchain. First, the determinism of Arcula guarantees that even on catastrophic hardware failure the service provider will be able to recover the entire wallet and all its signing keys. Besides, Arcula's public key derivation allows her to dynamically derive new addresses (*e.g.*, one for each product of her catalog) in an entirely untrusted environment

(*e.g.*, an online web-server) while keeping all her signing keys at rest in trusted storage. As a result, the provider obtains the flexibility of handling incoming payments on dynamic addresses and minimizes the risk of losing the coins associated with them. When compared with the financial costs associated with this risk, the additional fees required by the Arcula transactions are only negligible. The public-key derivation also brings other significant benefits. Many financial regulations require, indeed, companies to be accountable for all the payments that they receive. With Arcula, an auditor can reach this goal by merely inspecting the blockchain while looking for any address that contains the cold storage public key wpk that identifies the company. Finally, many companies leverage m-of-n signatures, where redeeming a transaction requires m valid signatures among n authorized public keys. Their goal is to enforce the internal structure of the company (*e.g.*, so that either managers or employees can sign transactions) or to divide the responsibility of spending coins evenly. The unlocking scripts of m-of-n transactions have considerable size: They contain m signatures and n public keys. By leveraging Arcula and enforcing an appropriate hierarchy that reflects their internal structure, these companies could reduce the size of the unlocking scripts to only two signatures (the transaction signature and the certificate) and two public keys (the cold storage and signing public keys).

Optimizations and compatibility with Bitcoin The current implementation of Arcula does not require any modification to the underlying protocols and blockchains. Nevertheless, we also propose a set of optimizations that, through minimal modifications to these protocols, reduce both the cost of transactions to Arcula addresses and the amount of storage required on the blockchain. We begin by noting that any authorization certificate $\hat{\sigma}_i$ can be used more than once by the corresponding signing key pk'_i. For this reason, the first optimization that we propose is to *cache* the certificate $\hat{\sigma}_i$ as soon as it appears for the first time in an unlocking script. Then, any subsequent transaction signed by pk'_i could specify a pointer to the certificate (*e.g.*, with a shorter hash) instead of the certificate itself and, in turn, reduce the size of the unlocking script. As an example, by pointing to the certificate with a 20 bytes hash, we would reduce the size of the Arcula locking and unlocking scripts to be roughly 20 bytes longer than their traditional counterparts. Implementing this optimization requires a new operation in the scripting language to retrieve the certificate from the cache and to verify its validity. On the other hand, if we allow for more complex modifications, we can change the signature scheme of the underlying protocols to reduce these space requirements to their optimal value further—a single signature per transaction. Arcula can be implemented with a single signature by leveraging a sanitizable signature scheme [4], *i.e.* a scheme where an authorized party can modify a fraction of the message signed without interacting with the original signer. The intuition is to provide every user v_i with a signature of the wallet secret key wsk that authorizes their signing public key pk'_i and that also includes an additional modifiable portion (a blank transaction). To spend their coins, the users leverage their secret key sk'_i to modify the sanitizable portion of the message and to replace the blank transaction with the details of the new one that they intend to sign. In their work, Ateniese *et al.* [4] show how to construct a sanitizable signature scheme by combining any signature scheme with a chameleon hash function. Their construct would enable Arcula to be also used with the traditional Bitcoin blockchain by implementing the sanitizable signatures on top of the ECDSA signature scheme that it already uses and without changing the expressiveness of the Bitcoin scripting language (*i.e.*, without requiring the verification of a signature on an arbitrary message).

6.3 Comparison with BIP32

We compare Arcula with BIP32 by evaluating it in details at the three different security and privacy scenarios that we present in Section 2.1. In the **Public Chain Code** setting, Arcula achieves the same properties of BIP32. Indeed, when the chain code c of BIP32 is a public parameter of the wallet, then a passive attacker can correlate together all the public keys derived from the seed. Similarly, a passive attacker can inspect the locking scripts of any transaction to an Arcula address and group together those of them that include the same cold storage public key wpk. That said, Arcula shows a significant difference with BIP32: Even after compromising an arbitrary number of signing keys, an attacker can not escalate his privileges and spend the coins of a node higher in the hierarchy (*i.e.*, Arcula does not suffer from the key recovery vulnerability that we discuss in Section 2.1, Equation (1)).

On the other hand, in the **Hardened Derivation** scenario, BIP32 trades the public derivation property to solve the privilege escalation vulnerability. In this setting, Arcula achieves the same properties as well. When users are not interested in public derivation and aim at receiving payments on uncorrelated pseudonyms, they can identify nodes through their public signing keys pk'_i and ignore the wallet public keys wpk. To do so, they address standard Bitcoin transactions (with standard costs) to their public signing keys pk'_i and then sign new transactions redeeming the coins through the corresponding private keys sk'_i. We generate the signing keys by leveraging our Deterministic Hierarchical Key Assignment (DHKA) scheme, which is secure under key indistinguishability. In contrast, the security of BIP32 has never been formally proved.

Finally, we consider the **Private Chain Code** scenario. Here, BIP32 leverages a private chain code c in the key derivation process so that the public key of a child node looks uncorrelated from the public key of its parent. This result holds against any passive attacker as long as the chain code is private and running the public-key derivation requires the knowledge of the private chain code. With Arcula we achieve the same result without additional requirements. The intuition is to use the private chain code c to perturb the cold storage secret and public keys $(\mathsf{wsk}, \mathsf{wpk})$ so that they look uncorrelated from the original keys. As a result, we obtain a perturbed pair of cold storage keys for each node in our wallet, labeled $(\mathsf{wsk}_i, \mathsf{wpk}_i)$, that we use to sign the certificate $\hat{\sigma}_i$ that associates the public signing key pk'_i to v_i. Then, we address any Bitcoin payment of the node v_i to the i-th perturbation wpk_i of the cold storage public key wpk, that is uncorrelated from any other key of the wallet. More in details, let g be a generator of the `secp256k1` elliptic curve used in Bitcoin's ECDSA signature scheme. Then, we know that the public cold storage key $\mathsf{wpk} = g^{\mathsf{wsk}}$. Let c be the secret chain code and let F be a pseudorandom function. We create the i-th perturbation wpk_i of the cold storage key wpk as follows:

$$\mathsf{wpk}_i = g^{\mathsf{wsk}+\mathsf{F}_c(l_i)} = g^{\mathsf{wsk}} \cdot g^{\mathsf{F}_c(l_i)} = \mathsf{wpk} \cdot g^{\mathsf{F}_c(l_i)},$$

where l_i is the label of node v_i. Consequently, we modify the Item 2c of Construction 2 to sign the certificates $\hat{\sigma}_i \leftarrow\!\!\$\ \mathsf{Sign}_\Sigma(\mathsf{wsk}_i, \mathsf{pk}'_i)$ with the perturbed public key $\mathsf{wsk}_i = \mathsf{wsk} + \mathsf{F}_c(l_i)$. Note that we remove the label l_i from the certificate since now every pair of perturbed keys is uniquely associated to precisely one pair of signing keys and the perturbation already takes into explicit consider the label l_i of node v_i. Finally, we replace the cold storage public key wpk in the locking script with the i-th perturbed key wpk_i, that verifies the certificate $\hat{\sigma}_i$, as follows:

