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

December 10, 2019

Abstract

This work presents Arcula, a new design for hierarchical deterministic wallets that brings identity-based addresses to the blockchain. 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 public keys based on the identities of users, without requiring any secret information. Unlike other wallets, it achieves all these properties while being secure against privilege escalation. We formalize the security model of hierarchical deterministic wallets and 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 decentralized and democratic financial world where users exchange their assets without relying on any central authority and where transactions get delivered to the destination in a handful of seconds, by spending cents of a dollar in fees. For these reasons, these systems attracted the interest of financial operators, banks, (decentralized) exchanges, and e-commerce marketplaces that aim to increase the security and the usability of their systems, reduce their business costs, and prepare for the financial market of the future. With the adoption of these systems at scale, however, we face a new set of technological and financial hurdles that we have never experienced before: (1) How to secure the digital assets of a commercial company? (2) How to handle the delegation and separation of responsibilities within the enterprise's chain of command? (3) And how to enable a third-party (*e.g.*, an auditor) to evaluate past transactions of the company, and (4) to gather the list of its assets on the blockchain?

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

In this work, we aim at solving these challenges, and 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 our crypto-assets (*e.g.*, our Bitcoins, Ethers, and other coins). In particular, wallets hold the cryptographic keys that allow us to spend these coins. Our goal is to design a wallet that is cryptographically secure and easy to use: We specifically tackle the use cases of the legacy and future blockchain-based crypto-systems and, in turn, the needs of the large-scale enterprises that rely on them. The wallet that we aim to design is hierarchical and deterministic. This means that: 1) The users can securely organize their keys in a hierarchy that reflects, for example, the subdivision in departments of a company. This way, the managers of a company can spend funds on behalf of their departments, but each department can only spend its own funds. 2) The wallet deterministically generates every cryptographic key by starting from an initial seed (*e.g.*, a pseudorandom sequence derived from a mnemonic phrase). As a result, the users can reliably recover all their keys even in the case of total loss (*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 (HKAs)*, a popular cryptographic primitive that has been studied for more than 20 years. Arcula implements arbitrarily complex access hierarchies, allows for their dynamic modifications, and can incorporate temporal capabilities into the generation and storage of keys. In addition, Arcula ties the identities of users to their public signing keys without requiring additional secret information — bringing to fruition identity-based hierarchical cryptography (*e.g.*, signatures) within the blockchain ecosystem. Unlike other wallets, Arcula achieves these properties while being formally secure against privilege escalation. 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 verification of signatures on arbitrary messages. We show its implementation in action 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 the properties of hierarchical deterministic wallets and their applications. Section 3 and Section 4 respectively discuss the related work and introduce the notation and the cryptographic primitives that we use throughout the paper. Section 5 formalizes the security model of hierarchical deterministic wallets. Section 6 and Section 7 describe our design of Arcula and show it in action in the real world. Lastly, Section 8 concludes the paper.

2 Hierarchical Deterministic Wallets

A hierarchical deterministic wallet (HDW) enables a user to securely generate and store the cryptographic keys associated with her coins. Blockchain-based crypto-systems typically rely on digital signatures and pairs of secret and public signing keys: Users spend their assets by signing a transaction with the secret key; the others verify the authenticity of the signature through the public key. Public keys can be derived from the corresponding secret keys, but not vice-versa. On a high level, a hierarchical deterministic wallet holds a collection of secret signing keys.

An HDW deterministically generates its keys by starting from an initial seed provided by the user. As long as the user remembers the seed, she will be able to recover her keys, even in case of wallet loss. Besides, an HDW also organizes the keys under an access hierarchy, where each element represents a group of users and associated a pair of signing keys to them. The privileges of users depend on their level in the hierarchy. Users with higher privileges (*i.e.*, higher in the hierarchy), must 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 along the hierarchy, not even when colluding with others. Finally, an HDW should also provide an additional fundamental property: It must be possible to deterministically generate every public key of the wallet by starting from a *master public key* (depending on the user seed), without requiring any secret information in the process. As we shall see, this requirement enables a set of creative use cases for HDW (*e.g.*, public auditing of blockchain assets, and generation of new keys in untrusted environments).

2.1 Properties

In more detail, let sk_i be the secret signing key of a user v_i of a hierarchy and let pk_i be the corresponding public key. Let sk_j and pk_j be, respectively, the secret and public keys associated with a descendant v_j of v_i (*i.e.*, a user with lower privileges in the hierarchy). A hierarchical deterministic wallet shall have the following properties:

Property 2.1 (Security against privilege escalation)**.** For any set of colluding descendants of v_i it is computationally infeasible to recover the secret key sk_i of v_i.

Property 2.2 (Deterministic secret derivation)**.** For each descendant v_j of v_i, the secret keys sk_j is deterministically generated by using the secret information of v_i. If v_j has the highest privileges in the hierarchy, then her secret information are generated from a user-provided seed.

Property 2.3 (Public-key derivation)**.** The public key pk_j of each descendant v_j of v_i is deterministically generated only using public information; the generation process does not require the secret key sk_i or any other secret information.

The three properties, together, define the ideal hierarchical deterministic wallet. Property 2.1 guarantees the security of the wallet: Colluding users cannot escalate their privileges to recover the secret keys of others higher in the hierarchy. Property 2.2 ensures that all secrets are deterministically generated along the hierarchical path that links the most privileged users to their descendants, and so on. Property 2.3 requires the public keys to be dynamically derivable without requiring any private information. As we will see, in the past, this property proved to be particularly hard to achieve while also guaranteeing the security of the underlying wallets. Nonetheless, it is crucially important as it enables the novel applications of HDW within the blockchain that we discuss in Section 2.2. Finally, we note that every wallet in which the public derivation property holds, inevitably reveals the relation between the public keys of the wallet, making it impossible to achieve any privacy-related notion, *e.g.*, unlinkability of transactions. We discuss this issue in Section 7.5.

2.2 Applications

Hierarchical deterministic wallets enable different use cases, inspired by both well established and innovative financial applications that specifically tackle the needs of enterprises, governments, and financial institutions. Individual users will also find HDW useful, leveraging their increased security and their deterministic reliability and, if they choose so, to achieve unlinkability of their transactions.

We discuss the benefits of HDW for individuals in Section 7.5. Here, instead, we focus on the applications of HDW at scale, that target (hundreds of) thousands of customers distributed across the world.

Enterprises: In enterprises (*e.g.*, financial institutions, or exchanges), the hierarchy of a deterministic wallet might reflect the underlying chain of command or the subdivision in regions, departments, and teams. It allows managers to distribute funds among different branches and ensure fiscal responsibility. In particular, each branch can manage its funds but are not allowed to spend those of other units. The deterministic generation of keys simplifies the management of secrets and guarantees a reliable recovery of the wallet, even in case of catastrophic loss. Cryptocurrency exchanges that manage the keys of hundreds of thousands of users might find this feature particularly useful: Through the HDW, they generate pairs of keys that take into account the hierarchy of users and then rely on the master seed to handle their recovery. Finally, the property of public-key derivation enables enterprises to comply with financial laws and regulations without jeopardizing the security of their infrastructure: *E.g.* it allows a (possibly untrusted) auditor to inspect the funds that they hold by starting from the master public key and deriving all the public keys in the wallet without relying on any additional secret information.

e-commerce: An HDW is distinctly beneficial to an e-commerce marketplace. Marketplaces, such as Amazon, typically advertise and sell products to buyers. They also allow third-party vendors to do the same. Cryptocurrencies could help manage the payment flow to these vendors. When selling an item to a buyer, the marketplace generates a fresh payment address for each crypto-coin that it supports. As soon as the buyer transfers the required coins to one of the addresses and the blockchain confirms the transaction, the item gets shipped. The generation of fresh payment addresses leverages the properties of public-key derivation: Since it does not require any private information, it can take place in an untrusted environment (*e.g.*, a web server exposed to the internet) and allows the e-commerce owner to derive the corresponding secret keys only when actually spending the funds (*e.g.*, by deriving them offline starting from an intermediate key of the wallet). Even if an attacker compromises the webserver, he will not discover any secret keys, and thus funds received before the attack remain safe and intact. In addition, public-key derivation allows buyers or auditors to check the authenticity of the payment addresses since anyone can generate them from the public key of the marketplace.

