Towards a Smart Contract-based, Decentralized, Public-Key Infrastructure*

Christos Patsonakis[1], Katerina Samari†[1], Mema Roussopoulos[1], and Aggelos Kiayias†[2]

[1]National and Kapodistrian University of Athens, Greece ,
{c.patswnakis,ksamari,mema}@di.uoa.gr
[2]University of Edinburgh and IOHK, UK ,
Aggelos.Kiayias@ed.ac.uk

Abstract

Public-key infrastructures (PKIs) are an integral part of the security foundations of digital communications. Their widespread deployment has allowed the growth of important applications, such as, internet banking and e-commerce. Centralized PKIs (CPKIs) rely on a hierarchy of trusted Certification Authorities (CAs) for issuing, distributing and managing the status of *digital certificates*, i.e., unforgeable data structures that attest to the authenticity of an entity's public key. Unfortunately, CPKIs have many downsides in terms of security and fault tolerance and there have been numerous security incidents throughout the years. Decentralized PKIs (DPKIs) were proposed to deal with these issues as they rely on multiple, independent nodes. Nevertheless, decentralization raises other concerns such as what are the incentives for the participating nodes to ensure the service's availability.

In our work, we leverage the scalability, as well as, the built-in incentive mechanism of blockchain systems and propose a smart contract-based DPKI. The main barrier in realizing a smart contract-based DPKI is the size of the contract's state which, being its most expensive resource to access, should be minimized for a construction to be viable. We resolve this problem by proposing and using in our DPKI a *public-state* cryptographic accumulator with constant size, a cryptographic tool which may be of independent interest in the context of blockchain protocols. We also are the first to formalize the DPKI design problem in the Universal Composability (UC) framework and formally prove the security of our construction under the strong RSA assumption in the Random Oracle model and the existence of an ideal smart contract functionality.

1 Introduction

Public key, or asymmetric, cryptography is a critical building block for securing important communications across the Internet, such as, e-commerce and internet banking. To enable

*This is the full version of the paper [42].

†This research was supported by the ERC project CODAMODA and IOHK.

such applications, public-key infrastructures (PKIs) are essential because they provide a verifiable mapping from an entity's name to its corresponding public key. In essence, a PKI is a system that allows the creation, revocation, storage, and distribution of *digital certificates*, i.e., unforgeable data structures that attest to the authenticity of an entity's public key.

In a centralized PKI (CPKI), a Certification Authority (CA), is responsible for issuing, distributing and managing the status of digital certificates. Two assumptions must be made when deploying a CPKI. These are: 1) everyone knows the CA's (correct) public key and, 2) statements signed by the CA's private key are valid, i.e., everyone trusts the CA. In a CPKI, registration is handled in two phases. In the first phase, the user proves her claim on an identity to a Registration Authority (RA). Assuming the RA validates the claim, it forwards the user's request to the CA. In the second phase, the user receives her digital certificate, which is signed by the CA's private key, thus, attesting its validity. CAs *periodically* publish signed data structures that contain revoked certificates, e.g., a certificate revocation list (CRL). Distribution of certificate-related information is handled either by the CA (online CA), or, it is delegated to online, publicly accessible directories (offline CA).

While predominant in use, CPKIs have several shortcomings. A CA constitutes a single point of failure, both in terms of security and availability. There have been several incidents where CAs have been hacked that led to the issuance of false certificates for domains of high-profile corporations, such as Google ([3]). Other prominent examples are the Symantec ([4]) and TrustWave ([8]) incidents, as well as the growing concern of governments and private organizations being able to issue false certificates for surveillance, thus, violating the privacy of end-users ([50]). In practice, there exist multiple CAs, which are linked with well-defined, parent-child relationships, based on trust and other policies. The most notable example of this architecture is the SSL/TLS certificate chain. This hierarchical, tree-like, certification model is designed to increase the system's scalability and fault-tolerance. However, root, or even, subordinate CA compromises are still catastrophic ([26]).

In a decentralized PKI (DPKI), multiple, independent nodes cooperate and deliver the same set of services, without relying on one, or more, trusted third parties (TTPs). DPKIs have been proposed because, as distributed systems, they have the potential to offer a number of desirable properties that CPKIs cannot offer, such as scalability, fault-tolerance, load balancing and availability. Researchers have proposed DPKIs based on various distributed primitives, such as distributed hash tables (DHTs) (e.g., [9]). To account for malicious nodes and provide increased security, they employ secret sharing, threshold and byzantine agreement protocols (e.g., [11, 23]). These techniques, while more complex to design and implement correctly, lead to systems that do not exhibit single points of failure. Unfortunately, prior DPKIs do not provide incentives for the participating nodes to ensure that the offered service remains available in the long term, e.g., they fail to address the *free-riding* problem ([33]).

Blockchain protocols (e.g., Bitcoin [39]), feature a reward mechanism that incentivizes parties to engage in the protocol. The rewards come in the form of a digital currency that compensates its participants, thus, creating a counter-incentive to free-riding, while still retaining a highly scalable, free-entry system.

In this work, we present the design of a DPKI on top of a smart contract platform, a

new generation of blockchains that allow the development of smart contracts, i.e., stateful agents that "live" in the blockchain and can execute arbitrary state transition functions. The main barrier in realizing a smart contract-based DPKI is the size of the smart contract's state which, being its most expensive resource to access, should be minimized for a construction to be considered viable. Previous blockchain-based solutions, such as Namecoin ([6]) and Emercoin ([2]), fall short on this part as their state is linear to the number of registered entities. Fromknecht et al. [27] improve on this by harnessing the power of cryptographic accumulators, i.e., space-efficient data structures that allow for (non) membership queries. However, we believe that they do not exploit, sufficiently, their potential for the following reasons: 1) their system's state is still of logarithmic complexity, due to the use of a Merkle tree-based accumulator and, 2) their construction recomputes accumulator values to handle deletions of elements, i.e., each deletion (revocation) has a linear computational complexity. We resolve these inefficiencies by presenting a construction whose state is constant and avoids recomputing accumulator values. Our main building block is a public-state, additive, universal accumulator, based on the strong RSA assumption in the Random Oracle model, which, among others, has the following nice properties: 1) the accumulator and the structures for proving (non) membership (referred to as *witnesses*) have constant size and, 2) all of its operations can be performed efficiently by having access only to the accumulator's public key.

In short, the contributions of this paper are as follows:

- We propose the design of a DPKI on top of a smart contract platform. Due to the interoperability of smart contracts, our system provides a generic mechanism for on-blockchain authentication that, up to this point, was handled in an ad-hoc manner. Furthermore, the programmable nature of these platforms allows us to evolve our system with more efficient primitives, when such become available, without the need for a fork in the blockchain, which is the case for specialized PKI blockchains.

- We resolve the main barrier of realizing a viable smart contract-based DPKI by providing a construction that has the "constant-ness" property, i.e., both the smart contract's state, as well as, the structures for proving (non) membership, have constant size. We stress the importance of this property as it guarantees, in addition to efficiency, uniform digital currency costs for any given operation across all users, i.e., fairness in terms of costs.

- Our construction is based on a public-state, additive, universal accumulator, a cryptographic tool which may be of independent interest for protocols that employ blockchains for verifying, efficiently, the validity of information.

- We are the first to formalize the DPKI design problem in the Universal Composability (UC) framework ([19]) and we formally prove the security of our construction under the strong RSA assumption in the Random Oracle model and the existence of an ideal smart contract functionality.

- Even though our envisioned application is a PKI, we specifically model our service as a generic "Naming Service". Thus, our design can be ported to implement, efficiently,

other services that reside in this paradigm, e.g., a distributed domain name system (DDNS).

2 Related Work

Several previously proposed systems utilize the same underlying primitive, each in its own unique way, to decentralize the services of a PKI. In the interest of space, we focus on full-fledged DPKIs, i.e., systems that implement registration, revocation, certificate storage and retrieval. Thus, we will not be concerned with certification systems (e.g., [35]), which do not offer revocation, hybrid approaches, e.g., coupling CAs with structured overlays (e.g., [48]), or, even PGP ([52]), whose operation relies on centralized servers. We also review related work regarding cryptographic accumulators, which form the basis of our construction.

Researchers have proposed DPKIs based on the replicated state machine (RSM) paradigm ([51, 43]) to enforce a global, consistent view of the system's state. This is achieved by having nodes participate in an authenticated agreement protocol and typically assume: 1) a threshold t of faulty nodes, 2) *join()* and *leave()* protocols for nodes wishing to enter, or leave, a replica group, to adjust the system's threshold parameter and, 3) nodes are able to authenticate any (potential) participant. In RSM-based PKIs, registration requires one to perform an "out-of-band" negotiation with multiple administrative domains, which is cumbersome for the user. In addition, non-determinism, e.g., time-stamping, is a key difficulty of consistent replication since it can lead to replica state-divergence, thus, compromising fault-tolerance. However, time-stamping is essential in a PKI for tracking certificate lifetime. Blockchain-based systems, on the other hand, do not suffer from this issue and they have already been used for the implementation of time-stamping services (e.g., [31]). Furthermore, they employ a different form of agreement which is based on computation. This, alternative, agreement algorithm has the nice property of being adaptable as nodes freely join and leave the system. Experience has illustrated, that the blockchain approach has been highly favored by both the research community, as well as the industry, due to its highly scalable, adaptive and non-restrictive nature ([1, 5, 7]).

Structured overlays have also been proposed to distribute the services of a PKI ([11, 23]). These are, by design, scalable, load-balanced and provide for efficient storage and retrieval of data. Unfortunately, these systems do not defend against Sybil attacks. Douceur ([25]) has proved that to defend against the Sybil attack, distributed systems must employ either authentication, or, computational power. However, the aforementioned DHT-based systems do not employ either, thus, they are insecure. Blockchains, on the contrary, are resilient to the Sybil attack since their operation inherently depends on computational power.

In all of the above systems, nodes are expected to participate in resource-intensive protocols. Unfortunately, these systems do not incentivize node participation, nor enforce correct behavior of participating nodes.

The initial approach of constructing a blockchain-based PKI is based on the observation that there is an inherent similarity between the services of a DNS and a PKI, respectively. Both essentially map identity names to some value (be it an IP address, or, a public key). One of the biggest *altcoins*, Namecoin ([6]), provides a distributed DNS as its main function. In Namecoin, the blockchain is used both for storing, as well as, verifying/querying DNS records. Several DPKIs follow this approach, the most notable example of which is Emercoin

([2]). Unfortunately, this approach is inefficient as it forces each user to store an entire copy of the blockchain and traverse its contents every time she needs to validate a mapping. This limits the system's applicability significantly; for example, storing the entire blockchain on a smartphone is prohibitive. Moreover, validating mappings, which is the most frequent operation, requires an increasing amount of computation as more blocks are appended to the blockchain.