Locking: `OP_DUP OP_TOALTSTACK` <wpk_i> `OP_CHECKDATASIGVERIFY OP_FROMALTSTACK OP_CHECKSIG`

Unlocking: <σ> <$\hat{\sigma}_i$> <pk'_i>

As a result, all the Arcula addresses of the same wallet look uncorrelated when they appear in the locking script of a transaction. That said, we point out that, essentially, the perturbed

private keys wsk_i are equivalent to the original private key wsk: An attacker that compromises a perturbed key can invert the perturbation, recover the original cold storage key, and compromise the entire wallet by forging new certificates for key pairs that he controls. For this reason, the perturbed keys shall be kept in the same trusted environment of the cold storage private key.

To conclude, we briefly discuss how to derive deterministically the chain code c. We propose to generate a different chain code c_i for each node v_i of the wallet by running our DHKA a second time: An attacker that compromises a node in the hierarchy can only uncover the public identifiers of the nodes in its subtree, but would not gain any knowledge about the others in the hierarchy.

This last scenario depicts how to use Arcula in the real-world setting that BIP32 proposes, *i.e.* where the addresses of the wallet look uncorrelated from one another (*i.e.*, a user can have multiple pseudonyms) and public derivation is enabled (and requires the chain code c). As before, Arcula outperforms BIP32: It achieves its properties of private and public key derivation, it improves the privacy of the users by allowing them to use pseudonyms, but it does not suffer from its key recovery vulnerability (Section 2.1, Equation (1)).

7 Handling Dynamic Changes to a Deterministic Key Assignment Access Hierarchy

This section details how to handle dynamic changes to the access hierarchy (*e.g.*, insertion of a node or deletion of an edge) of our deterministic key assignment scheme of Section 4 and, in turn, within Arcula, our hierarchical deterministic wallet of Section 5.

Handling dynamic changes to the access hierarchy of the DHKA requires us to consider two problems. First, how to correctly enforce the hierarchy after the modification (*e.g.*, preventing a node from accessing a subtree after an edge to that subtree is removed); second, how to deal with modifications to the structure of G that change the path from the root to any node v_i along which we deterministically derive the secret values S_i (*e.g.*, removing the parent of a node). We solve these problems through the following strategies. First, we modify the graph G by adding an explicit root node to it, v_R, such that there exists an edge between v_R and any *root* node of G (*i.e.*, any minimal node in a topological ordering of G). More in details, we define $G' = (V \cup \{v_R\}, E \cup \{(v_R, v_i) \mid v_i \text{ has no predecessors in } G\})$. It is easy to prove that both G and G' define equivalent access hierarchies.

Next, we associate an additional identifier, that we call *version*, to each node by including it in its label. Let $\mathsf{Ver} : V \to \mathbb{N}$ be a public mapping associating an integer $w_i \in \mathbb{N}$ to any node $v_i \in V$. Every node v_i initially starts from version $w_i = 0$, and we modify Items 1 and 2 of Construction 1 to account for it when deriving the node label l_i:

$$l_i = v_i \| w_i$$

Every time we modify the graph G' in such a way that it would require updating the secret of a node v_i, we do so by updating its version w_i, deterministically computing its new label l_i, and, in turn, its new secret S_i.

In the remainder of this section, we leverage the version associated to each node to perform a *rekey* procedure, defined as follows for every node v_h and for every node v_p such that v_p is the parent of v_h in G'.

1. Increase the version w_h of the node v_h to a new value $w_h{}'$ and update the Ver data structure. Then, compute a new label $l'_h = v_h \| w'_h$ and update the corresponding entry in Pub. Finally, compute a new secret $S'_h = \mathsf{F}_{S_p}(11 \| l'_h)$, a new pair of secret and intermediate keys $x'_h = \mathsf{F}_{S'_h}(01 \| l'_h)$ and $t'_h = \mathsf{F}_{S'_h}(00 \| l'_h)$, and update the Sec mapping.

2. For each incoming edge (v_k, v_h) of v_h, update the public information $y'_{kh} \leftarrow\!\!\$\, \mathsf{Enc}_{r_{kh}}(t'_h \| x'_h)$ stored on the edge to reflect the updated values t'_h and x'_h.

Finally, we deal with the dynamic modifications of the graph:

Deletion of an edge: Let $(v_i, v_j) \in E$ be the edge that is to be removed from G'. Our goal is twofold: First, to prevent v_i from accessing the cryptographic keys of v_j. Second, to make sure that if the deletion of the edge changes the derivation path from the root v_R of the hierarchy to v_j, then the deterministic generation of the secret S_j changes accordingly. We begin by tackling this last problem. Let v_p be the parent node of v_j in G'. If $v_i = v_p$ and if there is no other edge $(v_{p'}, v_j) \in E$ (*i.e.*, there does not exist another predecessor of v_j that is a candidate to become its new parent), then the deletion of the edge $(v_i, v_j) \in E$ results in disconnecting of v_j from the access hierarchy. In that case, we add a connecting edge (v_R, v_j) to G' that creates a single-hop path from the root to v_j and allows the deterministic key derivation of its secret S_j. We note that the addition of this edge does not modify the access hierarchy, *i.e.* it does not allow v_j to derive the secrets of any node that was not previously between its descendants $Desc(v_j)$.

Next, we prevent v_i from accessing the cryptographic keys of v_j by performing the *rekey* procedure for each node $v_h \in Desc(v_j)$ (this includes v_j as well).

Deletion of a node: The deletion of any node v_i corresponds to first removing all the incoming and outgoing edges of v_i through the procedure specified above. Then, to removing the public and secret information associated with v_i from the Pub, Sec, and Ver data structures.

Insertion of an edge: Let $(v_i, v_j) \in E$ be the edge to be included into G'. We consider two cases:

- After the addition of the edge v_i is the parent of v_j in G'. As before, we perform the *rekey* procedure for each node $v_h \in Desc(v_j)$ to update their secret values and to allow the deterministic derivation.
- Otherwise, compute $r_{ij} = \mathsf{F}_{t_i}(10 \| l_j)$, $y_{ij} \leftarrow\!\!\$\, \mathsf{Enc}_{r_{ij}}(t_j \| x_j)$, and augment Pub to contain the mapping $(v_i, v_j) \mapsto y_{ij}$.