Decentralized Finance (DeFi): Decentralized Finance has recently started replacing many of the existing traditional financial tools (*e.g.*, loans and futures) with open-source alternatives based on the blockchain and its smart contracts. With DeFi, HDW can unleash their full potential: The execution of smart contracts, indeed, cannot rely on any secret information as both the source code and the processing inputs are stored in the clear on the public blockchain. By combining HDW and public derivation, a DeFi smart contract can autonomously derive the public key of the recipient of a transaction (*e.g.*, a user that will receive interests on a loan, or the employee of a company that will receive a percentage of its shares). The process could take place on the blockchain and does not require manual interactions. In turn, HDW and DeFi lay the foundations of a modern, democratic, and decentralized financial world.

2.3 Threats and Security Model

We divide the set of possible threats to the security of hierarchical deterministic wallet in three security levels that we describe according to increasing requirements of trust.

Untrusted Environment: This level is entirely untrusted. It refers, for example, to the executing environment of a DeFi smart contract that relies on public key derivation (Property 2.3) to generate fresh addresses to deliver payments.

Hot Environment: This level is semi-trusted. At this stage, the users can access their own secret keys and derive those of their descendants. An attacker that compromises a user

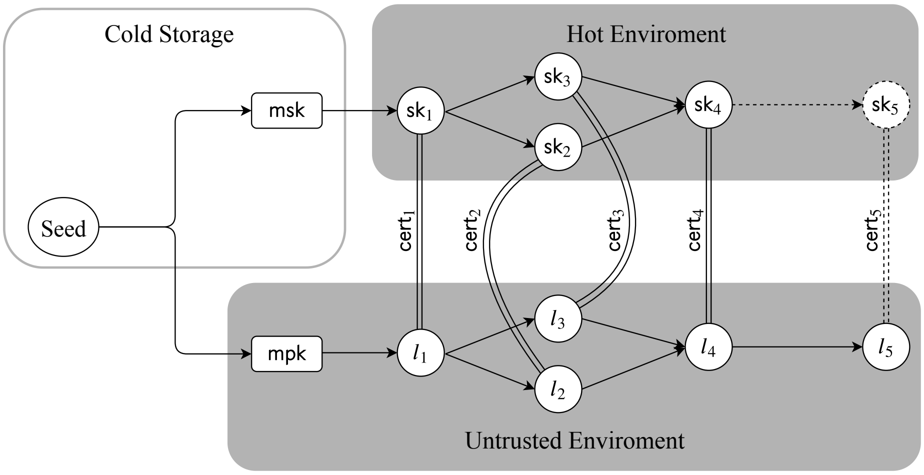

Figure 1: A glance at the deterministic generation of secrets and identity-based public key derivation within Arcula. The users l_1 to l_5 at the bottom of the figure create a hierarchy, encoded as a directed acyclic graph. Arcula starts by deriving a pair of master keys $(\mathsf{msk}, \mathsf{mpk})$ from the seed. The users are unequivocally identified by concatenating the master public key mpk with their identities (*e.g.*, l_5). The master secret key msk allows the deterministic generation of secrets, so that every secret transitively depends on the initial seed. The hierarchical deterministic key assignment at the core of Arcula relies on the msk and enables users with higher privileges to derive the secrets of their descendants (*e.g.*, enabling l_1 to start from her secret information and derive the secret key sk_4 of l_4). Finally, Arcula associates secret keys to their corresponding identities by providing each user with a certificate signed by the msk: cert_4, *e.g.*, associates l_4 with the secret key sk_4. This approach enables Arcula to flexibly add new users to the hierarchy without requiring secret information. The user l_5 was dynamically added to the hierarchy, and her public key was obtained through identity-based derivation, but she still does not have any secret information associated with her. This means that she can receive funds (*e.g.*, on the blockchain) and that the certificate and her secret information will only be required when she will spend them.

of the hot environment will compromise, in turn, only her descendants in the hierarchy (Property 2.1).

Cold Storage: This is the most trusted level of the security model, holding the seed used to generate every key within the wallet deterministically. It typically corresponds to an offline location (*e.g.*, a hardware token used to instantiate the wallet) that is physically secured (*e.g.*, in a safe). An attacker that compromises the cold storage has full access to the wallet and to every asset that it holds.

In Sections 2.4 and 3, we analyze our construction and the related work under the spotlight of this security model.

2.4 Arcula at First Glance

Figure 1 provides an overview of the design of Arcula, our hierarchical deterministic wallet. Arcula starts from the seed to deterministically generate a master pair of secret and public keys $(\mathsf{msk}, \mathsf{mpk})$. After the initial instantiation, the seed and the master secret key can be safely moved to cold storage. The master public key uniquely identifies the wallet and will be used in the public key derivation process. In more detail, Arcula generates the public key

Table 1: Comparison between Arcula and the existing state-of-the-art solutions.

	Security to Privilege Escalation (Property 2.1)	Public Key Derivation (Property 2.3)	Deterministic Generation (Property 2.2)	Hierarchy
BIP32 [21]	No	Yes	Yes	Tree
Hardened BIP32 [21]	Yes	No	Yes	Tree
Gutoski and Stebila [16]	No	Bounded	Yes	Tree
Fan *et al.* [12]	No	Yes	Yes	Tree
Poulami *et al.* [9]	–	–	–	No
Goldfeder *et al.* [15]	–	–	–	No
Gennaro *et al.* [13]	–	–	–	No
Dikshit and Singh [11]	–	–	–	No
Arcula (Section 6)	Yes	Yes	Bounded	DAG

of i-th user of the hierarchy by concatenating the master public key mpk and her identity l_i (*e.g.*, a numerical index or a bit string). As a result, public-key derivation in Arcula can be executed, by design, in any untrusted environment. The master secret key msk allows, instead, to deterministically generate the secret keys sk_i corresponding to the users of the hierarchy. In particular, Arcula generates the secret keys by relying on deterministic hierarchical key assignment: A provably secure cryptographic scheme through which we assign a derivation key to every user of the hierarchy, that, in turn, they will use to derive their own secret key and those of their descendants. The private key generation should be executed in the context of the hot environment, as compromising a single user will lead to compromising every descendant. Finally, Arcula explicitly associates secret keys to their corresponding identity-based public keys through a certificate cert_i, signed by the master secret key msk, that links the public identity l_i (top of Figure 1) to the secret key sk_i generated by the deterministic key assignment scheme (bottom of Figure 1).

The identity-based approach that we realized with Arcula provides several advantages: First, it solves the problem of distributing a public key for each user of the hierarchy and of associating, off the blockchain, to the identity of an individual. In addition, it allows for generating an unbounded number of addresses for receiving transactions. On the other hand, Arcula relies on certificates signed by the master secret key to associate the secret key of a user with their identity, to which the transaction was addressed. The users only require these certificates when they sign a transaction for the first time: This means that their creation can be delayed until that moment and that it can happen entirely offline.