A modern, more involved approach, is to employ cryptographic accumulators, which were first introduced in the work of Benaloh et al. [15] as a decentralized alternative to digital signatures. These are space-efficient data structures that allow for membership queries. Their initial construction was refined in the work of Bari et al. [14] by strengthening the original security notion to that of *collision freeness*. Camenisch et al. [18] extended previous works and presented the first accumulator scheme that allowed for elements to be dynamically added/deleted, based on the strong-RSA assumption. In this scheme, membership witnesses can be updated by utilizing only the accumulator's public key, i.e., no trapdoor information is required. Following this work, various constructions have been suggested in the literature with different features and based on different assumptions. Li et al. [36] introduced the notion of *universal accumulators*, i.e., accumulators that support both membership and non-membership queries and proposed an RSA-based construction for a universal accumulator. Other proposed accumulator schemes are based on Merkle trees (e.g., [41], [16], [44]), bilinear pairings (e.g., [40], [10], [17], [22]) and lattices (e.g., [32], [49]). A detailed classification of existing accumulator schemes can be found in [24], where a unified formal model for cryptographic accumulators is suggested.

Cryptographic accumulators provide a number of benefits. First, their compact (or even constant) size makes them suitable candidates for storage-limited devices (e.g., smartphones). Second, most (non) membership verifications have constant computational cost, regardless of the number of accumulated values (accumulation is the addition of an element to the accumulator). Third, their security properties are based on standard hardness assumptions, thus, making them suitable for critical security infrastructures. An additional benefit of accumulator-based constructions is that they do not employ the blockchain for enforcing consensus on the entire set of (identity,public-key) mappings, as is the case for Namecoin and Emercoin. Instead, the consensus object is the accumulator value(s), which has the following benefits. First, users are not required to perform a complete retrieval and verification of the entire transaction history, i.e., downloading and validating the entire blockchain. Instead, an outdated, or, new user, can download and validate only block headers to update her state, which is far more efficient both in terms of communication and computation. Second, it allows the introduction of an unreliable component that users can query to efficiently obtain, among others, a more compact version of the entire history of operations, compared to the full transaction history. Due to the verifiable nature of cryptographic accumulators, this increased efficiency comes at no cost.

Certcoin ([27]) is a blockchain-based PKI which deals with the aforementioned inefficiencies of Namecoin and Emercoin by decoupling information storage from its verification. It employs an authenticated DHT for storing digital certificates, based on Kademlia ([38]). These networks facilitate storage and retrieval queries in logarithmic complexity, i.e., they are very efficient. Furthermore, its authenticated nature makes it secure against the Sybil attack ([25]). To facilitate the verification of (identity,public-key) mappings, Certcoin

maintains two cryptographic accumulators in the blockchain. Certcoin's first accumulator is based on the strong RSA assumption and accumulates identity names. Thus, users can infer if an identity has been registered in the system. The second accumulator is based on Merkle trees (formally presented in [44]) and accumulates (identity,public-key) mappings. This allows clients to validate the authenticity of any mapping retrieved from the DHT network by downloading and validating the latest block and by performing the appropriate membership queries.

In the following, we highlight the differences of our design compared to Certcoin. We open the discussion with issues regarding accumulators. First, our design employs two RSA-based accumulators, which have constant size and small public-keys. On the contrary, Certcoin employs one RSA and one Merkle-based accumulator. The size of Merkle-based accumulators increases as more elements are accumulated, i.e., it is not constant. This is an issue in the blockchain world as miners prefer small blocks which can be hashed faster and with reduced operational costs to increase their profits. Therefore, it would be difficult to incentivize miners to support Certcoin's blockchain whose blocks are of variable size. Moreover, in blockchain-based systems, transaction execution costs are a function of their size. Thus, Certcoin does not guarantee *fairness* in terms of transaction costs. Our construction does not face these issues due to its constant state. Second, while Merkle-based accumulators have the nice property that elements can be deleted without the knowledge of trapdoor information, the same does not hold for RSA-based accumulators. Consequently, when an (identity,public-key) mapping is revoked, thus making the identity available again, Certcoin recomputes its RSA accumulator from scratch, which is, inefficient. To deal with the fact that deletions in RSA accumulators require access to their secret key, which, if publicly known, can break their security, we employ a trick that is presented in the work of Baldimtsi et al. [13]. Essentially, we use tags to mark elements as "added" (during registration) or "deleted" (during revocation). Thus, in contrast to Certcoin, we do not recompute any accumulator. Third, Certcoin tightly couples the process of mining with the blockchain's actual application, which is to deliver the services of a DPKI. These two issues are orthogonal to each other, in terms of the system's architecture and, we believe, should be addressed at different layers. Instead, we are the first to propose a DPKI that is based on a smart contract platform, i.e., programmable blockchains that decouple the blockchain's consensus protocol from the applications' functionalities that run on top of it. This key difference allows us to evolve our system with more efficient primitives, when such become available, without the need for a hard fork in the blockchain, which is not the case for application-specific blockchains, such as Certcoin. Additionally, in these platforms, contracts can interact with each other, thus, creating an ecosystem of applications that can interoperate. By leveraging this feature, our system can provide a generic mechanism for on-blockchain authentication that, up to this point, was handled in an ad-hoc manner. Fourth, Certcoin has no security model for the PKI it implements nor a proof that it provides the claimed service. In contrast, we formalize the DPKI design problem in the UC framework ([19]) and we formally prove the security of our construction under the strong-RSA assumption in the Random Oracle model. Canetti [20], provides a minimal formulation of an ideal certification authority functionality which supports registration and retrieval of public keys. Our formulation is more involved and allows for more operations, such as, revocation of public keys.

3 Preliminaries

Notation. We use λ to denote the security parameter and $\mathsf{negl}(\cdot)$ to denote a function negligible in some parameter.

Definition 3.1 (Strong-RSA Assumption [14])**.** *For any p.p.t adversary $\mathcal{A}$,*

$$\mathsf{Pr}[n \leftarrow \mathsf{KeyGen}(1^\lambda); x \leftarrow \mathbb{Z}_n^*; (y, e) \leftarrow \mathcal{A}(n, x) : y^e = x \mod n] = \mathsf{negl}(\lambda),$$

where, $n = pq$, p and q are safe primes.

Definition 3.2 (2-Universal Hash Function Family [21])**.** *Let $U = \{f | f : X \to Y\}$ be a family of functions. We say that U is a 2-Universal Hash Function Family if, for all $x_1, x_2 \in X$ with $x_1 \neq x_2$ and for all $y_1, y_2 \in Y$, $\mathsf{Pr}_{f \in U}[f(x_1) = y_1 \wedge f(x_2) = y_2] = (\frac{1}{|Y|})^2$.*

Definition 3.3 (Pseudorandom Generator)**.** *Let $G : \{0,1\}^k \to \{0,1\}^{p(k)}$ be a deterministic polynomial time algorithm and $p(\cdot)$ a polynomial in some parameter k. We say that G is a pseudorandom generator if, for any p.p.t. algorithm D,*

$$\left|\mathsf{Pr}[D(r) = 1] - \mathsf{Pr}[D(G(s)) = 1]\right| \leq \mathsf{negl}(k),$$

where, r is a string chosen uniformly at random from $\{0,1\}^{p(k)}$ and s, the seed, is chosen uniformly at random from $\{0,1\}^k$.

4 Public-State Accumulator

In this section, we present the main building block of our naming service. Specifically, in Section 4.1, we provide the definition of a public-state, additive, universal accumulator and, in Section 4.2, we present a construction for such an accumulator under the strong-RSA assumption in the Random Oracle model.

4.1 Definition of a Public-State, Additive, Universal Accumulator

At a high level, we consider an accumulator as *public-state*, if one can perform all of its operations by only having access to its public-key, i.e., no trapdoor knowledge is required. According to the terminology presented in [13], an accumulator is *additive*, if it only allows for addition of elements, and *universal*, if it allows for both membership and non-membership witnesses. In the following, we present the definition of a public-state, additive, universal accumulator. Our definition employs two trusted parties. The first one, T, runs the key-generation algorithm ($\mathsf{KeyGen}(1^\lambda)$) and publishes the accumulator's public-key. The second one, the "accumulator manager" T_{acc}, is responsible for maintaining the accumulator. [1]

Definition 4.1 (public-state, additive, universal accumulator)**.** Let D be the domain of the accumulator's elements, and X, the current accumulated set. A public-state, additive, universal accumulator consists of the following algorithms:

[1] Note that the notion of a *public-state* accumulator is different from that of a *strong* accumulator, as defined in [16]. In a strong accumulator, the KeyGen algorithm, which produces the initial value of the accumulator, is publicly executable and any party can verify the validity of its output. In contrast, in a public-state accumulator, the KeyGen algorithm is run by a trusted party T.

- KeyGen: On input a security parameter λ, it generates a key pair (pk, sk) and outputs pk. This algorithm is run by T.
- InitAcc: On input pk and the empty accumulated set $X = \emptyset$, it outputs an accumulator value c_0. This algorithm is run by T_{acc}.
- Add: On input pk, an element $x \in D$ to be added and an accumulator value c, it outputs (c', W), where c', is the new value of the accumulator, and W, is a membership witness for x.
- MemWitGen: On input pk, X, c and $x \in X$, it outputs a membership witness W for x.
- NonMemWitGen : On input pk, X, c, x, where, $x \in D$ and $x \notin X$, it outputs a non-membership witness W for x.
- UpdMemWit: On input pk, x, y, W, where, W is a membership witness for x, it outputs an updated membership witness W' for x. This algorithm is run after $(c', W_y) \leftarrow \mathsf{Add}(pk, y, c)$, where, W_y is a membership witness for y.
- UpdNonMemWit : On input pk, x, y, W, where, $x, y \in D$, $x \neq y$ and W is a non-membership witness for x, it outputs an updated non-membership witness W' for x. This algorithm is run after $(c', W_y) \leftarrow \mathsf{Add}(pk, y, c)$.
- VerifyMem : On input $pk, x \in D, W$ and c, it outputs 1 or 0.
- VerifyNonMem : On input $pk, x \in D, W$ and c, it outputs 1 or 0.

Informally, an accumulator is *correct* if, for any honestly produced membership witness, the membership verification algorithm outputs 1, and if, for any honestly produced non-membership witness, the non-membership verification algorithm outputs 1. Furthermore, we consider a universal accumulator as *secure* if, no p.p.t. adversary can produce a valid non-membership witness for a member of the accumulated set, nor, a valid membership witness for an element which is not a member of the accumulated set. The security property of an accumulator can be met as *collision-freeness*, or, *soundness* in the literature. A formal definition for security is given below (Definition 4.2), utilizing a game between a Challenger $\mathcal{C}$ and an adversary $\mathcal{A}$, as illustrated in Figure 1. For a formal definition of correctness, we refer the interested reader to [37].

Definition 4.2. *We say that an accumulator is secure if, for any p.p.t. adversary $\mathcal{A}$ interacting with a challenger $\mathcal{C}$, as illustrated in the security game of Figure 1, it holds that* $\mathsf{Pr}[\mathcal{G}_{\mathcal{A}}^{\mathsf{acc-sec}}(1^\lambda) = 1] = \mathsf{negl}(\lambda)$.