Insertion of a new node: Let v_i be the node to insert, together with a set of new edges in and out of it. Let v_j be the parent of v_i. We begin by computing a deterministic public label $l_i = v_i \| w_i$ (where $w_i = 0$) and a deterministic secret value $S_i = \mathsf{F}_{S_j}(11 \| l_i)$; then, we compute $k_i = \mathsf{F}_{S_i}(01 \| l_i)$ and we augment Pub with the mapping $v_i \mapsto l_i$, Sec with the mapping $v_i \mapsto (S_i, x_i)$, and Ver with the mapping $v_i \mapsto 0$. Finally, we proceed to insert the edges one by one using the edge insertion procedure specified above.

Key Replacement: To replace the cryptographic key x_i associated to any node v_i, we perform the *rekey* procedure for each node $v_h \in Desc(v_i)$.

This approach allows our deterministic key assignment scheme to handle dynamic changes to its access hierarchy and requires the manager of the key assignment (*e.g.*, a crypto-currencies exchange) to keep track of the version of the nodes stored within the Ver mapping in addition to the structure of the graph G. Because of the determinism of our scheme every change to a node v_j also propagates to all its descendants. As an example, the replacement of its secret key x_j requires incrementing its version w_j in the label l_j to compute a new secret value and a new cryptographic key. In turn, this causes the secret information and the cryptographic keys of all its descendants v_i to change as well (because of the deterministic derivation of the secret values $S_i = \mathsf{F}_{S_j}(11 \| l_i)$). The cost of such an update depends on the particular application and

the structure of the access hierarchy. If we use the DHKA to handle the keys associated to traditional Bitcoin transactions, for example, updating the cryptographic key of a node requires sending its funds to a new address and involves the payment of a transaction fee. Most of the times, however, we are particularly interested in appending new leaves to the access hierarchy (*e.g.*, to create a new node for an incoming payment). This operation is a particular case of the insertion of a new node with a single incoming edge. It never modifies any derivation path and, as a consequence, it does not perform the *rekey* procedure, it does not change any cryptographic key, and it does not require transactions on the blockchain. Finally, when we use Arcula to enable the public derivation of addresses, we identify the nodes of the wallet through the cold storage public key wpk and a label. Both the addition of a new node and the update of the label l_j of an existing node v_j result in a new Arcula address (*i.e.*, a locking script that contains a new label). Spending the funds destined to the new address requires a new certificate, signed by the cold storage public key wpk, that associates the public key pk_j to the new (or updated) label l_j.

8 Time-Bound Deterministic Hierarchical Key Assignment

A hierarchical key assignment scheme aims at assigning a cryptographic key to every user of an access hierarchy so that users with higher privileges can autonomously derive the keys of the others within their subtrees, *i.e.*, with lower privileges in the hierarchy. Many uses cases require constraining these assignments according to some time restrictions. For example, a service provider aims to provide a user with her cryptographic keys only as long as she pays for her subscription to the service. To achieve this goal, it can leverage a key assignment scheme that takes time into account, and that enables the users to deriver their cryptographic keys during a given period only (*e.g.*, one month). This section details how we incorporate these temporal capabilities into the deterministic hierarchical key assignment scheme of Section 4 and within Arcula, our design hierarchical deterministic wallet (Section 5).

In the last few years, many researchers focused on how to incorporate temporal capabilities into HKA schemes [5, 2, 9]. The solutions proposed first modify the hierarchy of the assignment to consider, at the same time, both the access privileges and the temporal constraints. Then, assign a set of secrets to the nodes of the augmented hierarchy so that the users can perform the key derivation according to the time constraints.

We add these constraints to our DHKA by relying on the work of De Santis *et al.* [9] that shows how to design a time-bound key-indistinguishable HKA scheme from any provably secure HKA scheme (and, in particular, from our DHKA). Let $G = (V, E)$ be an access hierarchy and let $\mathcal{T} = \{t_1, t_2, \ldots, t_n\}$ be a sequence of distinct time periods. Each user v_i belongs to a node of the hierarchy for a non-empty contiguous subsequence $\mathcal{T}_i = \{t_j, \ldots, t_k\} \subseteq \mathcal{T}$ of time periods.[5] Let $\mathcal{P} = \{\mathcal{T}_i\}_{v_i \in V}$ be the set of time subsequences $\mathcal{T}_i$ when every user $v_i \in V$ belongs to the hierarchy. The authors start from the observation that the contiguous subsequences $\mathcal{T}_i \in \mathcal{P}$ implicitly define a partially ordered hierarchy, where $\mathcal{T}_i < \mathcal{T}_j \iff \forall t_k \in \mathcal{T}_i \implies t_k \in \mathcal{T}_j$, *i.e.* iff $\mathcal{T}_i$ is included in $\mathcal{T}_j$. They call this relation the *interval hierarchy*, and they use its minimal representation, where every node except the leaves has precisely two edges, to augment the original access hierarchy encoded by the graph G. As a result, they build a new graph, $G_{\mathcal{T}} = (V_{\mathcal{T}}, E_{\mathcal{T}})$, that enforces both the access and the interval partially ordered hierarchies. $G_{\mathcal{T}}$ contains a copy of the interval hierarchy for each node in G. A user v_i derives the cryptographic key assigned to its descendant v_j for the period $t_k \in \mathcal{T}_j$ by following the path in the augmented graph $G_{\mathcal{T}}$ along the copy of the interval hierarchy related to v_j and then through the original access hierarchy encoded by G. The instantiation of the (D)HKA scheme on the graph $G_{\mathcal{T}}$ results in a (deterministic)

[5] In [9] the subsequence of time periods of a node $v_i \in V$ is denoted by λ_i.

time-bound hierarchical key assignment scheme.

By construction, the number of nodes and edges in $G_{\mathcal{T}}$ grows quadratically in the size of $\mathcal{T}$ and in the dimension of G. In turn, the amount of public information required by a generic HKA scheme on $G_{\mathcal{T}}$ grows comparably. As we have seen in Section 4.2, however, the determinism of our DHKA scheme allows us to reduce the amount of public information required significantly: The nodes of the access hierarchy can derive the secret information of their descendants by leveraging their own secrets and only rely on the public information when a node has two or more predecessors. In the same way, when we augment the access hierarchy encoded by G to account for the interval hierarchy into $G_{\mathcal{T}}$, the determinism of the scheme allows us to reduce the amount of public information required. The augmented hierarchy $G_{\mathcal{T}}$, indeed, stores a copy of the minimal interval hierarchy for every node of G. Every node of the minimal interval hierarchy only has a single predecessor and, as a result, does not require any public information associated to its edges. For this reason, when we leverage our design of DHKA to incorporate the temporal capabilities into an HKA scheme, the size of the public information required grows only linearly with the dimension of the access hierarchy G and, in particular, is independent of the cardinality of $\mathcal{T}$.