3 Related Work

Bitcoin Improvement Proposal 32 (BIP32) defines the state of the art implementation of hierarchical deterministic wallets [21]. In short, let g be the generator point of an Elliptic Curve. A private key sk_i is associated with its public key $\mathsf{pk}_i = g^{\mathsf{sk}_i}$. Let H be a hash function; the descendants' private keys sk_j is defined as:

$$\mathsf{sk}_j = \mathsf{H}(\mathsf{pk}_i \| j) + \mathsf{sk}_i \qquad (1)$$

The corresponding public keys pk_j, instead:

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

Equations (1) and (2) satisfy the properties of deterministic generation and public derivation (Properties 2.2 and 2.3). However, Equation (1) creates a privilege escalation vulnerability where the knowledge of a descendant private key sk_j and the parent public key pk_i allows recovering the parent private key sk_i:

$$\mathsf{sk}_i = \mathsf{sk}_j - \mathsf{H}(\mathsf{pk}_i \| j) \mod q$$

This privilege escalation vulnerability has been discussed extensively [21, 6, 7, 16]. In the context of our threat model, there is no distinction between the cold storage and the hot environment, since compromising any node leads to compromising the entire wallet.

BIP32 addresses this issue by designing a *hardened* key derivation method that generates a descendant private key sk_j^h as follows:

$$\mathsf{sk}_j^h = \mathsf{H}(\mathsf{sk}_i \| i) + \mathsf{sk}_i \mod q \tag{3}$$

The hardened derivation solves the privilege escalation vulnerability but looses the public key derivation (*i.e.*, trades Property 2.3 for Property 2.1). Generating a hardened public key pk_j^h now requires the parent secret key sk_i:

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

In Table 1 we compare Arcula, our hierarchical deterministic wallet, with (hardened) BIP32 and the related works. BIP32 does not satisfy the security to privilege escalation property. The hardened version of BIP32, instead, fails in deriving public keys without requiring additional secrets. Gutoski and Stebila [16] propose an HDW strengthens the security of BIP32. Their design splits each secret key into n shares, distributed to the descendants of the user; reconstructing the secret key requires at least m shares. This solution provides weaker security than Property 2.1, because m colluding descendants of a user can recover the original secret key (as opposed to preventing any set of colluding descendants from escalating their privileges). In addition, they support Property 2.3 by publishing the public keys of all the users in the wallet. They do not allow the generation of fresh public keys, and their derivation is bounded to the number of published keys. Fan *et al.* [12] develop an HDW based on Schnorr signatures and trapdoor hash functions that enables the users to sign new transactions without accessing their private keys. A generic user can sign transactions on behalf of its descendants only after authorization by the root of the hierarchy that needs to reveal her the master private trapdoor key. As a result, any authorized user is able not only to sign new transactions on behalf of its descendants but also of all the users of the hierarchy. Compromising a single authorized user leads to revealing every secret stored in the wallet — the cold storage and hot environment of our threat model overlap, and the scheme is not secure against privilege escalation. Poulami *et al.* [9] provide a formal definition of non-hierarchical deterministic wallets and show a set of modifications that make ECDSA-based deterministic wallets provably secure. Goldfeder *et al.* [15] and subsequently Gennaro *et al.* [13] 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. Dikshit and Singh [11] extend the threshold-based ECDSA signatures to assign different weights to the participants of the protocol. These works deal with non-hierarchical deterministic wallets, and they do not aim at achieving Properties 2.1 to 2.3 (depending on the hierarchical structure of the wallet).

Arcula, on the other hand, is the only solution secure against privilege escalation that, at the same time, enables identity-based unbound public key derivation. Spending the coins addressed to one of its users, however, requires the creation of a certificate that associates their identities to their keys. For this reason, Property 2.2 and, more precisely, spending coins within Arcula, is bound to the generation of a certificate, signed by the master secret key, that authorizes the users. The deterministic generation of secret keys, instead, relies on an unbound hierarchical key assignment. This approach provides some significant advantages and enables a set of novel use cases: Users (*e.g.*, enterprises, or smart contracts in decentralized finance) rely on the identity-based derivation to generate in untrusted environments fresh public keys and addresses on which they will securely receive coins. The certificates will only be required before spending these coins, and their generation can happen entirely offline (*e.g.*, in cold storage).

Finally, Arcula differentiates itself from the other solutions by supporting a complex access hierarchy — *e.g.*, a directed acyclic graph (DAG) — instead of a tree. Many of our novel use cases require more complex hierarchies. Consider, for instance, two or more departments of a company that collaborate on a project. They share a common budget and need to spend from it without revealing their own secret key. The access hierarchy encoding this context would derive a single node, the budget, as a successor of multiple others (the departments). Equations (1) to (3) cannot handle hierarchies where a node has more than one predecessor—constraining BIP32 only to implement access hierarchy that forms a tree. The hierarchical key assignment scheme at the core of Arcula, instead, supports complex hierarchies and, in addition, it allows us to dynamically modify the hierarchy (*e.g.*, by deleting an intermediate node) and to incorporate temporal capabilities (*e.g.*, key expiration) into the wallet.

4 Preliminaries

4.1 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. 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 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 write $\mathsf{negl}(\lambda)$ to denote an unspecified negligible function in the security parameter.

4.2 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 4.1 (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:

$$\Pr[\mathsf{Vrfy}(\mathsf{pk}, m, \mathsf{Sign}(\mathsf{sk}, m))] = 1,$$

where $(\mathsf{sk}, \mathsf{pk}) \leftarrow_{\$} \mathsf{KGen}(1^\lambda)$.

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 4.2 (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.

5 Security Model of Hierarchical Deterministic Wallet

One of the contributions of this work is to formalize, for the first time, the security model of hierarchical deterministic wallets (HDW). We define a hierarchical deterministic wallet over 5 algorithms $(\mathsf{Set}, \mathsf{DPub}, \mathsf{DPriv}, \mathsf{Sign}, \mathsf{Vrfy})$, where: 1) Set deterministically instantiates the wallet by generating the public parameters pp and a set of the derivation key d_i, one for each node $v_i \in V$ of the hierarchy. 2) DPriv and DPub are responsible of the derivation of signing and public keys (Properties 2.2 and 2.3). DPriv derives the signing key sk_j of a node v_j, descendent of v_i, by using the derivation key d_i associated to v_i; DPub derives the corresponding public key pk_j by using the only the public parameters pp. 3) Sign and Vrfy take inspiration from the standard signing and verification algorithms of a digital signature scheme.

Concretely, let $G = (V, E)$ be a directed acyclic graph (DAG) representing an access hierarchy. We define the set of descendants $Desc(v_i) = \{v_j \mid v_i \rightsquigarrow_w v_j\}$ of node v_j to be the set of nodes v_j such that there exists a direct path w from v_i to v_j in G. A hierarchical deterministic wallet $\Pi = (\mathsf{Set}, \mathsf{DPub}, \mathsf{DPriv}, \mathsf{Sign}, \mathsf{Vrfy})$, defined over a seed space $\mathcal{S}$ and message space $\mathcal{M}$ is defined in the following way:

$\mathsf{Set}(1^\lambda, G, S)$**:** The deterministic setup algorithm takes as input a security parameter, an access graph $G = (V, E)$, and an initial seed $S \in \mathcal{S}$, and outputs the public parameters pp and a set of derivation keys $\{\mathsf{d}_i\}_{v_i \in V}$.