4.2 Construction

In Figure 2, we present a construction of a public-state, additive, universal accumulator. We aim to accumulate identities or (identity,public-key) pairs, i.e., arbitrary strings. Thus, the accumulator's domain is $D = \{0, 1\}^*$. This construction is a combination of the RSA-based universal accumulator of Li et al. [36], accompanied with a procedure Map, which maps

$\mathcal{G}_{\mathcal{A}}^{\mathsf{acc-sec}}$: On input 1^{λ},

- $\mathcal{C}$ runs $\mathsf{KeyGen}(1^{\lambda})$, generates (pk, sk) and gives pk to the adversary $\mathcal{A}$.
- $\mathcal{A}$ first makes an $\mathsf{InitAcc}$ query to $\mathcal{C}$. $\mathcal{C}$ initializes a set $X \leftarrow \emptyset$, runs $\mathsf{InitAcc}$ and returns the value c_0 to $\mathcal{A}$.
- When $\mathcal{A}$ performs an Add query for an element x, $\mathcal{C}$ sets $X \leftarrow X \cup \{x\}$ and computes $(c', W) \leftarrow \mathsf{Add}(pk, x, c)$. Then, $\mathcal{C}$ returns the pair (c', W) to $\mathcal{A}$.
- $\mathcal{A}$ outputs (x^*, W^*).

The game returns 1 if at least one of the following conditions holds:

1. $x^* \notin X$ and $\mathsf{VerifyMem}(pk, x^*, W^*, c) = 1$,
2. $x^* \in X$ and $\mathsf{VerifyNonMem}(pk, x^*, W^*, c) = 1$.

Figure 1: The security game between the adversary $\mathcal{A}$ and a challenger $\mathcal{C}$, where $\mathcal{C}$ plays the roles of both T and T_{acc}.

arbitrary strings to prime numbers. Namely, for any algorithm run on input an element $z \in \{0,1\}^*$, the party who runs the algorithm, first, executes the procedure Map, which maps z to a prime number, e.g., z_p, and then, proceeds by running the same algorithm as in the accumulator of Li et al. [36] for the prime number z_p. The procedure Map that we utilize is a modified version of a procedure suggested in [30]. Thus, we first present the procedure suggested in [30], then, based on that, we present the modified algorithm Map, which is utilized in the construction of Figure 2. Lastly, we prove that the accumulator of Figure 2 is secure according to Definition 4.2.

Mapping arbitrary strings to primes [30]. Gennaro et al. [30] describe a procedure that utilizes a universal hash function family U of functions (Definition 3.2), which maps strings of $3k$ bits to strings of k bits with the additional property that, for any $y \in \{0,1\}^k$ and given $f \in U$, one can efficiently sample uniformly from the set $\{x \in \{0,1\}^{3k} : f(x) = y\}$. On input $z \in \{0,1\}^*$, it first computes $h(z)$, where $h : \{0,1\}^* \to \{0,1\}^k$ is a collision-resistant hash function. It then samples repeatedly from the set $\{x \in \{0,1\}^{3k} : f(x) = h(z)\}$ to find a prime number $O(k^2)$ times. This procedure is collision-resistant if h is collision resistant and will output a prime number with high probability due to the following Lemma.

Lemma 4.1 ([30])**.** *Let U be a Universal Hash Function Family from $\{0,1\}^{3k}$ to $\{0,1\}^k$. Then, for all but a $(1/2^k)$-fraction of functions $f \in U$ and for any $y \in \{0,1\}^k$, a fraction of at least $1/ck$ elements in the set $\{x \in \{0,1\}^{3k} : f(x) = y\}$ are primes, for a small constant c.*

Therefore, an algorithm which samples ck^2 times from the set $\{x \in \{0,1\}^{3k} : f(x) = h(z)\}$ will fail to find a prime number only with negligible probability. For completeness, we provide a proof of Lemma 4.1 in Appendix B. The proof presented in Appendix B is similar to that provided in [47].

The domain of the accumulator is $D = \{0,1\}^*$.

- KeyGen : On input 1^λ, it generates a pair of safe primes p, q of equal length, such that, $p = 2p' + 1$, $q = 2q' + 1$ and p', q' are also primes. It computes $n = pq$ and chooses g randomly from QR_n. It sets $\ell = \lfloor \lambda/2 \rfloor - 2$ and chooses a deterministic procedure Map (as described in the previous paragraph), which receives as input an arbitrary string and outputs a prime number less than $2^{\lfloor \lambda/2 \rfloor - 2}$. It sets $pk = (n, g, \mathsf{Map})$, $sk = (p, q)$, and outputs pk.
- $\mathsf{InitAcc}$: On input pk, it outputs $c_0 = g$.
- Add : On input pk, $x \in \{0,1\}^*$ and c, it invokes Map on input x and receives a prime number x_p. Then, it computes $c' = c^{x_p} \mod n$, sets $W = c$ and outputs (c', W).
- $\mathsf{MemWitGen}$: On input pk, X and $x \in \{0,1\}^*$, it computes and outputs $W = g^{\prod_{x_i \in X \setminus \{x\}} \mathsf{Map}(x_i)}$.
- $\mathsf{NonMemWitGen}$: On input $pk, X, c, x \notin X$, it invokes Map on input x and receives a prime number x_p. Then, it computes $u = \prod_{x_i \in X} \mathsf{Map}(x_i)$. Since $gcd(x_p, u) = 1$, it runs the extended Euclidean algorithm and computes $a, b \in \mathbb{Z}$, such that, $au + bx_p = 1$. By the Euclidean division, a can be written as $a = a' + qx_p$, where $0 \leq a' < x_p$. Therefore, $a'u + (b + qu)x_p = 1$. It sets $b' = b + qu$ and computes $d = g^{-b'}$. Finally, it outputs a non-membership witness $W = (a', d) = (a \mod x, g^{-b-qu})$.
- $\mathsf{UpdMemWit}$: On input pk, x, y, W, it invokes Map on input x and receives a prime number x_p. Then, it computes and outputs $W' = W^{x_p} \mod n$.
- $\mathsf{UpdNonMemWit}$: On input $pk, x \notin X, y \in X, c$ and $W = (a, d)$, it invokes Map on inputs x,y and receives the prime numbers x_p,y_p respectively. Since $y_p \neq x_p$, it runs the extended Euclidean algorithm and computes $a_0, r_0 \in \mathbb{Z}$, such that, $a_0 y_p + r_0 x_p = 1$. Then, it multiplies both sides by a, i.e., $aa_0 y_p + ar_0 x_p = a$ and computes $a' = a_0 a \mod x_p$. Then, it finds $r \in \mathbb{Z}$, such that, $a' y_p = a + r x_p$, computes $d' = dc^r \mod n$ and outputs $W' = (a', d')$.
- $\mathsf{VerifyMem}$: On input pk, x, W, c, it invokes Map on input x and receives a prime number x_p. Then, it outputs 1, if $W^{x_p} = c \mod n$, otherwise, it outputs 0.
- $\mathsf{VerifyNonMem}$: On input pk, x, W, c, where, $W = (a, d)$, it invokes Map on input x and receives a prime number x_p. Then, it outputs 1, if $c^a = d^{x_p} g \mod n$, otherwise, it outputs 0.

Figure 2: Construction of a public-state, additive, universal accumulator.

A modified version of the algorithm in [30]. In our construction (Figure 2), we employ a deterministic version of the aforementioned Map procedure, which we suggest below. Specifically, we utilize a pseudorandom generator $G : \{0,1\}^k \to \{0,1\}^{p(k)}$, where $p(k)$ is a polynomial in k, and a labeled hash function $h : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^k$, which is collision-resistant and which is modeled as a Random Oracle.

We start by picking two labels, i.e., $\mathsf{label}_0, \mathsf{label}_1 \in \{0,1\}^*$. Then, Map, on input $z \in \{0,1\}^*$, first computes $h(\mathsf{label}_0, z)$. Next, it computes $G(h(\mathsf{label}_1, z))$. Then, it sam-

ples elements from the set $\{x \in \{0,1\}^{3k} : f(x) = h(\mathsf{label}_0, z)\}$ using as randomness $G(h(\mathsf{label}_1, z))$ and stops when a prime number is found. It is easy to see that since $\mathsf{label}_0, \mathsf{label}_1$ are the same in any run of the algorithm, the algorithm on input $z \in \{0,1\}^*$ always outputs the same value. Below, in Lemma 4.2, we show that Map finds a prime, except with negligible probability, and, in Lemma 4.3, we prove that Map is collision-resistant.

Lemma 4.2. *Let $z \in \{0,1\}^*$. The algorithm* Map*, on input z, outputs a prime number, except with negligible probability, assuming that, G is a pseudorandom generator and, the hash function h, is a random oracle.*

Proof. Assume that Map, on input $z \in \{0,1\}^*$, fails to find a prime number with non-negligible probability α. We will construct a p.p.t distinguisher D, which breaks the property of the pseudorandom generator G, as this is defined in Definition 3.3. Recall that the only difference of the procedure in [30] and the algorithm Map described in the previous paragraph is the samping of elements from the set $X = \{x \in \{0,1\}^{3k} : f(x) = h(\mathsf{label}_0, z)\}$, i.e., in the former case, elements are sampled uniformly at random while, in the latter case, elements are sampled by using as randomness the output of the PRG G. Based on that, we consider the following p.p.t. distinguisher D:

- On input a string $x \in \{0,1\}^{p(k)}$, sample from the set $X = \{x \in \{0,1\}^{3k} : f(x) = h(\mathsf{label}_0, z)\}$ using as randomness the string x.
- If a prime number p is output, then, return 1, else, return 0.

First, since h is a random oracle, the seed $h(\mathsf{label}_1, z)$ is considered random. Then, if $x = r$, where r is chosen uniformly at random, by Lemma 4.1, we have that $\mathsf{Pr}[D(r) = 1] = 1 - \mathsf{negl}(k)$. By the assumption that Map fails to find a prime with non-negligible probability α, we have that

$$\left|\mathsf{Pr}[D(r') = 1] - \mathsf{Pr}[G(h(\mathsf{label}_1, z)) = 1]\right| = 1 - \mathsf{negl}(k) - (1 - \alpha) = \alpha - \mathsf{negl}(k), \quad (1)$$

which is a contradiction, according to Definition 3.3.

□

Lemma 4.3. *The algorithm* Map *is collision-resistant if the hash function h is collision-resistant. Namely, no p.p.t. adversary can find $z_1, z_2 \in \{0,1\}^*$ with $z_1 \neq z_2$, such that, the algorithm* Map *returns the same prime p.*

Proof. We assume that Map is not collision resistant, i.e., there is a p.p.t. adversary $\mathcal{A}$, which finds two different z_1, z_2, such that, $\mathsf{Map}(z_1) = \mathsf{Map}(z_2) = p$. This requires that the algorithm Map samples elements from the same set of solutions $X = \{x \in \{0,1\}^{3k} : f(x) = h(\mathsf{label}_0, z_1)\}$. Thus, $\mathcal{A}$ should find a collision in the hash function h, i.e., $\mathcal{A}$ finds z_1, z_2 such that $h(\mathsf{label}_0, z_1) = h(\mathsf{label}_0, z_2)$. However, this holds only with negligible probability. □

Security of the accumulator of Figure 2. Before formally proving the security of the accumulator of Figure 2, we first give a simple example as to why the procedure Map has to be deterministic in our construction, justifying in this way why we cannot use the procedure of [30] as it is. First, assume that we used the procedure of [30] without the

suggested modification and that an element $x \in \{0,1\}^*$ was added in our accumulator. This means that T_{acc} first produces a prime x_p and then adds x_p in the underlying RSA accumulator. Then, an adversary can produce a non-membership witness W for x simply by producing a different prime $x'_p \neq x_p$ for the element x and then running the non-membership witness generation algorithm for x'_p. Therefore, the security property of the accumulator, as defined in the security game of Figure 1, would not hold, since the adversary can output (x, W), such that, $x \in X$ and $\mathsf{VerifyNonMem}(pk, x, W, c) = 1$.