To conclude, we show how to incorporate these temporal capabilities into Arcula, our design of HDW based on DHKA and digital signatures. Our construction provides the users of the access hierarchy with a certificate and a signing key. The certificate, signed by the cold storage public key, authorizes the signing key to spend the coins addressed to their identities. When we add the temporal capabilities to the DHKA we assign a different signing key to each user v_i for each time period $t_j \in \mathcal{T}_i$; then, we provide her with a certificate $\sigma_{i,j}$ for each key. We prevent the users from signing new transactions through an outdated key by adding an expiration date to these certificates so that they are only valid until the end of the period t_j. As a result, every user v_i will require an updated certificate, signed by cold the storage key, after each time period passes. The stack-based scripting language of Bitcoin Cash does not allow yet to check for the expiration date of a certificate. For this reason, our design of time-bound Arcula requires, at the time of writing, a more powerful scripting language, *e.g.* an Ethereum smart contract.

9 Related Work

Hierarchical Deterministic Wallets in Bitcoin and other blockchain-based crypto-systems only attracted the interest of a few researchers in the past years. Courtois *et al.* [7] are the first to investigate on HDW and on the security issues that arise with the design of BIP32. They show that the public derivation property comes at a considerable security cost: By compromising a single private key at any level of the hierarchy, an attacker can recover the secrets of the whole access structure and spend all the assets held within the wallet.

Gutoski and Stebila [14] move a first step in solving the security issues of BIP32 and propose a new construct for HDW that tolerates key leakage. More in details, each node of their wallets holds n master private keys and as many corresponding public keys. The secret and public keys of each child are derived by computing a linear combination of the master private keys. As a result, the authors design a wallet that allows public-key derivation and that can tolerate the leakage of at most n private keys. Nonetheless, this comes at an increased storage cost, since each node must now explicitly store n public keys.

Goldfeder *et al.* [13] and subsequently Gennaro *et al.* [12] design a threshold-based ECDSA signature scheme useful for securing Bitcoin wallets. In [13], they leverage the threshold ECDSA signature to implement a two-factor wallet, allowing a user to share its secret key among two devices. On the other hand, [12] propose a non-hierarchical deterministic wallet where the secret key is shared among n parties and at least t of them are required to sign a transaction. Their

implementation allows the participants to derive new public keys while keeping the corresponding secret key shared among the n parties. Dikshit and Singh [10] extend the threshold-based ECDSA signatures to allow the participants of the protocol to have different weights (*e.g.*, so that the executives of a company have more weight than their employees).

Fan *et al.* develop an HDW that is secure against privilege escalation and that allows public-key derivation [11]. Their work leverages Schnorr signatures and trapdoor hash functions to enable the users to sign new transactions without accessing their private keys. On a high level, their wallet works as follows: The root node of the hierarchy creates the Schnorr signature of a generic message through the trapdoor hash function. Then, the users on the lower levels of the hierarchy leverage their private trapdoor hash function key to find a hash collision and to reuse the signature created by the root to validate a new message of their choice. That said, we note that the wallet proposed by Fan *et al.* suffers from the following issue. A generic node of the hierarchy is not allowed to sign on behalf of the users of its subtree (*i.e.*, the private derivation property does not hold) unless the root node explicitly authorizes it—and to do so, it needs to reveal its master private trapdoor key. This means that any node authorized by the root is able not only to sign new transactions on behalf of the users in its subtree but also of the users higher in the hierarchy (*i.e.*, to escalate its privileges).

10 Conclusions

In this work we presented Arcula, a new hierarchical deterministic wallet (HDW) that outperforms the state of the art BIP32, that is built on provably secure cryptographic primitives, and that, in particular, is secure against privilege escalation. To do so, we developed a key indistinguishable deterministic hierarchical key assignment (DHKA) scheme that we use to deterministically generate the set of cryptographic keys at the core of our wallet. As a result, an attacker that compromises an arbitrary number of users in the hierarchy can not escalate his privileges and compromise any user higher in the hierarchy. In addition, our wallet allows to derive a new address for receiving payments in an entirely untrusted environment, to recover every cryptographic key from an initial seed provided by the user, and also to spend coins on behalf of users lower in the hierarchy. Our design of Arcula considers the current and the future requirements of modern blockchains. In particular, Arcula is independent of the underlying signature scheme, and it works, out of the box, with any crypto-system that allows the verification of signatures on an arbitrary message (*e.g.*, Bitcoin Cash or Ethereum). For these reasons, we hope that the outcomes of this work will be twofold: To provide the secure and efficient hierarchical deterministic wallet that we need today and to set a new standard for future wallets.

References

[1] Gavik Andresen. BIP16: Pay to script hash, 2012. URL `https://github.com/bitcoin/bips/blob/master/bip-0016.mediawiki`. Last visited June 13, 2019.

[2] Mikhail J. Atallah, Marina Blanton, and Keith B. Frikken. Incorporating Temporal Capabilities in Existing Key Management Schemes. In *Computer Security ESORICS 2007*, pages 515–530. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. doi: 10.1007/978-3-540-74835-9_34. URL `http://link.springer.com/10.1007/978-3-540-74835-9_34`.

[3] Mikhail J. Atallah, Marina Blanton, Nelly Fazio, and Keith B. Frikken. Dynamic and efficient key management for access hierarchies. *ACM Trans. Inf. Syst. Secur.*, 12(3):

18:1–18:43, January 2009. ISSN 1094-9224. doi: 10.1145/1455526.1455531. URL http://doi.acm.org/10.1145/1455526.1455531.

[4] Giuseppe Ateniese, Daniel H. Chou, Breno de Medeiros, and Gene Tsudik. Sanitizable Signatures. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 3679 LNCS, pages 159–177, 2005. ISBN 3540289631. doi: 10.1007/11555827_10. URL http://link.springer.com/10.1007/11555827_10.

[5] Giuseppe Ateniese, Alfredo De Santis, Anna Lisa Ferrara, and Barbara Masucci. Provably-secure time-bound hierarchical key assignment schemes. *Journal of Cryptology*, 25(2): 243–270, Apr 2012. ISSN 1432-1378. doi: 10.1007/s00145-010-9094-6. URL https://doi.org/10.1007/s00145-010-9094-6.

[6] Vitalik Buterin. Deterministic wallets, their advantages and their understated flaws, 2013. URL https://bitcoinmagazine.com/articles/deterministic-wallets-advantages-flaw-1385450276/. Last visited June 13, 2019.

[7] Nicolas Courtois, Pinar Emirdag, and Filippo Valsorda. Private key recovery combination attacks: On extreme fragility of popular bitcoin key management, wallet and cold storage solutions in presence of poor rng events. *IACR Cryptology ePrint Archive*, 2014:848, 2014.