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

$\mathsf{DPriv}(\mathsf{pp}, \mathsf{d}_i, v_i, v_j)$**:** The deterministic private derivation algorithm takes as input the public parameters pp, the derivation key d_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. This means that 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 $G = (V, E)$, $\forall v_i, v_j \in V$, $\forall v_j \in Desc(v_i), \forall S \in \mathcal{S}, \forall m \in \mathcal{M}$ 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 $(\mathsf{pp}, \{\mathsf{d}_i\}_{v_i \in V}) = \mathsf{Set}(1^\lambda, G, S)$, $\mathsf{sk}_j = \mathsf{DPriv}(\mathsf{pp}, \mathsf{d}_i, v_i, v_j)$, and $\mathsf{pk}_j = \mathsf{DPub}(\mathsf{pp}, v_j)$.

The security of a hierarchical deterministic wallet draws inspiration from existentially unforgeable signatures. We allow an attacker to corrupt an arbitrary number of users in the hierarchy—by corrupting a user; the attacker implicitly corrupts also all her descendants. In addition, the attacker also has access to a signing oracle, that returns signatures on arbitrary messages from any uncorrupted node. We challenge the attacker to forge a signature for a new 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 $S \leftarrow_\$ \mathcal{S}$ and executes $(\mathsf{pp}, \{\mathsf{d}_i\}_{v_i \in V}) = \mathsf{Set}(1^\lambda, G, S)$. It gives the public parameters pp 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 d_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 $\mathsf{sk}_i = \mathsf{DPriv}(\mathsf{pp}, \mathsf{d}_0, v_0, v_i)$. 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 $\mathsf{pk}_i = \mathsf{DPub}(\mathsf{pp}, v_i)$ and $v_i \notin \mathcal{Q}_{\mathsf{Corr}}$, $(m, v_i) \notin \mathcal{Q}_{\mathsf{Sign}}$, return 1; otherwise return 0.

6 Arcula: A Secure Hierarchical Deterministic Wallet

Arcula, the design that we present in this paper, satisfies the properties of HDW formalized by Definitions 5.1 and 5.2. It is provably secure against key recovery; it deterministically derives the private information from an initial seed; and it enables identity-based public key derivation. It achieves such result by relying on two fundamental intuitions: Securely and deterministically generating a set of keys for the users of a hierarchy and then explicitly associating these keys with their identities. By doing so, Arcula provides a groundbreaking design that brings identity-based cryptography to the blockchain, tackles explicitly novel use cases and applications, and stands on more than 20 years of research in how to securely distribute keys to the users in a hierarchy.

At its core, indeed, Arcula derives the keys of the users by relying on a deterministic version of hierarchical key assignment schemes (HKA): A provably secure process that, precisely as it happens in an HDW, takes as input a hierarchy of users and assigns a secret key to every user, so that users with higher privileges can derive the keys of those with fewer privileges. To the best of our knowledge, hierarchical key assignment schemes have never been leveraged before implementing an HDW. As a result, one of the contributions of this work is to bind together, for the first time, these seemingly unrelated fields of research. In addition, hierarchical key assignment schemes provide several advantages: They're highly efficient and have been extensively studied in the past; they enable Arcula to implement arbitrarily complex hierarchies, to integrate temporal capabilities into the wallet, and to support the dynamic addition or removal of users to the hierarchy.

The following sections first provide a brief background on (deterministic) hierarchical key assignment schemes and then describe, in detail, our construction of Arcula.

6.1 Deterministic Hierarchical Key Assignment

A hierarchical key assignment scheme [3] assigns a set of cryptographic keys to a set of users in a hierarchy. The hierarchy, encoded as a directed acyclic graph, represents the access rights of users: A path from a node v_i to a node v_j implies that the user v_i has higher privileges than v_j and can assume the same access rights of v_j. An efficient hierarchical key assignment scheme (HKA) enforces the access hierarchy while minimizing the number of keys distributed to the users.

Typically, HKA schemes sample the cryptographic secrets that they assign at random. Our goal, however, is to leverage an HKA at the core of our wallet, where each secret is deterministically derived from a seed provided by the user. To do so, we propose a deterministic modification of the HKA developed by Atallah *et al.* [3] that is secure under key indistinguishability (and as a consequence guarantees Property 2.1 by design).

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

$\mathsf{Set}(1^\lambda, G, S)$**:** The deterministic setup algorithm takes as input the security parameter, a DAG $G = (V, E)$, and an initial seed $S \in \mathcal{S}$, 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 $\mathsf{Sec} : V \rightarrow \{0,1\}^\lambda \times \{0,1\}^\lambda$, 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 $v_j \in Desc(v_i)$.

Informally, the correctness of a DHKA scheme requires that every user must be able to derive, correctly, the secret key of any other user lower in the hierarchy. Its security definition, instead, requires that even if an attacker corrupts an arbitrary number of descendants of a node, he cannot distinguish its secret key from a uniformly random string. We formalize the security model and the construction of our DHKA in Appendix B.

6.2 Constructing Arcula from DHKA and signatures

Arcula, our implementation of a hierarchical deterministic wallet, relies on deterministic hierarchical key assignment schemes (DHKA) and digital signatures. This section details our construction, that we describe through the $(\mathsf{Set}, \mathsf{DPub}, \mathsf{DPriv}, \mathsf{Sign}, \mathsf{Vrfy})$ algorithms that define a hierarchical deterministic wallet. The Set algorithm instantiates the scheme, starting from the initial seed to deterministically generate a pair of *master public and secret* keys $(\mathsf{msk}, \mathsf{mpk})$. The master keys respectively serve two purposes: To identify the users of the wallet and to assign a pair of *signing* keys $(\overline{\mathsf{sk}}_i, \overline{\mathsf{pk}}_i)$ to each of them. We generate the signing keys through the key indistinguishable DHKA scheme of Section 6.1, so that users can sign transactions on behalf of their descendants while guaranteeing the security of their keys against privilege escalation. In more detail, the DHKA assigns to each user v_i a derivation key d_i that allows them to generate their own signing keys $\overline{\mathsf{sk}}_i$ and to derivate those of their descendants. We set the master secret key to the derivation key of the user v_0 with highest privileges in the hierarchy[1], *i.e.*, $\mathsf{msk} = \mathsf{d}_0$. The master public key mpk, on the other hand, unequivocally identifies the wallet. We set it to the public signing key of v_0 $(\mathsf{mpk} = \overline{\mathsf{pk}}_0)$, and we combine it with the identifiers of users to achieve public key derivation. In particular, we identify a user v_i of the wallet by concatenating the master public key mpk with the public label l_i associated with her. Finally, the Set algorithm binds the identifies of users to their signing key pair $(\overline{\mathsf{sk}}_i, \overline{\mathsf{pk}}_i)$ through a certificate, signed by the master secret key msk[2], that explicitly authorizes the signing key $\overline{\mathsf{sk}}_i$ to spend the coins destined to v_i. The DPriv and DPub algorithms respectively derive the signing keys (through the DHKA scheme) and the corresponding public keys (by combining the master public key mpk and the node identifiers). Finally, the Sign and the Vrfy algorithms handle the creation and verification of digital signatures. Any node v_i runs the Sign algorithm with its signing key $\overline{\mathsf{sk}}_i$ to create a signature and the Vrfy algorithm checks that there exists a certificate authorizing $\overline{\mathsf{sk}}_i$ to spend funds on behalf of the node v_i (identified by the label l_i) under the master secret key msk.