The security of our accumulator is derived by the security of the accumulator of Li et al. [36], which is proven secure under the strong-RSA assumption, and the properties of the algorithm Map, as proven in Lemma 4.2 and Lemma 4.3.

Theorem 4.1. *The accumulator of Figure 2 is secure according to Definition 1 under the strong-RSA assumption and the collision-resistance of* Map *in the Random Oracle model.*

Proof. Assume there is a p.p.t. adversary $\mathcal{A}$, which breaks the security of the accumulator of Figure 2. Then, according to Definition 1, $\mathcal{A}$ outputs (x^*, W^*), such that: (1) $x^* \notin X$ and $\mathsf{VerifyMem}(pk, x^*, W^*, c) = 1$, or, (2) $x^* \in X$ and $\mathsf{VerifyNonMem}(pk, x^*, W^*, c) = 1$. Suppose that (1) holds. Then, there are two possible cases: (a) $\mathcal{A}$ comes up with x, x^*, such that, $\mathsf{Map}(x) = \mathsf{Map}(x^*)$ and $x \in X$, thus, breaking the collision-resistance of the Map procedure, or, (b) $\mathcal{A}$ computes a valid membership witness W^* for a prime x^*_p, where, $\mathsf{Map}(x^*) = x^*_p$ and $x^* \notin X$. In the latter case, we can construct a p.p.t adversary $\mathcal{B}$, which breaks the strong-RSA assumption. We refer for further details to the proof of Li et al. [36]. Next, assume that case (2) holds. This implies two possible scenarios: First, $\mathcal{A}$ comes up with a valid non-membership witness W^* for a prime x^*_p, where, $\mathsf{Map}(x^*) = x^*_p$ and $x \in X$. This means that we can construct a p.p.t. adversary $\mathcal{B}$, which breaks the strong-RSA assumption (see the proof of Li et al. [36]) and, therefore, we have a contradiction. In the second scenario, the procedure Map, on input x^*, outputs two different primes (with non-negligible probability) if we run it twice, e.g., $x^*_{p_1}$ and $x^*_{p_2}$. This means that if $x^*_{p_1}$ is added in the accumulator first, then it would be possible for $\mathcal{A}$ to compute a valid non-membership witness W^* for $x^*_{p_2}$. However, this is impossible, since the procedure Map is deterministic . □

Constructing a universal accumulator from an additive, universal accumulator [13]. Assume that ACC^{add}_U is an additive, universal accumulator, which accumulates elements of the form (x, i, op), where, x is the element to be added, i, is an index, and op, is either a or d. We construct a universal accumulator ACC_U, from ACC^{add}_U, as follows. When an element x is added to ACC_U for the first time, T_{acc} adds the value $(x, 1, a)$ to ACC^{add}_U. Otherwise, it adds (x, i, a), where, the index i indicates that this is the i-th time that x is added to ACC^{add}_U. When an element x is deleted from ACC_U, T_{acc} adds (x, i, d) to ACC^{add}_U. In order to prove membership of x in ACC_U, one should find an index i, such that, $\big((x, i, a) \in ACC^{add}_U\big) \wedge \big((x, i, d) \notin ACC^{add}_U\big)\big)$. Accordingly, to prove that $x \notin X$, one should either prove that $(x, 1, a) \notin ACC^{add}_U$, or, find an index i, such that, $\big((x, i-1, d) \in ACC^{add}_U\big) \wedge \big((x, i, a) \notin ACC^{add}_U\big)\big)$.

The naming service functionality $\mathcal{F}_{ns}$:

- On input (sid, Init) by a server S_i, $\mathcal{F}_{ns}$ sends $(sid, \mathsf{Init}, S_i)$ to $\mathcal{S}$. If $\mathcal{S}$ returns allow, $\mathcal{F}_{ns}$ sets $Y \leftarrow Y \cup \{S_i\}$ (where Y is initialized as $Y = \emptyset$) and returns $\mathsf{success}$ to S_i. When all servers have sent a message (sid, Init), $\mathcal{F}_{ns}$ sets $\mathsf{flag} = \mathsf{start}$. Also, $\mathcal{S}$ corrupts a number of clients. We denote as C_{cor} the set of corrupted clients.
- On input (sid, Setup, R) by T, $\mathcal{F}_{ns}$ checks if $\mathsf{flag} = \mathsf{start}$ and forwards (sid, Setup, R) to $\mathcal{S}$. If $\mathcal{S}$ returns allow, $\mathcal{F}_{ns}$ stores R, initializes a set $X \leftarrow \emptyset$, sets $\mathsf{flag} = \mathsf{manage}$ and returns $\mathsf{success}$ to T.
- On input $(sid, \mathsf{Register}, id, pk)$ by a client C, $\mathcal{F}_{ns}$ checks if $\mathsf{flag} = \mathsf{manage}$ and forwards $(sid, \mathsf{Register}, id, pk)$ to $\mathcal{S}$.
  - If $\mathcal{S}$ returns allow, $\mathcal{F}_{ns}$ checks if there is $(id, \cdot) \in X$. If there is no $(id, \cdot) \in X$, it sets $X \leftarrow X \cup (id, pk)$ and returns $\mathsf{success}$ to C via public delayed output, otherwise, it returns fail to C via public delayed output.
  - If $\mathcal{S}$ returns fail, then $\mathcal{F}_{ns}$ returns fail to C via public delayed output.
  - If $\mathcal{S}$ returns $(sid, \mathsf{Register}, id', pk', C)$, $\mathcal{F}_{ns}$ checks if $C \in C_{cor}$ and if there is $(id', \cdot) \in X$. If there is no $(id', \cdot) \in X$, it sets $X \leftarrow X \cup (id', pk')$ and returns $\mathsf{success}$ to C via public delayed output, otherwise, it returns fail to C via public delayed output.
- On input $(sid, \mathsf{Revoke}, id, pk, \mathsf{aux})$ by C, $\mathcal{F}_{ns}$ checks if $\mathsf{flag} = \mathsf{manage}$ and forwards $(sid, \mathsf{Revoke}, id, pk, \mathsf{aux})$ to $\mathcal{S}$.
  - If $\mathcal{S}$ returns allow, $\mathcal{F}_{ns}$ checks whether $R(pk, \mathsf{aux}) = 1$ and $(id, pk) \in X$. If both conditions hold, $\mathcal{F}_{ns}$ computes $X \leftarrow X \setminus (id, pk)$ and returns $\mathsf{success}$ to C via public delayed output, otherwise, it returns fail to C via public delayed output.
  - If $\mathcal{S}$ returns fail, then $\mathcal{F}_{ns}$ returns fail to C via public delayed output.
  - If $\mathcal{S}$ returns $(sid, \mathsf{Revoke}, id', pk', \mathsf{aux}', C)$, $\mathcal{F}_{ns}$ checks if $C \in C_{cor}$, $R(pk', \mathsf{aux}') = 1$ and $(id', pk') \in X$. If so, $\mathcal{F}_{ns}$ computes $X \leftarrow X \setminus (id', pk')$, and returns $\mathsf{success}$ to C via public delayed output, otherwise, it returns fail to C via public delayed output.
- On input $(sid, \mathsf{Retrieve}, id)$ by C, if $\mathsf{flag} = \mathsf{manage}$, $\mathcal{F}_{ns}$ forwards this message to $\mathcal{S}$. If $\mathcal{S}$ returns allow, then, if there is a pair $(id, pk) \in X$, for some pk, $\mathcal{F}_{ns}$ returns pk to C via public delayed output, otherwise, it returns $\perp$. If $\mathcal{S}$ returns fail to $\mathcal{F}_{ns}$, then $\mathcal{F}_{ns}$ returns fail to C via public delayed output.
- On input $(sid, \mathsf{VerifyID}, id)$ by a client C, if $\mathsf{flag} = \mathsf{manage}$, $\mathcal{F}_{ns}$ forwards this message to $\mathcal{S}$. If $\mathcal{S}$ returns allow, then, if there is a pair $(id, pk) \in X$, for some pk, $\mathcal{F}_{ns}$ returns 1 to C via public delayed output, otherwise, it returns 0. If $\mathcal{S}$ returns fail to $\mathcal{F}_{ns}$, then $\mathcal{F}_{ns}$ returns fail to C via public delayed output.
- On input $(sid, \mathsf{VerifyMapping}, id, pk)$ by a client C, if $\mathsf{flag} = \mathsf{manage}$, $\mathcal{F}_{ns}$ forwards this message to $\mathcal{S}$. If $\mathcal{S}$ returns allow, then, if there is a pair $(id, pk) \in X$, $\mathcal{F}_{ns}$ returns 1 to C via public delayed output, otherwise, it returns 0. If $\mathcal{S}$ returns fail to $\mathcal{F}_{ns}$, then $\mathcal{F}_{ns}$ returns fail to C via public delayed output.

Figure 3: The naming service functionality $\mathcal{F}_{ns}$ interacts with a set of n clients, a set of m servers, a trusted party T and the simulator $\mathcal{S}$. It allows clients to register, revoke, retrieve and verify (id, pk) mappings.

5 Defining a Naming Service Functionality

In this section, we describe the security of a naming service in the UC framework ([19]) by defining it as an ideal functionality $\mathcal{F}_{ns}$ (Figure 3). $\mathcal{F}_{ns}$ interacts with n clients, m servers, a party T, which is responsible for the setup, and an adversary $\mathcal{S}$, which is called the simulator. It stores (identity,public-key) pairs and supports a number of operations. The servers are responsible for running the naming service and, therefore, before the setup, we require that all servers send a (sid, Init) message. During setup, the party T specifies a relation R, which defines under which condition a public key can be revoked. In practice, this relation might be a verification algorithm for a NIZK proof, or, a signature on a randomly selected message. After the setup phase, a client can register an (identity,public-key) pair, assuming the identity is available, and, can revoke an (identity,public-key) pair, assuming her public key satisfies relation R. Furthermore, she is able to retrieve the public key of a registered identity and check, whether an identity, or, an (identity,public-key) pair, is registered or not. Our model considers only static corruptions, thus, we assume that the simulator specifies the set of corrupted clients, C_{cor}, before setup. The party T is considered trusted, thus, the simulator, $\mathcal{S}$, is not allowed to corrupt T. Also, note that, in practice, this functionality cannot be realized for any corruption model for the m servers. However, the corruption model for the servers depends on the protocol with which we aim to realize the functionality $\mathcal{F}_{ns}$.

6 Naming Service Implementation

At a high-level, the service must allow the storage, retrieval and deletion of (identity,public-key) pairs. The main barrier in realizing a smart contract-based DPKI is the size of its state, which, being its most expensive resource to access, must be minimized. We resolve this issue in a twofold manner. First, we separate storage from the process of verifying the validity of mappings by maintaining two public-state, universal accumulators (as presented in Section 4) as our smart contract's state. Second, our accumulators are RSA-based and have constant size. The public-state property is required as the contract's state is publicly accessible. To circumvent the issue that deletions require access to an accumulator's private key, we employ the methodology that we presented at the end of Section 4.