[8] Jason Crampton, Naomi Farley, Gregory Gutin, Mark Jones, and Bertram Poettering. Cryptographic enforcement of information flow policies without public information via tree partitions. *Journal of Computer Security*, 25(6):511–535, 2017. ISSN 0926227X. doi: 10.3233/JCS-16863.

[9] Alfredo De Santis, Anna Lisa Ferrara, and Barbara Masucci. New constructions for provably-secure time-bound hierarchical key assignment schemes. *Theoretical Computer Science*, 407(1-3):213–230, 2008. ISSN 03043975. doi: 10.1016/j.tcs.2008.05.021. URL http://dx.doi.org/10.1016/j.tcs.2008.05.021.

[10] Pratyush Dikshit and Kunwar Singh. Efficient weighted threshold ECDSA for securing bitcoin wallet. In *2017 ISEA Asia Security and Privacy (ISEASP)*, volume 2, pages 1–9. IEEE, jan 2017. ISBN 978-1-5090-5942-3. doi: 10.1109/ISEASP.2017.7976994. URL http://ieeexplore.ieee.org/document/7976994/.

[11] Chun-I Fan, Yi-Fan Tseng, Hui-Po Su, Ruei-Hau Hsu, and Hiroaki Kikuchi. Secure hierarchical bitcoin wallet scheme against privilege escalation attacks. In *IEEE Conference on Dependable and Secure Computing, DSC 2018, Kaohsiung, Taiwan, December 10-13, 2018*, pages 1–8. IEEE, 2018. ISBN 978-1-5386-5790-4. doi: 10.1109/DESEC.2018.8625151. URL https://doi.org/10.1109/DESEC.2018.8625151.

[12] Rosario Gennaro, Steven Goldfeder, and Arvind Narayanan. Threshold-Optimal DSA/ECDSA Signatures and an Application to Bitcoin Wallet Security. In Jianying Zhou, Moti Yung, and Yongfei Han, editors, *Applied Cryptography and Network Security*, volume 2846 of *Lecture Notes in Computer Science*, pages 156–174. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016. ISBN 978-3-540-20208-0. doi: 10.1007/978-3-319-39555-5_9. URL http://link.springer.com/10.1007/978-3-319-39555-5_9.

[13] Steven Goldfeder, Rosario Gennaro, Harry Kalodner, Joseph Bonneau, Joshua A. Kroll, Edward W. Felten, and Arvind Narayanan. Securing bitcoin wallets via a new dsa/ecdsa threshold signature scheme. Unpublished, 2015.

[14] Gus Gutoski and Douglas Stebila. Hierarchical deterministic bitcoin wallets that tolerate key leakage. In *Financial Cryptography and Data Security*, pages 497–504. Springer Berlin Heidelberg, 2015. doi: 10.1007/978-3-662-47854-7_31. URL `https://doi.org/10.1007/978-3-662-47854-7_31`.

[15] Marek Palatinus and Pavol Rusnak. BIP43: Purpose field for deterministic wallets, 2014. URL `https://github.com/bitcoin/bips/blob/master/bip-0043.mediawiki`. Last visited June 13, 2019.

[16] Marek Palatinus and Pavol Rusnak. BIP44: Multi-account hierarchy for deterministic wallets, 2014. URL `https://github.com/bitcoin/bips/blob/master/bip-0044.mediawiki`. Last visited June 13, 2019.

[17] Marek Palatinus, Pavol Rusnak, Aaron Voisine, and Sean Bowe. BIP39: Mnemonic code for generating deterministic keys, 2013. URL `https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki`. Last visited June 13, 2019.

[18] Pieter Wuille. BIP32: Hierarchical deterministic wallets, 2012. URL `https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki`. Last visited June 13, 2019.

A Security proofs

A.1 Proof of Theorem 4.1

We prove the theorem by contradiction, using a hybrid argument. Let v^* be the challenge chosen by an adversary A in the game $\mathbf{G}^{\mathsf{sk-ind}}_{\Pi,\mathsf{A}}(\lambda, G)$. We define the following hybrid experiments:

$\mathbf{G}_{-1}$: is exactly the game $\mathbf{G}^{\mathsf{sk-ind}}_{\Pi,\mathsf{A}}(\lambda, G)$.

$\mathbf{G}_0$: is the same as $\mathbf{G}_{-1}$, except that the secret S_0 of the root node $v_0 \in Anc(v^*)$ is sampled at random.

$\mathbf{G}^{(a)}_i$: is the same as $\mathbf{G}^{(c)}_{i-1}$ (for $i = 1$ is the same as $\mathbf{G}_0$), except that t_{i-1}, x_{i-1} associated to the node $v_{i-1} \in Anc(v^*)$ and S_i of the node $v_i \in Anc(v^*)$ are sampled at random.

$\mathbf{G}^{(b)}_i$: is the same as $\mathbf{G}^{(a)}_i$, except that r_{ij} associated to the edge (v_i, v_j) (where $v_i, v_j \in Anc(v^*)$) is sampled at random.

$\mathbf{G}^{(c)}_i$: is the same as $\mathbf{G}^{(b)}_i$, except that y_{ij} associated to the edge (v_i, v_j) (where $v_i, v_j \in Anc(v^*)$) is an encryption of a random message, *i.e.*, $y_{ij} \leftarrow\!\!\$\ \mathsf{Enc}_{r_{ij}}(\hat{m})$ where $\hat{m}$ is sampled at random.

Our DHKA is identical to the HKA of Atallah *et al.* [3], except that the secret S_i of a node v_i is computed by evaluating $S_i = \mathsf{F}_{S_j}(11 \| l_i)$ where S_j is the secret of the parent v_j of v_i (in [3] each S_i is sampled at random). Hence, the proof is analogous to [3, Theorem 5.3] except that we need to prove that each S_i is indistinguishable from random. For this reason, we modify the game $\mathbf{G}^{(a)}_i$ (defined in [3, Theorem 5.3]) in such a way that the secret S_i is sampled at random too (in addition to t_{i-1}, k_{i-1}). Then, we prove the same result for the root node v_0 by adding an additional game $\mathbf{G}_{-1}$ and by showing $\mathbf{G}_{-1} \approx_c \mathbf{G}_0$.