On the blockchain, the master public key mpk and the public label l_i of a user form her address for receiving payments. The users spend funds by signing transactions through their signing key $\overline{\mathsf{sk}}_i$ and by presenting, at the same time, the certificate that associates their keys to their identity and that authorizes them to spend funds. This approach provides several advantages. Receiving funds does not require the certificate—the identity-based public key derivation allows us to generate the address of any destination node without any private information. Users only

[1] If there exist multiple users with maximum privileges it is always possible to modify the structure of the DAG and add a *minimal* node that does not correspond to any real user, does not change the hierarchical ordering of the others, and has the highest privileges. We describe the modification in detail in Appendix B.

[2] The master secret key msk deterministically generates the (master) signing key $\overline{\mathsf{sk}}_0$, that we use to sign the certificates.

require their certificate when spending funds for the first time, *i.e.*, when signing a transaction through their secret key $\overline{\mathsf{sk}}_i$, and the creation of the certificate can happen entirely offline (*i.e.*, in cold storage). In addition, the signing keys $(\overline{\mathsf{sk}}_i, \overline{\mathsf{pk}}_i)$ can also provide users with unlinkability of their transactions. As we describe in Section 7.5, users that do not need the identity-based public key derivation can directly leverage the signing public key $\overline{\mathsf{pk}}_i$ as a pseudonym address on which to receive funds and then spend them through the corresponding private key $\overline{\mathsf{sk}}_i$.

Construction 1. 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:

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

1. Compute $(\mathsf{Pub}, \mathsf{Sec}) = \mathsf{Set}_\Gamma(1^\lambda, G, S)$.
2. For each node $v_i \in V$:
   (a) Let $(S_i, x_i) = \mathsf{Sec}(v_i)$ and set $\mathsf{d}_i = S_i$.
   (b) $(\overline{\mathsf{sk}}_i, \overline{\mathsf{pk}}_i) = \mathsf{KGen}_\Sigma(1^\lambda; x_i)$.
3. Output $\mathsf{pp} = (G, \mathsf{Pub}, \{\mathsf{cert}_i\}_{v_i \in V}, \overline{\mathsf{pk}}_0)$ and $\{\mathsf{d}_i\}_{v_i \in V}$ where $\mathsf{cert}_i \leftarrow\!\!\$\; \mathsf{Sign}_\Sigma(\overline{\mathsf{sk}}_0, (\overline{\mathsf{pk}}_i, l_i))$ for $v_i \in V$, and $l_i = \mathsf{Pub}(v_i)$.

$\mathsf{DPub}(\mathsf{pp}, v_j)$**:** On input the public parameters $\mathsf{pp} = (G, \mathsf{Pub}, \{\mathsf{cert}_i\}_{v_i \in V}, \overline{\mathsf{pk}}_0)$ and a node $v_j \in V$, the algorithm returns $\mathsf{pk}_j = (\overline{\mathsf{pk}}_0, l_j)$ where $l_j = \mathsf{Pub}(v_j)$.

$\mathsf{DPriv}(\mathsf{pp}, \mathsf{d}_i, v_i, v_j)$**:** On input the public parameters $\mathsf{pp} = (G, \mathsf{Pub}, \{\mathsf{cert}_i\}_{v_i \in V}, \overline{\mathsf{pk}}_0)$, the derivation key $\mathsf{d}_i = S_i$, and two nodes $v_i, v_j \in V$ such that $v_j \in \mathit{Desc}(v_i)$, the algorithm runs $x_j = \mathsf{Derive}_\Gamma(G, \mathsf{Pub}, v_i, v_j, S_i)$ and $(\overline{\mathsf{sk}}_j, \overline{\mathsf{pk}}_j) = \mathsf{KGen}_\Sigma(1^\lambda; x_j)$. Finally, it returns $\mathsf{sk}_j = (\overline{\mathsf{sk}}_j, \overline{\mathsf{pk}}_j, \mathsf{cert}_j)$.

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

$\mathsf{Vrfy}(\mathsf{pk}_i, m, \sigma)$**:** On input a public key $\mathsf{pk}_i = (\overline{\mathsf{pk}}_0, l_i)$, a message m, and a signature $\sigma = (\overline{\mathsf{pk}}_i, \sigma', \mathsf{cert}_i)$, the algorithms returns 1 if $\mathsf{Vrfy}_\Sigma(\overline{\mathsf{pk}}_0, (\overline{\mathsf{pk}}_i, l_i), \mathsf{cert}_i) = 1$ and $\mathsf{Vrfy}_\Sigma(\overline{\mathsf{pk}}_i, m, \sigma') = 1$; otherwise it returns 0.

Remark. *Arcula shares a significant number of similarities with identity-based hierarchical signature schemes [14]. Its design, indeed, associates a pair of signing keys to the identity of a user and allows her to sign messages on behalf of her descendants. We point out, however, some fundamental differences. Most hierarchical identity-based signature schemes leverage a conspicuous number of public parameters and rely on bilinear mappings. This makes them unpractical to use in the existing blockchains: The underlying protocols should efficiently handle the bilinear mappings, and the public parameters that define the instantiation of the scheme should be stored on the blockchain itself. Our design of Arcula, on the other hand, explicitly takes into consideration the characteristics and the limitations of blockchain systems: We do not rely on bilinear mappings, and we only store a small portion of the public parameters* pp *(one certificate per transaction, typically a single group element) on the blockchain.*

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 E.

Theorem 6.1. *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 B.2) and* Σ *is existentially unforgeable (Definition 4.2), then the HDW* Π *from Construction 1 is hierarchically existentially unforgeable (Definition 5.2).*

In the context of the security model that we defined in Section 2.3, Arcula's public derivation belongs to the **Untrusted Environment**—it merely relies on the concatenation of two public values, the master public key mpk and the identifier l_i of node v_i. Redeeming coins requires, instead, the **Hot Environment**. The certificate cert_i that associates v_i to its public key $\overline{\mathsf{pk}}_i$ is a public parameter of the wallet, but we require the node's corresponding private signing key $\overline{\mathsf{sk}}_i$ to sign a new transaction. Compromising the secrets of node v_i leads to compromising all its descendants, but none of the other nodes. Finally, the master secret key msk and the related signing key $\overline{\mathsf{sk}}_0$ must be safely kept in **Cold Storage**. We leverage these keys in the setup phase to prepare the authorization certificate cert_i of v_i and, for this reason, it is critical to the security of the wallet: An attacker can use it to forge a certificate that associates any pair of keys to any target node and spend, in turn, the coins held in the entire wallet.

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

1. Is secure against privilege escalation (Property 2.1).

2. Generates every cryptographic key from an initial seed (Property 2.2).

3. Enables identity-based public-key derivation so that users can dynamically derive new public keys without accessing their own private keys (see Property 2.3).

4. Enables secret-key derivation so that users can sign transactions 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 dynamic modifications to the hierarchy (*i.e.*, by adding or removing nodes, as we detail in Appendix C) and controlling the key assignment according to some temporal constraints (Appendix D).

7 Arcula in the real world

With Arcula, we design a future-proof HDW that brings identity-based signatures to the blockchain item and that, at the same time, is also suitable to the most widely used cryptosystems of today. We aim to join theory and practice, to create a wallet that fulfills our current and future needs in the crypto-coins space.

For this reason, 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 that associates the signing key to their corresponding user). This design makes it immediately compatible with the Ethereum blockchain, which 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, initially, it allowed).

In this section, we show how Arcula performs in the real world, and we show how to spend and receive funds, out of the box, on the Bitcoin Cash blockchain. Next, we discuss the modifications that would make it compatible with the original Bitcoin protocol and how it is possible to disable public derivation to obtain unlinkability of transactions.