In Figure 4, we define the functionality $\mathcal{F}_{TP}$, which captures the role of the smart contract in our protocol. This functionality interacts with a party T, a set of n clients and a set of m servers, some of which may be corrupted by the adversary prior to the initialization phase. $\mathcal{F}_{TP}$ is initialized by a trusted party T by receiving as input a program P . The state of $\mathcal{F}_{TP}$ is updated after a call to the program P and the output is received by the calling party. Note that the implementation of $\mathcal{F}_{TP}$ requires an honest majority of servers, along the lines of [28, 29, 12]. The adversary has always full knowledge of all the computations performed and may interfere by either aborting, or, allowing, an execution of P at will. However, he is restricted from modifying the output. Implementing $\mathcal{F}_{TP}$ using a blockchain protocol has the servers acting as "miners" and T and the clients interacting with the blockchain by posting transactions. Installing a program P is a special transaction that includes P in the blockchain and, subsequently, executing P requires it to be run by all miners and recording its state update in the blockchain. The security properties of the

$\mathcal{F}_{TP}$:

- On input (sid, InitTP) by a server S_i, $\mathcal{F}_{TP}$ sets $S_{init} \leftarrow S_{init} \cup \{S_i\}$(initialized as $S_{init} \leftarrow \emptyset$) and informs the adversary $\mathcal{A}$ that S_i is initialized by sending $(sid, \mathsf{InitTP}, S_i)$. Then, $\mathcal{F}_{TP}$ returns $\mathsf{success}$ to S_i. If all the servers $S_1, \ldots, S_m$ have sent a message (sid, InitTP), $\mathcal{F}_{TP}$ sets $\mathsf{flag} = \mathsf{ready}$.
- On input $(sid, \mathsf{Install}, P)$ by T, if $\mathsf{flag} = \mathsf{ready}$, send $(sid, \mathsf{Install}, P)$ to $\mathcal{A}$. If $\mathcal{A}$ returns allow, set $\mathsf{flag} = \mathsf{start}$, store P, set $state \leftarrow \varepsilon$, where ε is the empty string, and return $\mathsf{success}$ to T.
- On input (sid, x) by a client C, if $\mathsf{flag} = \mathsf{ready}$, send (sid, x) to $\mathcal{A}$. If $\mathcal{A}$ returns allow, run P on input $(x, state)$ and output $(y, state')$. Set $state \leftarrow state'$ and return $(y, state')$ to C via public delayed output. If x is an invalid input for program P, return $\perp$ to C via public delayed output.

Figure 4: The functionality $\mathcal{F}_{TP}$ captures the role of the smart contract. It interacts with a trusted party T, a set of n clients, a set of m servers and the adversary $\mathcal{A}$.

1. On input $(\mathsf{Setup}, params)$, where $params$ is of the form (pk_1, pk_2, R), run $\mathsf{InitAcc}$ on input pk_1 and on input pk_2 and compute $c_{0,1}, c_{0,2}$ respectively. This procedure initializes two public-state, additive, universal accumulators c_1 and c_2 by setting $c_1 \leftarrow c_{0,1}, c_2 \leftarrow c_{0,2}$. It sets $state \leftarrow (params, c_1, c_2)$ and returns $state$.
2. On input $(\mathsf{Register}, id, pk, i, W_1, W_2)$,
   (a) If $i = 1$, check if $\mathsf{VerifyNonMem}(pk_2, (id, 1, a), W_1, c_2) = 1$.
   (b) If $i \geq 2$, check if $\mathsf{VerifyNonMem}(pk_2, (id, i, a), W_1, c_2) = 1$ and $\mathsf{VerifyMem}(pk_2, (id, i-1, d), W_2, c_2) = 1$.

   If all the above checks succeed, run $(c_1', W_1') \leftarrow \mathsf{Add}(pk_1, (id, pk, i, a), c_1)$ and $(c_2', W_2') \leftarrow \mathsf{Add}(pk_2, (id, i, a), c_2)$. Update $state$ by setting $c_1 \leftarrow c_1'$ and $c_2 \leftarrow c_2'$, and return $((c_1', W_1'), (c_2', W_2'))$. Otherwise, return fail.
3. On input $(\mathsf{Revoke}, id, pk, i, W_1, W_2, W_3, \mathsf{aux})$,
   (a) Check if $R(pk, \mathsf{aux}) = 1$.
   (b) Check if $\mathsf{VerifyMem}(pk_2, (id, i, a), W_1, c_2) = 1$, $\mathsf{VerifyNonMem}(pk_2, (id, i, d), W_2, c_2) = 1$ and $\mathsf{VerifyMem}(pk_1, (id, pk, i, a), W_3, c_1) = 1$.

   If none of the above verifications fail, run $(c_2', W_2') \leftarrow \mathsf{Add}(pk_2, (id, i, d), c_2)$ and $(c_1', W_1') \leftarrow \mathsf{Add}(pk_1, (id, pk, i, d), c_1)$. Update $state$ by setting $c_1 \leftarrow c_1'$ and $c_2 \leftarrow c_2'$ and return $((c_1', W_1'), (c_2', W_2'))$. Otherwise, return fail.
4. On input $\mathsf{RetrieveState}$, return $state \leftarrow (params, c_1, c_2)$.

Figure 5: The program P which is input to $\mathcal{F}_{TP}$, during initialization, in our construction.

$\mathcal{F}_{UDB}$:

- On input $(sid, \mathsf{InitUDB})$ by a server S_i, $\mathcal{F}_{UDB}$ sets $S'_{init} \leftarrow S'_{init} \cup \{S_i\}$ (initialized as $S'_{init} \leftarrow \emptyset$) and sends $(sid, \mathsf{InitUDB}, S_i)$ to $\mathcal{A}$. Then, $\mathcal{F}_{UDB}$ returns $\mathsf{success}$ to S_i. If all servers $S_1, \ldots, S_m$ have sent a message $(sid, \mathsf{InitUDB})$, $\mathcal{F}_{UDB}$ sets $\mathsf{flag} = \mathsf{ready}$, $DBstate \leftarrow \emptyset$ and $p \leftarrow 0$.
- On input (sid, Post, x) by a client C, if $\mathsf{flag} = \mathsf{ready}$, forward $(sid, \mathsf{Post}, x, C)$ to $\mathcal{A}$. If $\mathcal{A}$ sends allow, set $p \leftarrow p + 1$, $DBstate[p] \leftarrow x$ and return $\mathsf{success}$ to C.
- On input $(sid, \mathsf{RetrieveDB})$ by a client C, if $\mathsf{flag} = \mathsf{ready}$, forward $(sid, \mathsf{RetrieveDB}, C)$ to $\mathcal{A}$. If $\mathcal{A}$ returns allow, return $DBstate$ to C.
- On input $(sid, \mathsf{ChangeDBstate}, DBstate')$ by $\mathcal{A}$, set $DBstate \leftarrow DBstate'$.

Figure 6: The functionality $\mathcal{F}_{UDB}$ models an unreliable database and interacts with a set of n clients, a set of m servers and the adversary $\mathcal{A}$.

underlying blockchain, specifically related to peristence of transactions, cf. [28, 29, 12], imply the security of $\mathcal{F}_{TP}$'s realization.

Furthermore, we assume that all operations are completed in a synchronous, atomic fashion. In practice, some time is required for an operation (transaction) to be validated, i.e., to be recorded in the blockchain. Nevertheless, blockchains enforce a total ordering of transactions and execute them serially, which has the same net result. In our protocol, $\mathcal{F}_{TP}$ is input the program P of Fig. 5, thus, $\mathcal{F}_{TP}$ essentially maintains the aforementioned accumulators as its state, i.e., it acts as the accumulator manager T_{acc}. To simplify the description and security analysis of our design, we assume a trusted setup phase that establishes the relation R and generates the accumulators' keys. This assumption does not introduce a single point of failure in our design as it can be replaced, in a practical implementation, with distributed protocols for generating parameters (e.g., [46]).

In Figure 6, we introduce the functionality $\mathcal{F}_{UDB}$, which handles the storage of information that are relevant to our protocol, e.g., (identity,public-key) pairs. $\mathcal{F}_{UDB}$ interacts with n clients, a set of ℓ servers and the adversary. This functionality models an "unreliable database", i.e., the adversary may tamper with its contents. Its involvement in our protocol is twofold. First, a client queries this functionality to retrieve all the necessary information that will allow her to, subsequently, interact with $\mathcal{F}_{TP}$. Second, following the completion of an interaction with $\mathcal{F}_{TP}$, the client stores in $\mathcal{F}_{UDB}$, among others, information that were output by the smart contract and reflect the new state of the system. We elaborate more on the information that clients query/store from/to $\mathcal{F}_{UDB}$ later on in this section where we provide a high-level description of each operation. A practical realization of $\mathcal{F}_{UDB}$ is out of the scope of this paper. However, an authenticated DHT network comprised of nodes that have registered in our PKI would be a suitable candidate, both in terms of security (i.e., it is Sybil resilient), as well as efficiency, due to its logarithmic message complexity.

In our scheme, we accumulate (id, pk, i, op) tuples in c_1, where, $\mathsf{op} = a$ or $\mathsf{op} = d$ mark an element as "added" or "deleted", respectively. This allows clients to infer if an (identity,public-key) mapping is valid. In c_2, we accumulate (id, i, op) tuples, which allows

clients to infer if an identity is registered in the system. In the following, and due to space limitations, we present a high-level description of the Register, Revoke, Retrieve, VerifyID, and VerifyMapping operations.

Informally, a client that is interested in registering an (identity,public-key) mapping must prove to the smart contract that the identity is, currently, available. To achieve this, she generates two witnesses. First, a membership witness of the tuple (id, i, d) for c_2, which proves that the i-th instance of this identity has been marked as deleted. Second, a non-membership witness of the tuple $(id, i+1, a)$ for c_2, which proves that the $(i+1)$-th instance of this identity, i.e., the one she is interested in registering, has not been marked as added. If both of the aforementioned conditions hold, she can convince the smart contract to accumulate her mapping in c_1. These witnesses are constructed by, first, querying $\mathcal{F}_{UDB}$ for the history of operations and, second, locating records regarding id, in an attempt to find the proper value for index i. Following a successful registration, the client posts a $(\mathsf{Register}, id, pk, i, W_1, W_2, W_3)$ record to $\mathcal{F}_{UDB}$. The witnesses W_1, W_2, W_3 facilitate queries from future clients regarding, e.g., the validity of the $(i+1)$-th instance of her (identity,public-key) mapping.

To revoke an (identity,public-key) mapping, a client generates the following proofs. First, a proof of ownership of the corresponding secret key, which is captured by the relation R. Second, a membership witness of the tuple (id, i, a) for c_2, which proves that this identity has been marked as added for index i. Third, a non-membership witness of the tuple (id, i, d) for c_2, which proves that this identity has not been marked as deleted for index i. Fourth, a membership witness of the tuple (id, pk, i, a) for c_1, which proves that the identity is indeed mapped to the same public-key that satisfies the relation R. Assuming that witnesses are generated honestly, the client convinces the contract to revoke her mapping and, then, she proceeds on posting $(\mathsf{Revoke}, id, pk, i)$ to $\mathcal{F}_{UDB}$.