Lemma A.1. *Let $\{\mathsf{F}_k\}_{k \in \mathcal{K}_\lambda}$ be a secure pseudorandom function, then $\mathbf{G}_{-1} \approx_c \mathbf{G}_0$.*

Proof. We assume that there exists a DAG $G = (V, E)$ and a distinguisher D that has a non-negligible advantage in distinguishing between $\mathbf{G}_{-1}$ and $\mathbf{G}_0$. Then, we build an adversary A that distinguishes $\mathbf{G}^{\mathsf{prf}-0}_{\mathsf{F},\mathsf{A}}(\lambda)$ and $\mathbf{G}^{\mathsf{prf}-1}_{\mathsf{F},\mathsf{A}}(\lambda)$ as follows:

1. D outputs the challenge v^*.
2. A simulates Set as follows: For the root node v_0, set $S_0 = \mathsf{O}_{\mathsf{F}}(11||l_0)$. For any other node v_j, compute S_j as described in Construction 1. Then, for each node $v_i \in V$ and for each edge $(v_i, v_j) \in E$, compute the secret values $t_i = \mathsf{F}_{S_i}(00\|l_i)$, $x_i = \mathsf{F}_{S_i}(01\|l_i)$, $r_{ij} = \mathsf{F}_{t_i}(10\|l_j)$, $y_{ij} \leftarrow\!\!\$\, \mathsf{Enc}_{r_{ij}}(t_j\|x_j)$ as described in Construction 1. A sets $x^0_{v^*} = x_{v^*}$ and $x^1_{v^*} = \bar{x}_{v^*}$ where $\bar{x}_{v^*}$ is sampled at random. Finally, A sends Pub and $x^d_{v^*}$ to D where d is a random bit.
3. A answers any $\mathsf{O}^{\Pi}_{\mathsf{Corr}}(v_i)$ query by returning S_i.
4. D outputs a bit d' and A completes the simulation of the experiments $\mathbf{G}_{-1}$ and $\mathbf{G}_0$ by returning 1 if $d = d'$; otherwise it returns 0.
5. Lastly, D outputs its guess. A outputs any bit b that D outputs.

When A is playing respectively $\mathbf{G}^{\mathsf{prf}-0}_{\mathsf{F},\mathsf{A}}(\lambda)$ and $\mathbf{G}^{\mathsf{prf}-1}_{\mathsf{F},\mathsf{A}}(\lambda)$, then the reduction perfectly simulates $\mathbf{G}_{-1}$ and $\mathbf{G}_0$. Indeed, if A is playing with $\mathbf{G}^{\mathsf{prf}-0}_{\mathsf{F},\mathsf{A}}(\lambda)$ (resp. $\mathbf{G}^{\mathsf{prf}-1}_{\mathsf{F},\mathsf{A}}(\lambda)$) then, $S_0 = \mathsf{O}_{\mathsf{F}}(11\|l_0)$ (resp. S_0 is randomly sampled from $\{0,1\}^*$). In addition, A computes all the secrets and edge information following Construction 1. As such, the advantage of the attacker A in distinguishing $\mathbf{G}^{\mathsf{prf}-0}_{\mathsf{F},\mathsf{A}}(\lambda)$ and $\mathbf{G}^{\mathsf{prf}-1}_{\mathsf{F},\mathsf{A}}(\lambda)$ is non negligible. This concludes the proof. □

The rest of the proof is analogous to the one of Atallah *et al.*, except that in $\mathbf{G}^{(a)}_i$ we additionally sample S_i at random. We refer to [3, Theorem 5.3] for the proofs that $\mathbf{G}_0 \approx_c \mathbf{G}^{(a)}_1$ and $\mathbf{G}^{(a)}_i \approx_c \mathbf{G}^{(b)}_i$, $\mathbf{G}^{(b)}_i \approx_c \mathbf{G}^{(c)}_i$, $\mathbf{G}^{(c)}_{i-1} \approx_c \mathbf{G}^{(a)}_i$ for any $i \in \{2, \ldots, |Anc(v^*)| - 1\}$.

A.2 Proof of Theorem 5.1

We prove the theorem by contradiction, using a hybrid argument. Let (v_j, m, σ) be the forgery returned by A in the game $\mathbf{G}^{\mathsf{heuf}}_{\Pi,\mathsf{A}}(\lambda, G)$. We define the following hybrid experiments:

$\mathbf{G}_0$: is exactly the game $\mathbf{G}^{\mathsf{heuf}}_{\Pi,\mathsf{A}}(\lambda, G)$.

$\mathbf{G}_t$: is the same as $\mathbf{G}_{t-1}$, except that the challenger generates at random the signature key pairs $(\mathsf{sk}'_i, \mathsf{pk}'_i)$ for the first t nodes in $Anc(v_j)$. More in details, let $Anc(v_j) = \{v_0, \ldots, v_t, \ldots, v_j\}$, for every $v_i \in \{v_0, \ldots, v_t\}$ the challenger generates the signature key pair $(\mathsf{sk}'_i, \mathsf{pk}'_i)$ by running $\mathsf{KGen}_{\Sigma}(1^{\lambda})$.

The proof idea is to first show, using a hybrid argument, that $\mathbf{G}_0 \approx_c \mathbf{G}_{|Anc(v_j)|}$. Hence, a potential adversary A has the same advantage in both $\mathbf{G}_0$ and $\mathbf{G}_{|Anc(v_j)|}$, with overwhelming probability. Then, we show that an adversary A for $\mathbf{G}_{|Anc(v_j)|}$ implies an adversary A' for $\mathbf{G}^{\mathsf{euf}}_{\Sigma,\mathsf{A}'}(\lambda)$.

Lemma A.2. *If Γ is key indistinguishable, then $\mathbf{G}_{t-1} \approx_c \mathbf{G}_t$ for every $1 \leq t \leq |Anc(v_j)|$.*

Proof. We assume that there exists a DAG $G = (V, E)$ and a distinguisher D that has a non-negligible advantage in distinguishing between $\mathbf{G}_{t-1}$ and $\mathbf{G}_t$. Then, we build an adversary A against the experiment $\mathbf{G}^{\mathsf{sk}-\mathsf{ind}}_{\Gamma,\mathsf{A}}(\lambda, G)$ (defined in Definition 4.2) as follows:

1. A samples at random v^*. Let $Anc(v^*) = \{v_0, \ldots, v_t, \ldots, v^*\}$ be the set of ancestors of v^* according to an ordering of the nodes of the graph (*e.g.*, a topological sorting). A sends v_t to the challenger and receives Pub and x_t .

2. A samples the pair $(\mathsf{wsk}, \mathsf{wpk}) \leftarrow\!\!\$\ \mathsf{KGen}_\Sigma(1^\lambda)$.

3. A executes the remaining steps of Set_Π (Item 2 of Construction 2), except that it skips Item 2a and it replaces Item 2b with the following:

   - If $v_i \in \{v_0, \ldots, v_{t-1}\}$, then compute $(\mathsf{sk}'_i, \mathsf{pk}'_i) \leftarrow\!\!\$\ \mathsf{KGen}_\Sigma(1^\lambda)$.
   - Otherwise, if $v_i = v_t$, then compute $(\mathsf{sk}'_t, \mathsf{pk}'_t) = \mathsf{KGen}_\Sigma(1^\lambda; x_t)$.
   - Otherwise, send a $\mathsf{O}^\Gamma_{\mathsf{Corr}}(v_i)$ query to the challenger and receive $S_i = \mathsf{msk}_i$. Compute $x_i = \mathsf{Derive}_\Gamma(G, \mathsf{Pub}, v_i, v_i, S_i)$ and $(\mathsf{sk}'_i, \mathsf{pk}'_i) = \mathsf{KGen}_\Sigma(1^\lambda; x_i)$.