7.1 Technical Implementation

Our open-source implementation of Arcula is available online.[3] We instantiate the underlying DHKA leveraged by Arcula with the pseudorandom function $\mathsf{F}_k(x) = \mathsf{H}(k \| x)$ (where $\mathsf{H}(x)$ is the hash function SHA3-256(x)) and the authenticated AES256 with Galois/Counter Mode (GCM) as the symmetric encryption scheme. We generate a hierarchal deterministic wallet based on the tree defined in BIP43 and BIP44 [18, 19], 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 of 512 bits by following the specification of BIP39 [20] 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 $\mathsf{H}($`correct horse battery staple`$)$.

7.2 Arcula in Bitcoin Cash

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 to redeem the coins being transferred. More in detail, 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 <H(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 master public key $\mathsf{mpk} = \overline{\mathsf{pk}}_0$ and to their public label l_i. For this reason, an Arcula address is simply the 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 $\mathsf{cert}_i \leftarrow\!\!\$\, \mathsf{Sign}_\Sigma(\overline{\mathsf{sk}}_0, (\overline{\mathsf{pk}}_i, l_i))$ and associating the signing public key $\overline{\mathsf{pk}}_i$ to the node v_i with label l_i, and a signature σ of the transaction verifiable through the public signing key $\overline{\mathsf{pk}}_i$. With Arcula, the locking and the unlocking scripts respectively become:

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

Table 2: The script bytes sizes of a transaction to a standard Bitcoin address and to an Arcula address.

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

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

Unlocking: <σ> <cert_i> <$\overline{\mathsf{pk}}_i$>

The two scripts: 1) Verify that the certificate cert_i is a valid signature of the message $(\overline{\mathsf{pk}}_i, l_i)$ under the master public key mpk; 2) verify the validity of the transaction signature σ under the signing public key $\overline{\mathsf{pk}}_i$. 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 mpk and l_i obtained through the operation `OP_CAT`). The scripting language of the original Bitcoin does not implement such an 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 cryptocurrency 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 7.1, identified through the master public key mpk (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 computed using the signing key $\overline{\mathsf{sk}}_i$, an appropriate certificate cert_i signed by the master secret key, and the public signing key $\overline{\mathsf{pk}}_i$. 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 l_i of the node v_i with 4 bytes.

7.3 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 2 compares the sizes, in bytes, of the locking 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

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

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 <mpk> OP_CHECKDATASIGVERIFY OP_FROMALTSTACK OP_CHECKSIG

Locking: OP_HASH160 <H(Script)> OP_EQUAL

Unlocking: <σ> <cert_i> <$\overline{\mathsf{pk}}_i$> <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. An e-commerce marketplace, as an example, can leverage Arcula's public key derivation 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 every 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 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 master public key mpk that identifies the company. Finally, many enterprises 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 master and signing public keys).

7.4 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 cert_i can be used more than once. For this reason, the first optimization that we propose is to *cache* the certificate cert_i as soon as it appears for the first time in an unlocking script. Then, any subsequent transaction signed by $\overline{\mathsf{sk}}_i$ 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 combine the certificates with the signatures that authorize transactions: Now, the certificate of user v_i that associates her to the signing public key $\overline{\mathsf{pk}}_i$ also includes an additional modifiable portion that will be filled with the transaction details. To spend their coins, the users leverage their sanitizable key to replace the blank transaction with the details that they intend to sign.[5] 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. This construction would allow Arcula to be used with the traditional Bitcoin blockchain by implementing the sanitizable signatures on top of the ECDSA signature scheme that it already uses. In addition, it would not change the expressiveness of the Bitcoin scripting language: Instead of enabling the verification of signatures on arbitrary messages, it would simply extend the signature verification protocol to account for the certificate embedded in the sanitized signatures.

7.5 Unlinkability of Transactions

Individual users of hierarchal deterministic wallets are typically not interested in public key derivation. Differently from enterprises and e-commerce marketplaces, for instance, they simply rely on HDW to recover their keys in case of hardware failure or catastrophic loss. On the other hand, they are often interested in achieving unlinkability of their transactions, *i.e.* in making sure that multiple transactions sent to their wallet can not be correlated together by an observer that passively monitors the blockchain. In other words, they typically desire to

[5] The sanitizable keys can be hierarchically deployed by leveraging a second instance of DHKA.

trade the derivation of public keys in an untrusted setting for the ability to receive payments on uncorrelated pseudonyms.

Arcula allows them to reach this goal. In more detail, these users can ignore the identity-based public key derivation that Arcula provides (and its associated master public key mpk) and identify the nodes of the wallet with their public signing key $\overline{\mathsf{pk}}_i$. On the blockchain, they can receive standard transactions (costing standard transaction fees) on the public signing key $\overline{\mathsf{pk}}_i$ and then sign new transactions to redeem the coins through the corresponding private key $\overline{\mathsf{sk}}_i$. We generate this pair of keys through a DHKA scheme that is secure under key indistinguishability. This means that every public signing key in the wallet is unlinkable from the others, because the DHKA cryptographic keys x_i that we use as randomness to generate them are, in turn, indistinguishable from random strings of the appropriate length. As a result, Arcula provides a provably secure alternative to the hardened mode of BIP32: Individual users can generate as many pseudonyms as they need by branching or deepening the DAG that encodes their hierarchy, and then leverage the DHKA to generate keys and reliably recover them in case of loss. Note that this modified version of Arcula does not require validation of signatures of arbitrary messages, enabling its usage with any blockchain system, including Bitcoin.

In addition, Arcula also allows users to achieve unlinkability of transactions while maintaining identity-based public key derivation. The intuition is to use a chain code c, private to the environment where we execute the public derivation, to *perturb* the master secret and public keys so that they look uncorrelated from the original keys to a passive observer. We perform the perturbation once for each node in the wallet. As a result, we associate them with a set of perturbed pairs of keys, labeled $(\mathsf{msk}_i, \mathsf{mpk}_i)$, that we use to sign the certificate cert_i that associates their public signing key $\overline{\mathsf{pk}}_i$ to their identity v_i. Now, we can address Bitcoin payments for the node v_i to the i-th perturbation mpk_i of the master public key mpk, that is uncorrelated from any other key of the wallet.

More in detail, let g be a generator of the elliptic curve used in Bitcoin's ECDSA signature scheme. The master public key is defined as $\mathsf{mpk} = \mathsf{mpk}_0 = \overline{\mathsf{pk}}_0 = g^{\overline{\mathsf{sk}}_0}$. Let c be the secret chain code and let F be a pseudorandom function. We create the i-th perturbed key mpk_i of the master public key mpk as follows:

$$\mathsf{mpk}_i = g^{\overline{\mathsf{sk}}_0 + \mathsf{F}_c(l_i)} = g^{\overline{\mathsf{sk}}_0} \cdot g^{\mathsf{F}_c(l_i)} = \mathsf{mpk} \cdot g^{\mathsf{F}_c(l_i)},$$

where l_i is the label of node v_i. As long as the chain code c is private, this construction ensures the unlinkability of transactions sent to the perturbed addresses [17].