To retrieve an identity's public-key, the client queries $\mathcal{F}_{UDB}$ to check whether the last record related to this identity is a registration record. If so, the client updates the witnesses W_1, W_2, W_3 stored in the retrieved registration record and, subsequently, invokes the smart contract to validate the (identity,public-key) mapping. VerifyID and VerifyMapping follow the same procedure as Retrieve to verify if an identity, or, an (identity,public-key) mapping, is registered.

In Figure 7, we present the formal description of protocol π, which realizes the functionality $\mathcal{F}_{ns}$. Recall that the entities that participate in the protocol are: 1) a trusted party T, which is used for setup purposes, 2) a functionality $\mathcal{F}_{TP}$, 3) n clients $C_1, \ldots, C_n$ and, 4) a functionality $\mathcal{F}_{UDB}$. We denote with X_1 and X_2 the sets of accumulated elements of c_1 and c_2, respectively. These sets are constructed by the client as follows. For any record of the form $(\mathsf{Register}, id, pk, i, \cdot)$, a client adds (id, pk, i, a) to X_1 and (id, i, a) to X_2. For any record of the form $(\mathsf{Revoke}, id, pk, i)$, a client adds (id, pk, i, d) to X_1 and (id, i, d) to X_2.

For ease of presentation, we have described our protocol using two accumulators. We can achieve the same net result using only one accumulator since both c_1 and c_2 accumulate arbitrary strings. Thus, we are able to accumulate both types of tuples, i.e., (id, i, op) and (id, pk, i, op), in one accumulator, while still being able to generate the (non) membership witnesses required in our protocol. To achieve this, we modify the Register and Revoke operations of program P as follows. First, the second call to Add, in either operation, receives as parameter the accumulator value that is returned from the first call to Add.

1. On input (sid, Init), a server S_i sends (sid, InitTP) and $(sid, \mathsf{InitUDB})$ to $\mathcal{F}_{TP}$ and $\mathcal{F}_{UDB}$. If S_i receives $\mathsf{success}$ by both $\mathcal{F}_{TP}$ and $\mathcal{F}_{UDB}$, then S_i returns $\mathsf{success}$.

2. On input (sid, Setup, R), the party T sends $(sid, \mathsf{Install}, P)$ to $\mathcal{F}_{TP}$, where P is the program of Figure 5. If $\mathcal{F}_{TP}$ returns as $\mathsf{success}$, then T runs $\mathsf{KeyGen}(1^\lambda)$ twice, sets $params = (pk_1, pk_2, R)$ and sends $(sid, (\mathsf{Setup}, params))$ to $\mathcal{F}_{TP}$, which executes the program P on input $(\mathsf{Setup}, params)$ (if $\mathcal{A}$ returns allow to $\mathcal{F}_{TP}$). Therefore, if $\mathcal{F}_{TP}$ returns $state \leftarrow (params, c_1, c_2)$ to T, T returns $\mathsf{success}$.

3. On input $(sid, \mathsf{Register}, id, pk)$, C sends $(sid, \mathsf{RetrieveDB})$ to $\mathcal{F}_{UDB}$. Upon receiving $DBstate$, C checks for records regarding id.

   (a) it finds the last record regarding id and checks if it is of the form $(\mathsf{Revoke}, id, pk, i)$. If it is not, C returns fail. Otherwise, she computes a non-membership witness W_1 for $(id, i+1, a)$ and a membership witness W_2 for (id, i, d) by running $\mathsf{NonMemWitGen}(pk_2, (id, i+1, a), X_2, c_2)$ and $\mathsf{MemWitGen}(pk_2, (id, i, d), X_2, c_2)$ respectively.

   (b) If no record is found, C computes a non-membership witness W_1 for $(id, 1, a)$ by running $\mathsf{NonMemWitGen}(pk_2, (id, 1, a), X_2, c_2)$, sets $W_2 = \perp$ and $i = 0$.

   Then, C sends $(sid, \mathsf{Register}, id, pk, i+1, W_1, W_2)$ to $\mathcal{F}_{TP}$, which runs P on this input. If $\mathcal{F}_{TP}$ returns $((c_1', W_1'), (c_2', W_2'), state)$, where W_1' is a membership witness for $(id, pk, i+1, a)$ in c_1' and W_2' is a membership witness for $(id, i+1, a)$ in c_2', C computes a non-membership witness W_3' for $(id, i+1, d)$ by running $\mathsf{NonMemWitGen}((id, i+1, d), X_2, c_2')$ and sends $\big(sid, \mathsf{Post}, (\mathsf{Register}, id, pk, i+1, W_1', W_2', W_3')\big)$ to $\mathcal{F}_{UDB}$. If $\mathcal{F}_{UDB}$ returns $\mathsf{success}$, C outputs $\mathsf{success}$. Otherwise, C outputs fail.

4. On input $(sid, \mathsf{Revoke}, id, pk, \mathsf{aux})$, C sends $(sid, \mathsf{RetrieveDB})$ to $\mathcal{F}_{UDB}$. Upon receiving $DBstate$, C searches for records that precede her registration record. Assuming ℓ such records, and depending if an encountered record is of the form $\mathsf{Register}$, or, Revoke, C, on each iteration, updates her witnesses as follows:

   (a) C encounters a $(\mathsf{Register}, id', pk', j, W_1^{id'}, W_2^{id'}, W_3^{id'})$. She updates W_1' by running $W_1' \leftarrow \mathsf{UpdMemWit}(pk_1, (id, pk, i, a), (id', pk', j, a), W_1')$. W_2', W_3' are updated accordingly.

   (b) C encounters a $(\mathsf{Revoke}, id', pk', j)$. She updates W_1' by running $W_1' \leftarrow \mathsf{UpdMemWit}(pk_1, (id, pk, i, a), (id', pk', j, d), W_1')$. W_2', W_3' are updated accordingly.

   Then, C sends $(sid, \mathsf{Revoke}, id, pk, i, W_1', W_2', W_3', \mathsf{aux})$ to $\mathcal{F}_{TP}$. If $\mathcal{F}_{TP}$ returns $((c_1', W_1'), (c_2', W_2'), state)$, C sends $\big(sid, \mathsf{Post}, (\mathsf{Revoke}, id, pk, i)\big)$ to $\mathcal{F}_{UDB}$. If $\mathcal{F}_{UDB}$ returns $\mathsf{success}$, C outputs $\mathsf{success}$. Otherwise, C outputs fail.

5. On input $(sid, \mathsf{Retrieve}, id)$, C sends $(sid, \mathsf{RetrieveDB})$ to $\mathcal{F}_{UDB}$. If there is no record related to id, C outputs fail, otherwise, C:

   (a) Checks if the last record related to id is of the form $\mathsf{Register}$. If so, she retrieves W_1', W_2', W_3' from the record and runs Steps 4a and 4b to compute the updated witnesses W_1', W_2', W_3'. Otherwise, C outputs fail.

   (b) Sends $(sid, \mathsf{RetrieveState})$ to $\mathcal{F}_{TP}$. If $\mathcal{F}_{TP}$ returns $state$ then C runs $\mathsf{VerifyMem}(pk_1, (id, pk, i, a), W_1', c_1), \mathsf{VerifyMem}(pk_2, (id, i, a), W_2', c_2)$ and $\mathsf{VerifyNonMem}(pk_2, (id, i, d), W_3', c_2)$. If all algorithms output 1, C outputs pk as the retrieved public key, otherwise, C outputs $\perp$.

6. On input $(sid, \mathsf{VerifyID}, id)$, C runs Step 5. If Step 5 outputs some pk, C outputs 1, otherwise, C outputs 0.

7. On input $(sid, \mathsf{VerifyMapping}, id, pk)$, C runs Step 5. If Step 5 outputs pk, C outputs 1, otherwise, C outputs 0.

Figure 7: Description of the protocol π built upon the program P of Figure 5.

Second, we invoke UpdMemWit after the second call to Add, to update the membership witness that was returned by the first call to Add. This approach, cuts down in half the contract's state, but, increases the computation of both Register and Revoke by one exponentiation and one invocation of the Map procedure.

Lastly, we show that our construction is secure by proving Theorem 6.1. The proof of Theorem 6.1 is provided in Appendix A.

Theorem 6.1. *The protocol π of Figure 7 securely realizes the functionality $\mathcal{F}_{ns}$ of Figure 3 in the $(\mathcal{F}_{TP}, \mathcal{F}_{UDB})$-hybrid world under the strong-RSA assumption in the Random Oracle model.*

References

[1] Ascap, prs and sacem join forces for blockchain copyright system. https://www.musicbusinessworldwide.com/ascap-prs-sacem-join-forces-blockchain-copyright-system/. Accessed: 2017-07-06.

[2] Emercoin - distributed blockchain services for business and personal use. http://www.emercoin.com. Accessed: 2010-09-30.

[3] Final report on diginotar hack shows total compromise of ca servers. https://threatpost.com/final-report-diginotar-hack-shows-total-compromise-ca-servers-103112/77170/. Accessed: 2017-04-07.

[4] Google takes symantec to the woodshed for mis-issuing 30,000 https certs. https://arstechnica.com/information-technology/2017/03/google-takes-symantec-to-the-woodshed-for-mis-issuing-30000-https-certs/. Accessed: 2017-04-07.

[5] Ibm pushes blockchain into the supply chain. https://www.wsj.com/articles/ibm-pushes-blockchain-into-the-supply-chain-1468528824. Accessed: 2017-07-06.

[6] Namecoin. https://namecoin.org/. Accessed: 2017-04-07.

[7] Swiss industry consortium to use ethereums blockchain. https://www.ccn.com/swiss-industry-consortium-use-ethereums-blockchain/. Accessed: 2017-07-06.

[8] Trustwave admits it issued a certificate to allow company to run man-in-the-middle attacks. https://www.techdirt.com/articles/20120208/03043317695/trustwave-admits-it-issued-certificate-to-allow-company-to-run-man-in-the-middle-attacks.shtml. Accessed: 2017-04-07.

[9] Karl Aberer. *P-Grid: A Self-Organizing Access Structure for P2P Information Systems*. Springer Berlin Heidelberg, 2001.

[10] Man Ho Au, Patrick P. Tsang, Willy Susilo, and Yi Mu. Dynamic universal accumulators for DDH groups and their application to attribute-based anonymous credential systems. In Marc Fischlin, editor, *Topics in Cryptology - CT-RSA 2009, The Cryptographers' Track at the RSA Conference 2009, San Francisco, CA, USA, April 20-24,*

2009. Proceedings, volume 5473 of *Lecture Notes in Computer Science*, pages 295–308. Springer, 2009.

[11] Agapios Avramidis, Panayiotis Kotzanikolaou, Christos Douligeris, and Mike Burmester. Chord-pki: A distributed trust infrastructure based on p2p networks. *Computer Networks*, 56, 2012.

[12] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. Bitcoin as a transaction ledger: A composable treatment. In Katz and Shacham [34], pages 324–356.

[13] Foteini Baldimtsi, Jan Camenisch, Maria Dubovitskaya, Anna Lysyanskaya, Leonid Reyzin, Kai Samelin, and Sophia Yakoubov. Accumulators with applications to anonymity-preserving revocation. In *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*, pages 301–315. IEEE, 2017.