   Finally, A outputs the master public key $\mathsf{mpk} = (G, \mathsf{Pub}, \{\hat{\sigma}_i\}_{v_i \in V}, \mathsf{wpk})$.

4. A answers oracle queries in the following way:

   - On input v_i for $\mathsf{O}^\Pi_{\mathsf{Corr}}$, A invokes $\mathsf{O}^\Gamma_{\mathsf{Corr}}(v_i)$ and returns the output.
   - On input (m, v_i) for $\mathsf{O}^\Pi_{\mathsf{Sign}}$, A returns $\sigma = (\mathsf{pk}'_i, \sigma', \hat{\sigma}_i)$ where $\sigma' \leftarrow\!\!\$\ \mathsf{Sign}_\Sigma(\mathsf{sk}'_i, m)$.

5. A receives the forgery (v_j, m, σ). It aborts the simulation if $v^* \neq v_j$; otherwise it completes the simulation by returning the result of $\mathsf{Vrfy}_\Pi(\mathsf{pk}_j, m, \sigma)$, where $l_j = \mathsf{Pub}(v_j)$ and $\mathsf{pk}_j = (\mathsf{wpk}, l_j)$.

6. A outputs the decisional bit received from D.

Let E_{abort} be the event that A aborts the simulation. It is easy to see that $\Pr[\neg E_{\mathsf{abort}}] = \Pr[v^* = v_j] = \frac{1}{|V|}$. Let $\mathbf{G}^{\mathsf{sk-ind}-b}_{\Gamma,\mathsf{A}}(\lambda, G)$ be the key indistinguishability game with bit b. Conditioned on the event $\neg E_{\mathsf{abort}}$, when A is playing respectively $\mathbf{G}^{\mathsf{sk-ind}-0}_{\Gamma,\mathsf{A}}(\lambda, G)$ and $\mathbf{G}^{\mathsf{sk-ind}-1}_{\Gamma,\mathsf{A}}(\lambda, G)$, then the reduction perfectly simulates $\mathbf{G}_{t-1}$ and $\mathbf{G}_t$, because D can not corrupt any node $v \in Anc(v^*)$. Hence, the advantage of the attacker A in winning the game $\mathbf{G}^{\mathsf{sk-ind}}_{\Gamma,\mathsf{A}}(\lambda, G)$ is non-negligible. This concludes the proof. □

Lemma A.3. *If Σ is existentially unforgeable, then for every DAG $G = (V, E)$ and PPT adversary A,* $\Pr\left[\mathbf{G}_{|Anc(v_j)|,\mathsf{A}}(\lambda, G) = 1\right] \leq \mathsf{negl}(\lambda)$.

Proof. We assume that there exists a DAG $G = (V, E)$ and an adversary A that has a non-negligible advantage against $\mathbf{G}_{|Anc(v_j)|,\mathsf{A}}(1^\lambda, G)$. Then, we build an adversary A' against $\mathbf{G}^{\mathsf{euf}}_{\Sigma,\mathsf{A}'}(\lambda)$ as follows:

1. A' receives pk^* from the challenger.

2. A' flips a bit $d \leftarrow\!\!\$\ \{0, 1\}$ and samples at random $v^* \leftarrow\!\!\$\ V$, $S_{-1} = (S^0_{-1}, S^1_{-1}) \leftarrow\!\!\$\ \mathcal{S}$.

3. A' simulates Set_Π. It runs $(\mathsf{Pub}, \mathsf{Sec}) = \mathsf{Set}_\Gamma(1^\lambda, G, S^0_{-1})$. If $d = 0$, it sets $\mathsf{wpk} = \mathsf{pk}^*$; otherwise it runs $(\mathsf{wsk}, \mathsf{wpk}) = \mathsf{KGen}_\Sigma(1^\lambda; S^1_{-1})$. Lastly, A' executes the remaining steps of Set_Π (Item 2 of Construction 2), except it replaces Item 2b and Item 2c with the following:

   Item 2b: A' proceeds as follow:

   - If $v_i \in Anc(v^*) \setminus \{v^*\}$, then compute $(\mathsf{sk}'_i, \mathsf{pk}'_i) \leftarrow\!\!\$\ \mathsf{KGen}_\Sigma(1^\lambda)$.
   - If $v_i = v^*$, set $\mathsf{pk}'_i = \mathsf{pk}^*$ if $d = 1$; otherwise run $(\mathsf{sk}'_i, \mathsf{pk}'_i) \leftarrow\!\!\$\ \mathsf{KGen}_\Sigma(1^\lambda)$.

- Otherwise (if $v_i \notin Anc(v^*)$), run $(\mathsf{sk}'_i, \mathsf{pk}'_i) = \mathsf{KGen}_\Sigma(1^\lambda; x_i)$ where $(S_i, x_i) = \mathsf{Sec}(v_i)$.

Item 2c: If $d = 1$, then retrieve the label $l_i = \mathsf{Pub}(v_i)$ and compute $\hat{\sigma}_i \leftarrow\!\!\$\ \mathsf{Sign}_\Sigma(\mathsf{wsk}, (\mathsf{pk}'_i, l_i))$; otherwise, set $\hat{\sigma}_i \leftarrow\!\!\$\ \mathsf{O}^\Sigma_{\mathsf{Sign}}((\mathsf{pk}'_i, l_i))$.

Finally, A' sends to A the master public key $\mathsf{mpk} = (G, \mathsf{Pub}, \{\hat{\sigma}_i\}_{v_i \in V}, \mathsf{wpk})$.

4. A' answers oracle queries in the following way:
   - On input v_i for $\mathsf{O}^\Pi_{\mathsf{Corr}}$, A' returns $\mathsf{msk}_i = S_i$ where $(S_i, x_i) = \mathsf{Sec}(v_i)$.
   - On input (m, v_i) for $\mathsf{O}^\Pi_{\mathsf{Sign}}$, if $d = 1 \wedge v_i = v^*$, A' sets $\sigma' \leftarrow\!\!\$\ \mathsf{O}^\Sigma_{\mathsf{Sign}}(m)$; otherwise, it computes $\sigma' \leftarrow\!\!\$\ \mathsf{Sign}_\Sigma(\mathsf{sk}'_i, m)$. Lastly, it returns $\sigma = (\mathsf{pk}'_i, \sigma', \hat{\sigma}_i)$.
5. A' receives the forgery $(v_j, \tilde{m}, \tilde{\sigma})$ such that $\tilde{\sigma} = (\mathsf{pk}^\bullet_j, \sigma^\bullet, \hat{\sigma}^\bullet_j)$ and aborts the simulation if $v^* \neq v_j \vee (d = 0 \wedge \mathsf{pk}^\bullet_j = \mathsf{pk}'_j) \vee (d = 1 \wedge \mathsf{pk}^\bullet_j \neq \mathsf{pk}'_j)$. Otherwise, if $d = 0$, it sends the forgery $((\mathsf{pk}^\bullet_j, l_j), \hat{\sigma}^\bullet_j)$ to challenger where $l_j = \mathsf{Pub}(v_j)$; if $d = 1$ sends $(m, \sigma^\bullet)$.