In Arcula, we modify the Item 3 of Construction 1 to sign the certificates $\mathsf{cert}_i \leftarrow\$ \mathsf{Sign}_\Sigma(\mathsf{msk}_i, \overline{\mathsf{pk}}_i)$ with the perturbed secret key $\mathsf{msk}_i = \overline{\mathsf{sk}}_0 + \mathsf{F}_c(l_i)$. Note that we remove the label l_i from the certificate since now every pair of perturbed keys is uniquely associated with 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 master public key mpk in the locking script with the i-th perturbed key mpk_i, that verifies the certificate cert_i, as follows:

Locking: OP_DUP OP_TOALTSTACK <mpk_i> OP_CHECKDATASIGVERIFY
OP_FROMALTSTACK OP_CHECKSIG

Unlocking: <σ> <cert_i> <$\overline{\mathsf{pk}}_i$>

As a result, all the Arcula addresses of the same wallet look uncorrelated when they appear in the locking script of a transaction. The perturbed private keys msk_i are effectively equivalent to the original master secret key: An attacker that compromises any perturbed key can recover the master secret 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 cold storage, or better yet, destroyed after the generation of the corresponding certificate.

To conclude, we briefly discuss how to derive, deterministically, the chain code c. We propose to assign a different chain code c_i to each node v_i of the wallet by running our DHKA a second time: In this way, 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.

8 Conclusions

In this work, we presented Arcula, a new hierarchical deterministic wallet (HDW) that brings identity-based signatures to the blockchain, and that is secure against privilege escalation. We first 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 other users higher in the hierarchy. In addition, our wallet allows us to dynamically derive new addresses 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 legacy and future requirements of modern blockchains. In particular, Arcula is independent of the underlying signature scheme, and it works on top of any protocol 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 propose a future-proof design that supports the financial applications and tools of enterprises and companies at scale.

References

[1] Gavik Andresen. BIP16: Pay to script hash, 2012. URL `https://github.com/bitcoin/bips/blob/master/bip-0016.mediawiki`. Last visited December 10, 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 December 10, 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] Poulami Das, Sebastian Faust, and Julian Loss. A formal treatment of deterministic wallets. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, 2019.

[10] 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.

[11] 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/.

[12] 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.

[13] 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.

[14] Craig Gentry and Alice Silverberg. Hierarchical id-based cryptography. In Yuliang Zheng, editor, *Advances in Cryptology — ASIACRYPT 2002*, pages 548–566, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. ISBN 978-3-540-36178-7.

[15] 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.

[16] 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`.

[17] Gregory Maxwell et al. Deterministic wallets, 2011.

[18] 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 December 10, 2019.

[19] 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 December 10, 2019.

[20] 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 December 10, 2019.

[21] Pieter Wuille. BIP32: Hierarchical deterministic wallets, 2012. URL `https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki`. Last visited December 10, 2019.

A Further Preliminaries

A.1 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 A.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 $\mathcal{K}_\lambda = \mathcal{Y}_\lambda = \{0,1\}^\lambda$.

A.2 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 A.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 A.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:

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

where $\mathbf{G}^{\mathsf{sem}}_{\Pi,\mathsf{A}}(\lambda)$ 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.

B Security Model of Deterministic Hierarchical Key Assignment

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

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

where $(\mathsf{Pub}, \mathsf{Sec}) = \mathsf{Set}(1^\lambda, G, S)$, $(S_i, x_i) = \mathsf{Sec}(v_i)$, 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. We define the set of ancestors $Anc(v_i) = \{v_j \mid v_j \rightsquigarrow_w v_i\}$ of a node v_i to be the set of nodes v_j such that there exists a path w from v_j to v_i in G.

Definition B.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 $S \leftarrow\!\!\$\, \mathcal{S}$, then runs $\mathsf{Set}(1^\lambda, G, S)$, 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.*

B.1 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[6] (*i.e.*, $l_i = v_i$) and we derive its secret information S_i deterministically (through a pseudorandom function) from the secret information of its parent. We formally define our implementation of DHKA as follows.

Construction 2. 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 $S_0 = \mathsf{F}_S(11\|l_0)$ 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 $y_{ij} \leftarrow\!\!\$\ \mathsf{Enc}_{r_{ij}}(t_j\|x_j)$.[7]

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 2. 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).

[6] 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 Appendix C for more details.

[7] 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.

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 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, *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 a 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.

Theorem B.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 A.1) and $\mathcal{E}$ is semantically secure (Definition A.3), then the DKHA scheme Π from Construction 2 is key indistinguishable.*

Proof. 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.

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

$\mathbf{G}_i^{(c)}$**:** is the same as $\mathbf{G}_i^{(b)}$, except that y_{ij} associated to the edge (v_i, v_j) (where $v_i, v_j \in \mathit{Anc}(v^*)$) 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 $\mathbf{G}_i^{(a)}$ (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 B.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 $\mathbf{G}_{\mathsf{F,A}}^{\mathsf{prf}-0}(\lambda)$ and $\mathbf{G}_{\mathsf{F,A}}^{\mathsf{prf}-1}(\lambda)$ 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 2. 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 2. A sets $x_{v^*}^0 = x_{v^*}$ and $x_{v^*}^1 = \bar{x}_{v^*}$ where $\bar{x}_{v^*}$ is sampled at random. Finally, A sends Pub and $x_{v^*}^d$ to D where d is a random bit.
3. A answers any $\mathsf{O}_{\mathsf{Corr}}^{\Pi}(v_i)$ 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 $\mathbf{G}_{\mathsf{F,A}}^{\mathsf{prf}-0}(\lambda)$ and $\mathbf{G}_{\mathsf{F,A}}^{\mathsf{prf}-1}(\lambda)$, then the reduction perfectly simulates $\mathbf{G}_{-1}$ and $\mathbf{G}_0$. Indeed, if A is playing with $\mathbf{G}_{\mathsf{F,A}}^{\mathsf{prf}-0}(\lambda)$ (resp. $\mathbf{G}_{\mathsf{F,A}}^{\mathsf{prf}-1}(\lambda)$) 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 2. As such, the advantage of the attacker A in distinguishing $\mathbf{G}_{\mathsf{F,A}}^{\mathsf{prf}-0}(\lambda)$ and $\mathbf{G}_{\mathsf{F,A}}^{\mathsf{prf}-1}(\lambda)$ is non negligible. This concludes the proof. □

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

C 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 Appendix B and, in turn, within Arcula, our hierarchical deterministic wallet of Section 6.

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 2 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 with 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 master public key mpk 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 master secret key, that associates the new public key $\overline{\mathsf{pk}}'_j$ (obtained from the new intermediate key x'_h after the *rekey* procedure) to the new (or updated) label l_j.

D 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 use 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 Appendix B and within Arcula, our design hierarchical deterministic wallet (Section 6).

In the last few years, many researchers focused on how to incorporate temporal capabilities into HKA schemes [5, 2, 10]. 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.* [10] 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.[8] 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) 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 Appendix B.1, 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 with 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 master secret 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 $\mathsf{cert}_{i,j}$ for each key. We prevent the

[8] In [10] the subsequence of time periods of a node $v_i \in V$ is denoted by λ_i.

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 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.

E Proof of Theorem 6.1

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 $(\overline{\mathsf{sk}}_i, \overline{\mathsf{pk}}_i)$ 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 $(\overline{\mathsf{sk}}_i, \overline{\mathsf{pk}}_i)$ 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 E.1. *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 $\mathbf{G}^{\mathsf{sk-ind}}_{\Gamma,\mathsf{A}}(\lambda, G)$ (defined in Definition B.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 executes the remaining steps of Set_Π, 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 $(\overline{\mathsf{sk}}_i, \overline{\mathsf{pk}}_i) \leftarrow\!\!\$\, \mathsf{KGen}_\Sigma(1^\lambda)$.
   - Otherwise, if $v_i = v_t$, then compute $(\overline{\mathsf{sk}}_t, \overline{\mathsf{pk}}_t) = \mathsf{KGen}_\Sigma(1^\lambda; x_t)$.
   - Otherwise, send a $\mathsf{O}^{\Gamma}_{\mathsf{Corr}}(v_i)$ query to the challenger and receive $S_i = \mathsf{d}_i$. Compute $x_i = \mathsf{Derive}_\Gamma(G, \mathsf{Pub}, v_i, v_i, S_i)$ and $(\overline{\mathsf{sk}}_i, \overline{\mathsf{pk}}_i) = \mathsf{KGen}_\Sigma(1^\lambda; x_i)$.