[14] Niko Bari and Birgit Pfitzmann. Collision-free accumulators and fail-stop signature schemes without trees. In Walter Fumy, editor, *Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceeding*, volume 1233 of *Lecture Notes in Computer Science*, pages 480–494. Springer, 1997.

[15] Josh Cohen Benaloh and Michael de Mare. One-way accumulators: A decentralized alternative to digital sinatures (extended abstract). In Tor Helleseth, editor, *Advances in Cryptology - EUROCRYPT '93, Workshop on the Theory and Application of of Cryptographic Techniques, Lofthus, Norway, May 23-27, 1993, Proceedings*, volume 765 of *Lecture Notes in Computer Science*, pages 274–285. Springer, 1993.

[16] Philippe Camacho, Alejandro Hevia, Marcos A. Kiwi, and Roberto Opazo. Strong accumulators from collision-resistant hashing. In Tzong-Chen Wu, Chin-Laung Lei, Vincent Rijmen, and Der-Tsai Lee, editors, *Information Security, 11th International Conference, ISC 2008, Taipei, Taiwan, September 15-18, 2008. Proceedings*, volume 5222 of *Lecture Notes in Computer Science*, pages 471–486. Springer, 2008.

[17] Jan Camenisch, Markulf Kohlweiss, and Claudio Soriente. An accumulator based on bilinear maps and efficient revocation for anonymous credentials. In Stanislaw Jarecki and Gene Tsudik, editors, *Public Key Cryptography - PKC 2009, 12th International Conference on Practice and Theory in Public Key Cryptography, Irvine, CA, USA, March 18-20, 2009. Proceedings*, volume 5443 of *Lecture Notes in Computer Science*, pages 481–500. Springer, 2009.

[18] Jan Camenisch and Anna Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In Moti Yung, editor, *Advances in Cryptology - CRYPTO 2002, 22nd Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 2002, Proceedings*, volume 2442 of *Lecture Notes in Computer Science*, pages 61–76. Springer, 2002.

[19] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. *IACR Cryptology ePrint Archive*, 2000:67, 2000.

[20] Ran Canetti. Universally composable signature, certification, and authentication. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA*, page 219. IEEE Computer Society, 2004.

[21] Larry Carter and Mark N. Wegman. Universal classes of hash functions. *J. Comput. Syst. Sci.*, 18(2):143–154, 1979.

[22] Ivan Damgård and Nikos Triandopoulos. Supporting non-membership proofs with bilinear-map accumulators. *IACR Cryptology ePrint Archive*, 2008:538, 2008.

[23] Anwitaman Datta, Manfred Hauswirth, and Karl Aberer. Beyond "web of trust": Enabling P2P e-commerce. In *CEC 2003*, pages 303–312.

[24] David Derler, Christian Hanser, and Daniel Slamanig. Revisiting cryptographic accumulators, additional properties and relations to other primitives. In Kaisa Nyberg, editor, *Topics in Cryptology - CT-RSA 2015, The Cryptographer's Track at the RSA Conference 2015, San Francisco, CA, USA, April 20-24, 2015. Proceedings*, volume 9048 of *Lecture Notes in Computer Science*, pages 127–144. Springer, 2015.

[25] John R. Douceur. The sybil attack. IPTPS '01.

[26] Carl Ellison and Bruce Schneier. Ten risks of pki: What you're not being told about public key infrastructure. 2000.

[27] Conner Fromknecht, Dragos Velicanu, and Sophia Yakoubov. A decentralized public key infrastructure with identity retention. *IACR Cryptology ePrint Archive*, 2014:803, 2014.

[28] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT*, volume 9057 of *Lecture Notes in Computer Science*, pages 281–310. Springer, 2015.

[29] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol with chains of variable difficulty. In Katz and Shacham [34], pages 291–323.

[30] Rosario Gennaro, Shai Halevi, and Tal Rabin. Secure hash-and-sign signatures without the random oracle. In Jacques Stern, editor, *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding*, volume 1592 of *Lecture Notes in Computer Science*, pages 123–139. Springer, 1999.

[31] Bela Gipp, Norman Meuschke, and André Gernandt. Decentralized trusted timestamping using the crypto currency bitcoin. *CoRR*, abs/1502.04015, 2015.

[32] Mahabir Prasad Jhanwar and Reihaneh Safavi-Naini. Compact accumulator using lattices. In Rajat Subhra Chakraborty, Peter Schwabe, and Jon A. Solworth, editors, *Security, Privacy, and Applied Cryptography Engineering - 5th International Conference, SPACE 2015, Jaipur, India, October 3-7, 2015, Proceedings*, volume 9354 of *Lecture Notes in Computer Science*, pages 347–358. Springer, 2015.

[33] Murat Karakaya, Ibrahim Korpeoglu, and Özgür Ulusoy. Free riding in peer-to-peer networks. *IEEE Internet Computing*, 13(2).

[34] Jonathan Katz and Hovav Shacham, editors. *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I*, volume 10401 of *Lecture Notes in Computer Science*. Springer, 2017.

[35] F. Lesueur, L. Me, and V. V. T. Tong. An efficient distributed pki for structured p2p networks. In *IEEE P2PC*, 2009.

[36] Jiangtao Li, Ninghui Li, and Rui Xue. Universal accumulators with efficient nonmembership proofs. In Jonathan Katz and Moti Yung, editors, *Applied Cryptography and Network Security, 5th International Conference, ACNS 2007, Zhuhai, China, June 5-8, 2007, Proceedings*, volume 4521 of *Lecture Notes in Computer Science*, pages 253–269. Springer, 2007.

[37] Atefeh Mashatan and Serge Vaudenay. A fully dynamic universal accumulator. *Proceedings of the Romanian Academy*, 14:269–285, 2013.

[38] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. IPTPS '01.

[39] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `http://bitcoin.org/bitcoin.pdf`. Accessed: 2017-04-07.

[40] Lan Nguyen. Accumulators from bilinear pairings and applications. In Alfred Menezes, editor, *Topics in Cryptology - CT-RSA 2005, The Cryptographers' Track at the RSA Conference 2005, San Francisco, CA, USA, February 14-18, 2005, Proceedings*, volume 3376 of *Lecture Notes in Computer Science*, pages 275–292. Springer, 2005.

[41] Kaisa Nyberg. Fast accumulated hashing. In Dieter Gollmann, editor, *Fast Software Encryption, Third International Workshop, Cambridge, UK, February 21-23, 1996, Proceedings*, volume 1039 of *Lecture Notes in Computer Science*, pages 83–87. Springer, 1996.

[42] Christos Patsonakis, Katerina Samari, Mema Roussopoulos, and Aggelos Kiayias. Towards a smart contract-based, decentralized, public-key infrastructure. In *CANS*, 2017.

[43] Michael K. Reiter, Matthew K. Franklin, John B. Lacy, and Rebecca N. Wright. The ω key management service. CCS '96.

[44] Leonid Reyzin and Sophia Yakoubov. Efficient asynchronous accumulators for distributed PKI. In Vassilis Zikas and Roberto De Prisco, editors, *Security and Cryptography for Networks - 10th International Conference, SCN 2016, Amalfi, Italy, August 31 - September 2, 2016, Proceedings*, volume 9841 of *Lecture Notes in Computer Science*, pages 292–309. Springer, 2016.

[45] Barkley Rosser. Explicit bounds for some functions of prime numbers. *American Journal of Mathematics*, 63:211–232, 1941.

[46] Tomas Sander. Efficient accumulators without trapdoor extended abstracts. In Vijay Varadharajan and Yi Mu, editors, *Information and Communication Security, Second International Conference, ICICS'99, Sydney, Australia, November 9-11, 1999, Proceedings*, volume 1726 of *Lecture Notes in Computer Science*, pages 252–262. Springer, 1999.

[47] Tomas Sanderand Amnon Ta-Shma and Moti Yung. Blind, auditable membership proofs. In Yair Frankel, editor, *Financial Cryptography, 4th International Conference, FC 2000 Anguilla, British West Indies, February 20-24, 2000, Proceedings*, volume 1962 of *Lecture Notes in Computer Science*, pages 53–71. Springer, 2000.

[48] Rita H. Wouhaybi and Andrew T. Campbell. Keypeer: A scalable, resilient distributed public-key system using chord, 2008.

[49] Zuoxia Yu, Man Ho Au, Rupeng Yang, Junzuo Lai, and Qiuliang Xu. Lattice-based universal accumulator with nonmembership arguments. In Willy Susilo and Guomin Yang, editors, *Information Security and Privacy - 23rd Australasian Conference, ACISP 2018, Wollongong, NSW, Australia, July 11-13, 2018, Proceedings*, volume 10946 of *Lecture Notes in Computer Science*, pages 502–519. Springer, 2018.

[50] Emre Yüce and Ali Aydin Selçuk. Server notaries: A complementary approach to the web pki trust model. *IACR Cryptology ePrint Archive*, 2016:126, 2016.

[51] Lidong Zhou, Fred B. Schneider, and Robbert Van Renesse. Coca: A secure distributed online certification authority. *ACM Trans. Comput. Syst. 2002.*

[52] Philip Zimmermann. Pretty good privacy. `https://philzimmermann.com`.

A Proof of Theorem 6.1

Theorem 6.1. *The protocol π of Figure 7 securely realizes the functionality $\mathcal{F}_{ns}$ in the $(\mathcal{F}_{TP}, \mathcal{F}_{UDB})$-hybrid world under the Strong-RSA assumption in the Random Oracle Model. Namely, for any p.p.t. adversary $\mathcal{A}$ interacting with the protocol π in the $(\mathcal{F}_{TP}, \mathcal{F}_{UDB})$-hybrid world, there is a p.p.t. simulator $\mathcal{S}$ interacting with the functionality $\mathcal{F}_{ns}$, such that, for any p.p.t. environment $\mathcal{Z}$, it holds that*

$$EXEC^{\mathcal{F}_{ns}}_{\mathcal{Z},\mathcal{S}} \overset{c}{\approx} EXEC^{\pi^{\mathcal{F}_{TP},\mathcal{F}_{UDB}}}_{\mathcal{Z},\mathcal{A}}.$$

Proof. We construct a simulator $\mathcal{S}$ (Figure 8) which emulates an execution of the protocol π in the $(\mathcal{F}_{TP}, \mathcal{F}_{UDB})$-hybrid world, in the presence of an adversary $\mathcal{A}$. $\mathcal{S}$ plays the role of T, $\mathcal{F}_{TP}$, $\mathcal{F}_{UDB}$, the role of the servers and acts on behalf of a number of honest clients in the simulation of the hybrid-world protocol π. Based on the construction of our simulator $\mathcal{S}$, we essentially show that an environment can distinguish between the executions in the hybrid and the ideal world *only* by influencing the way the membership or non-membership tests take place in the hybrid world protocol. In other words, the only inconistency between the two executions can be derived if an adversary manages to convince the functionality $\mathcal{F}_{TP}$ about false statements (of whether an element belongs to an accumulated set or not)

by thus breaking the security of at least one of the accumulators of the protocol π. Note that the relation $R(pk, \mathsf{aux})$ does not provide an opportunity for distinguishing since it is the same in both worlds.