Let E_{abort} be the event that A' wins the game $\mathbf{G}^{\mathsf{euf}}_{\Sigma,\mathsf{A}'}(\lambda)$ and aborts the simulation. First of all, note that:

$$\begin{aligned}
\neg E_{\mathsf{abort}} &= \neg\left[v^* \neq v_j \vee (d = 0 \wedge \mathsf{pk}^\bullet_j = \mathsf{pk}'_j) \vee (d = 1 \wedge \mathsf{pk}^\bullet_j \neq \mathsf{pk}'_j)\right] \\
&= \left[v^* = v_j \wedge \neg(d = 0 \wedge \mathsf{pk}^\bullet_j = \mathsf{pk}'_j) \wedge \neg(d = 1 \wedge \mathsf{pk}^\bullet_j \neq \mathsf{pk}'_j)\right] \\
&= \left[v^* = v_j \wedge (d = 1 \vee \mathsf{pk}^\bullet_j \neq \mathsf{pk}'_j) \wedge (d = 0 \vee \mathsf{pk}^\bullet_j = \mathsf{pk}'_j)\right] \\
&= \left[v^* = v_j \wedge ((d = 0 \wedge d = 1) \vee (d = 1 \wedge \mathsf{pk}^\bullet_j = \mathsf{pk}'_j)\right. \\
&\qquad \left.\vee (d = 0 \wedge \mathsf{pk}^\bullet_j \neq \mathsf{pk}'_j) \vee (\mathsf{pk}^\bullet_j \neq \mathsf{pk}'_j \wedge \mathsf{pk}^\bullet_j = \mathsf{pk}'_j))\right] \\
&= \left[v^* = v_j \wedge ((d = 1 \wedge \mathsf{pk}^\bullet_j = \mathsf{pk}'_j) \vee (d = 0 \wedge \mathsf{pk}^\bullet_j \neq \mathsf{pk}'_j))\right]
\end{aligned}$$

Let $\Pr\left[\mathsf{pk}^\bullet_j = \mathsf{pk}'_j\right] = p$. We can express $\Pr[\neg E_{\mathsf{abort}}]$ in the following way:

$$\begin{aligned}
\Pr[\neg E_{\mathsf{abort}}] &= \Pr\left[v^* = v_j \wedge (d = 0 \wedge \mathsf{pk}^\bullet_j \neq \mathsf{pk}'_j) \vee (d = 1 \wedge \mathsf{pk}^\bullet_j = \mathsf{pk}'_j)\right] \\
&= \Pr[v^* = v_j] \cdot \left(\Pr\left[d = 0 \wedge \mathsf{pk}^\bullet_j \neq \mathsf{pk}'_j\right] + \Pr\left[d = 1 \wedge \mathsf{pk}^\bullet_j = \mathsf{pk}'_j\right]\right) \\
&= \Pr[v^* = v_j] \cdot \left(\Pr[d = 0] \cdot \Pr\left[\mathsf{pk}^\bullet_j \neq \mathsf{pk}'_j\right] + \Pr[d = 1] \cdot \Pr\left[\mathsf{pk}^\bullet_j = \mathsf{pk}'_j\right]\right) \\
&= \frac{1}{|\,V\,|} \cdot \left(\frac{1-p}{2} + \frac{p}{2}\right) = \frac{1}{2 \cdot |\,V\,|}
\end{aligned}$$

Let $\mathcal{Q}^\Sigma_{\mathsf{Sign}}$ and $\mathcal{Q}^\Pi_{\mathsf{Sign}}$ be respectively the set of queries submitted by A' to $\mathsf{O}^\Sigma_{\mathsf{Sign}}$ and the set of queries submitted by A to $\mathsf{O}^\Pi_{\mathsf{Sign}}$. Conditioned on $\neg E_{\mathsf{abort}}$ and since A is a valid adversary for $\mathbf{G}^{\mathsf{heuf}}_{\Pi,\mathsf{A}}(\lambda, G)$, then, with non-negligible probability, $\mathsf{Vrfy}_\Pi(\mathsf{pk}_j, \tilde{m}, \tilde{\sigma}) = 1$ if and only if $\mathsf{Vrfy}_\Sigma(\mathsf{wpk}, (\mathsf{pk}^\bullet_j, l_j), \hat{\sigma}^\bullet_j) = 1$ and $\mathsf{Vrfy}_\Sigma(\mathsf{pk}^\bullet_j, \tilde{m}, \sigma^\bullet) = 1$, where $\mathsf{pk}_j = (\mathsf{wpk}, l_j)$ and $l_j = \mathsf{Pub}(v_j)$. Note that A' outputs a valid forgery for $\mathbf{G}^{\mathsf{euf}}_{\Sigma,\mathsf{A}'}(\lambda)$ with probability $\frac{1}{2}$:

1. Whenever $d = 0$, we have $\mathsf{wpk} = \mathsf{pk}^*$ and $\mathsf{pk}^\bullet_j \neq \mathsf{pk}'_j$. This allows us to conclude that A' never asked $(\mathsf{pk}^\bullet_j, l_j)$ to oracle $\mathsf{O}^\Sigma_{\mathsf{Sign}}$ (*I.e.,* $(\mathsf{pk}^\bullet_j, l_j) \notin \mathcal{Q}^\Sigma_{\mathsf{Sign}}$). Hence, $((\mathsf{pk}^\bullet_j, l_j), \hat{\sigma}^\bullet_j)$ is a valid forgery for $\mathbf{G}^{\mathsf{euf}}_{\Sigma,\mathsf{A}'}(\lambda)$.
2. On the other hand, if $d = 1$, we have $\mathsf{pk}^\bullet_j = \mathsf{pk}'_j = \mathsf{pk}^*$. Since, A is a valid adversary it must produces a valid signature for a new fresh message. Hence, we can conclude that $(v^*, \tilde{m}) \notin \mathcal{Q}^\Pi_{\mathsf{Sign}}$ and $(\tilde{m}, \sigma^\bullet)$ is a valid forgery for $\mathbf{G}^{\mathsf{euf}}_{\Sigma,\mathsf{A}'}(\lambda)$.

This concludes the proof. □

By combining Lemma A.2 and Lemma A.3 we have that Construction 2 is hierarchically existentially unforgeable.