   Finally, A outputs the public parameters $\mathsf{pp} = (G, \mathsf{Pub}, \{\mathsf{cert}_i\}_{v_i \in V}, \overline{\mathsf{pk}}_0)$.

3. 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 $\sigma = (\overline{\mathsf{pk}}_i, \sigma', \mathsf{cert}_i)$ where $\sigma' \leftarrow\!\!\$\, \mathsf{Sign}_\Sigma(\overline{\mathsf{sk}}_i, m)$.

4. 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 $\mathsf{pk}_j = (\overline{\mathsf{pk}}_0, l_j)$.

5. 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 $v \in \mathit{Anc}(v^*)$. 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 E.2. *If Σ is existentially unforgeable, then for every DAG $G = (V, E)$ and PPT adversary A,* $\Pr\left[\mathbf{G}_{|\mathit{Anc}(v_j)|,\mathsf{A}}(\lambda, G) = 1\right] \leq \mathsf{negl}(\lambda)$.

Proof. We assume that there exists a DAG $G = (V, E)$ and an adversary A that has a non-negligible advantage against $\mathbf{G}_{|\mathit{Anc}(v_j)|,\mathsf{A}}(1^\lambda, G)$. 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 $v^* \leftarrow\!\!\$\, V$ and $S \leftarrow\!\!\$\, \mathcal{S}$.

3. A' simulates Set_Π. It runs $(\mathsf{Pub}, \mathsf{Sec}) = \mathsf{Set}_\Gamma(1^\lambda, G, S)$. If $d = 0$, it sets $\overline{\mathsf{pk}}_0 = \mathsf{pk}^*$; otherwise it runs $(\overline{\mathsf{sk}}_0, \overline{\mathsf{pk}}_0) = \mathsf{KGen}_\Sigma(1^\lambda; x_0)$ where $(S_0, x_0) = \mathsf{Sec}(v_0)$. Lastly, A' executes the remaining steps of Set_Π, except that it replaces Item 2b and Item 3 with the following:

   Item 2b: A' proceeds as follow:

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

   Item 3: If $d = 1$, then retrieve the label $l_i = \mathsf{Pub}(v_i)$ and compute $\mathsf{cert}_i \leftarrow\!\!\$\, \mathsf{Sign}_\Sigma(\overline{\mathsf{sk}}_0, (\overline{\mathsf{pk}}_i, l_i))$; otherwise, set $\mathsf{cert}_i \leftarrow\!\!\$\, \mathsf{O}^{\Sigma}_{\mathsf{Sign}}((\overline{\mathsf{pk}}_i, l_i))$.

   Finally, A' sends to A the public parameters $\mathsf{pp} = (G, \mathsf{Pub}, \{\mathsf{cert}_i\}_{v_i \in V}, \overline{\mathsf{pk}}_0)$.

4. A' answers oracle queries in the following way:

   - On input v_i for $\mathsf{O}^{\Pi}_{\mathsf{Corr}}$, A' returns $\mathsf{d}_i = S_i$ 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 $\sigma' \leftarrow\!\!\$\, \mathsf{Sign}_\Sigma(\overline{\mathsf{sk}}_i, m)$. Lastly, it returns $\sigma = (\overline{\mathsf{pk}}_i, \sigma', \mathsf{cert}_i)$.

5. A' receives the forgery $(v_j, \widetilde{m}, \widetilde{\sigma})$ such that $\widetilde{\sigma} = (\mathsf{pk}^\bullet_j, \sigma^\bullet, \mathsf{cert}^\bullet_j)$ and aborts the simulation if $v^* \neq v_j \vee (d = 0 \wedge \mathsf{pk}^\bullet_j = \overline{\mathsf{pk}}_j) \vee (d = 1 \wedge \mathsf{pk}^\bullet_j \neq \overline{\mathsf{pk}}_j)$. Otherwise, if $d = 0$, it sends the forgery $((\mathsf{pk}^\bullet_j, l_j), \mathsf{cert}^\bullet_j)$ to challenger where $l_j = \mathsf{Pub}(v_j)$; if $d = 1$ sends $(\widetilde{m}, \sigma^\bullet)$.

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:

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

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

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

Let $\mathcal{Q}_{\mathsf{Sign}}^{\Sigma}$ and $\mathcal{Q}_{\mathsf{Sign}}^{\Pi}$ be respectively the set of queries submitted by A' to $\mathsf{O}_{\mathsf{Sign}}^{\Sigma}$ and the set of queries submitted by A to $\mathsf{O}_{\mathsf{Sign}}^{\Pi}$. Conditioned on $\neg E_{\mathsf{abort}}$ and since A is a valid adversary for $\mathbf{G}_{\Pi,\mathsf{A}}^{\mathsf{heuf}}(\lambda, G)$, then, with non-negligible probability, $\mathsf{Vrfy}_{\Pi}(\mathsf{pk}_j, \widetilde{m}, \widetilde{\sigma}) = 1$ if and only if $\mathsf{Vrfy}_{\Sigma}(\mathsf{wpk}, (\mathsf{pk}_j^\bullet, l_j), \mathsf{cert}_j^\bullet) = 1$ and $\mathsf{Vrfy}_{\Sigma}(\mathsf{pk}_j^\bullet, \widetilde{m}, \sigma^\bullet) = 1$, where $\mathsf{pk}_j = (\overline{\mathsf{pk}}_0, l_j)$ and $l_j = \mathsf{Pub}(v_j)$. Note that A' outputs a valid forgery for $\mathbf{G}_{\Sigma,\mathsf{A}'}^{\mathsf{euf}}(\lambda)$ with probability $\frac{1}{2}$:

1. Whenever $d = 0$, we have $\overline{\mathsf{pk}}_0 = \mathsf{pk}^*$ and $\mathsf{pk}_j^\bullet \neq \overline{\mathsf{pk}}_j$. This allows us to conclude that A' never asked $(\mathsf{pk}_j^\bullet, l_j)$ to oracle $\mathsf{O}_{\mathsf{Sign}}^{\Sigma}$ (*I.e.*, $(\mathsf{pk}_j^\bullet, l_j) \notin \mathcal{Q}_{\mathsf{Sign}}^{\Sigma}$). Hence, $((\mathsf{pk}_j^\bullet, l_j), \mathsf{cert}_j^\bullet)$ is a valid forgery for $\mathbf{G}_{\Sigma,\mathsf{A}'}^{\mathsf{euf}}(\lambda)$.

2. On the other hand, if $d = 1$, we have $\mathsf{pk}_j^\bullet = \overline{\mathsf{pk}}_j = \mathsf{pk}^*$. Since, A is a valid adversary it must produces a valid signature for a new fresh message. Hence, we can conclude that $(v^*, \widetilde{m}) \notin \mathcal{Q}_{\mathsf{Sign}}^{\Pi}$ and $(\widetilde{m}, \sigma^\bullet)$ is a valid forgery for $\mathbf{G}_{\Sigma,\mathsf{A}'}^{\mathsf{euf}}(\lambda)$.

This concludes the proof. □

By combining Lemma E.1 and Lemma E.2 we have that Construction 1 is hierarchically existentially unforgeable.