We assume that $\mathcal{Z}$ is a p.p.t. environment. For any message sent by $\mathcal{Z}$ to a party (e.g. a client C or the party T), we examine the output both in the hybrid and ideal world. Note that in the UC model, the parties in the ideal world are dummy, in the sense that they simply forward any message they receive by the environment to the functionality and vice versa. Below, we show that for any message sent by the environment, the outputs of the parties in the hybrid and ideal world are indistinguishable and thus the environment cannot distinguish between the executions in the hybrid and ideal world.

$\mathcal{Z}$ sends (sid, Init) to a server S_i: In the hybrid world, S_i sends (sid, InitTP) to $\mathcal{F}_{TP}$ and $(sid, \mathsf{InitUDB})$ to $\mathcal{F}_{UDB}$. The functionalities $\mathcal{F}_{TP}$ and $\mathcal{F}_{UDB}$ add the server S_i to the set S_{init} and S'_{init} respectively and inform the adversary $\mathcal{A}$ that S_i is initialized. They both return $\mathsf{success}$ to S_i, which, in turn returns $\mathsf{success}$. In the ideal world, the simulator $\mathcal{S}$, upon receiving $(sid, \mathsf{Init}, S_i)$ from $\mathcal{F}_{ns}$, according to Step 1 of Figure 8, it sends allow to $\mathcal{F}_{ns}$, which returns $\mathsf{success}$ to S_i.

$\mathcal{Z}$ sends (sid, Setup, R) to the party T : In the hybrid world, the party T sends $(sid, \mathsf{Install}, P)$ to $\mathcal{F}_{TP}$. If $\mathsf{flag} = ready$ and $\mathcal{A}$ returns allow, then $\mathcal{F}_{TP}$ stores P, sets $state \leftarrow \varepsilon$ and returns $\mathsf{success}$ to T. Then, T runs $\mathsf{KeyGen}(1^\lambda)$ twice (Step 2, Figure 7), sets $params = (pk_1, pk_2, R)$ and next, sends $(sid, (\mathsf{Setup}, params))$ to $\mathcal{F}_{TP}$. If $\mathcal{A}$ returns allow, $\mathcal{F}_{TP}$ runs P on input $(\mathsf{Setup}, params)$ and returns $state \leftarrow (c_{0,1}, c_{0,2}, params)$ to T and T returns $\mathsf{success}$ to $\mathcal{Z}$. In the ideal world, assuming that $\mathsf{flag} = \mathsf{start}$, the functionality $\mathcal{F}_{ns}$ sends (sid, Setup, R) to $\mathcal{S}$. The simulator $\mathcal{S}$ follows Step 2 of Figure 8 and simulates T and $\mathcal{F}_{TP}$ in the execution of the protocol π in the presence of $\mathcal{A}$. Given that $\mathcal{A}$ returns allow to $\mathcal{S}$ when the latter plays the role of $\mathcal{F}_{TP}$, $\mathcal{S}$ returns allow to $\mathcal{F}_{ns}$. Then, $\mathcal{F}_{ns}$ returns $\mathsf{success}$ to T. Therefore, T returns $\mathsf{success}$ both in the hybrid and ideal world.

$\mathcal{Z}$ sends a message $(sid, \mathsf{Register}, id, pk)$ to a client C: We will distinguish different cases related to whether the adversary $\mathcal{A}$ has sent a message to $\mathcal{F}_{UDB}$ which changes the contents of the database before $\mathcal{Z}$ sends $(sid, \mathsf{Register}, id, pk)$ to C. In all cases, we will examine both the output of an honest client C and a corrupted client C (i.e. a client which is controlled by the adversary $\mathcal{A}$) both in the hybrid and the ideal world. We consider the following three cases:

Reg1: $\mathcal{A}$ has **not** sent $(sid, \mathsf{ChangeDBstate}, DBstate')$ to $\mathcal{F}_{UDB}$ before $(sid, \mathsf{Register}, id, pk)$ is sent to the client C. Below, we examine two subcases related to whether the identity id is already registered or not :

Reg1(a): *The identity id is not registered:* In the hybrid world, an *honest* client C sends $(sid, \mathsf{RetrieveDB})$ to $\mathcal{F}_{UDB}$ and, if $\mathcal{A}$ returns allow, then C receives $DBstate$ and computes the witnesses W_1, W_2 according to Steps 3a, 3b of Figure 7. Then, C sends $(sid, (\mathsf{Register}, id, pk, i+1, W_1, W_2))$ to $\mathcal{F}_{TP}$. Supposing that $\mathcal{A}$ returns allow to $\mathcal{F}_{TP}$, $\mathcal{F}_{TP}$ returns $((c'_1, W'_1), (c'_2, W'_2), state)$ and C computes the witness W'_3. Then, C sends $(sid, \mathsf{Post}, (\mathsf{Register}, id, pk, i+1, W'_1, W'_2, W'_3))$ to $\mathcal{F}_{UDB}$ and

Simulator $\mathcal{S}$:

1. Upon receiving $(sid, \mathsf{Init}, S_i)$ by $\mathcal{F}_{ns}$, $\mathcal{S}$, on behalf of $\mathcal{F}_{TP}$ and $\mathcal{F}_{UDB}$ sends $(sid, \mathsf{InitTP}, S_i)$ and $(sid, \mathsf{InitUDB}, S_i)$ to $\mathcal{A}$ and sends allow to $\mathcal{F}_{ns}$.

2. Upon receiving (sid, Setup, R) by $\mathcal{F}_{ns}$, $\mathcal{S}$, on behalf of the party T in the real-world protocol, runs $\mathsf{KeyGen}(1^\lambda)$ twice (according to Step 2 of Figure 7) and then, playing the role of the functionality $\mathcal{F}_{TP}$ in the real-world protocol, sends $(sid, \mathsf{Install}, P)$ to $\mathcal{A}$. If $\mathcal{A}$ returns allow then $\mathcal{S}$ sends $(sid, \mathsf{Setup}, params)$ to $\mathcal{A}$. If $\mathcal{A}$ returns allow then $\mathcal{S}$ runs P on input $(\mathsf{Setup}, params)$ and sends allow to $\mathcal{F}_{ns}$.

3. Upon receiving $(sid, \mathsf{Register}, id, pk)$ by $\mathcal{F}_{ns}$, $\mathcal{S}$, playing the role of an honest client C and the role of $\mathcal{F}_{UDB}$ in the hybrid-world protocol π, sends $(sid, \mathsf{RetrieveDB}, C)$ to $\mathcal{A}$. If $\mathcal{A}$ returns allow, $\mathcal{S}$ runs Step 3a of Figure 7. If the last record related to id is a register record, $\mathcal{S}$ sends fail to $\mathcal{F}_{ns}$, otherwise, it proceeds by running Step 3b if necessary. Then $\mathcal{S}$, playing the role of the $\mathcal{F}_{TP}$ in the real-world protocol, sends $(sid, (\mathsf{Register}, id, pk, i+1, W_1', W_2'))$ to $\mathcal{A}$. If $\mathcal{A}$ returns allow, then $\mathcal{S}$ runs P on input $(\mathsf{Register}, id, pk, i+1, W_1', W_2')$. If P outputs fail, $\mathcal{S}$ sends fail to $\mathcal{F}_{ns}$, otherwise, P returns $((c_1', W_1'), (c_2', W_2'), state)$ and $\mathcal{S}$ computes a non-membership witness W_3' for $(id, i+1, d)$. Then, on behalf of $\mathcal{F}_{UDB}$, $\mathcal{S}$ sends $(sid, \mathsf{Post}, (\mathsf{Register}, id, pk, i, W_1', W_2', W_3'))$ to $\mathcal{A}$. If $\mathcal{A}$ returns allow then $\mathcal{S}$ sends allow to $\mathcal{F}_{ns}$ and updates $DBstate$ by storing $(\mathsf{Register}, id, pk, i, W_1', W_2', W_3')$, as $\mathcal{F}_{UDB}$ does in Figure 6.

4. Upon receiving $(sid, \mathsf{Revoke}, id, pk, \mathsf{aux})$ by $\mathcal{F}_{ns}$, $\mathcal{S}$, playing the role of an honest client C and the role of $\mathcal{F}_{UDB}$, sends $(sid, \mathsf{RetrieveDB}, C)$ to $\mathcal{A}$. If $\mathcal{A}$ returns allow, $\mathcal{S}$ runs Steps 4a, 4b of Figure 7 and computes the updated witnesses W_1', W_2', W_3'. Then, $\mathcal{S}$, on behalf of $\mathcal{F}_{TP}$, sends $(sid, \mathsf{Revoke}, id, pk, i, W_1', W_2', W_3', \mathsf{aux})$ to $\mathcal{A}$. If $\mathcal{A}$ returns allow, $\mathcal{S}$ runs P on input $(\mathsf{Revoke}, id, pk, i, W_1', W_2', W_3', \mathsf{aux})$ and if it outputs $(\mathsf{fail}, state)$ then $\mathcal{S}$ sends fail to $\mathcal{F}_{ns}$. Otherwise, $\mathcal{S}$ returns allow to $\mathcal{F}_{ns}$ and updates $DBstate$ by storing $(\mathsf{Revoke}, id, pk, i)$.

5. Upon receiving $(sid, \mathsf{Retrieve}, id)$ by $\mathcal{F}_{ns}$, $\mathcal{S}$, on behalf of an honest client C and playing the role of $\mathcal{F}_{UDB}$, sends $(sid, \mathsf{RetrieveDB})$ to $\mathcal{A}$. If $\mathcal{A}$ returns allow, $\mathcal{S}$ runs Step 5a of Figure 7. If Step 5a returns fail, then, $\mathcal{S}$ returns fail to $\mathcal{F}_{ns}$, otherwise $\mathcal{S}$ runs Step 5b and simulating $\mathcal{F}_{TP}$, sends $(sid, \mathsf{RetrieveState})$ to $\mathcal{A}$. If $\mathcal{A}$ returns allow and all the algorithms at Step 5b return 1, then, $\mathcal{S}$ sends allow to $\mathcal{F}_{ns}$, otherwise it sends fail.

6. Upon receiving $(sid, \mathsf{VerifyID}, id)$ or $(sid, \mathsf{VerifyMapping}, id, pk)$ by $\mathcal{F}_{ns}$, $\mathcal{S}$ runs the simulation similarly to Step 5.

7. Upon receiving $(sid, \mathsf{ChangeDBstate}, DBstate')$ by $\mathcal{A}$, $\mathcal{S}$ sets $DBstate \leftarrow DBstate'$.

8. Upon receiving $(sid, \mathsf{Register}, id, pk)$ or $(sid, \mathsf{Revoke}, id, pk)$ by $\mathcal{F}_{ns}$ for a *corrupted* client C, $\mathcal{S}$ waits for the actions of $\mathcal{A}$, and simulates $\mathcal{F}_{TP}$ and $\mathcal{F}_{UDB}$ as in previous cases. If $\mathcal{S}$ receives $(\mathsf{Register}, id', pk', i, W_1', W_2')$ by $\mathcal{A}$, checks if $id \neq id'$ or/and $pk \neq pk'$. If the program P does not return $(\mathsf{fail}, state)$ on this input then $\mathcal{S}$ sends $(\mathsf{Register}, id', pk', C)$ to $\mathcal{F}_{ns}$. $\mathcal{S}$ runs similarly when it receives $(\mathsf{Revoke}, id', pk', i, W_1', W_2', W_3')$.

Figure 8: Simulator $\mathcal{S}$.

if $\mathcal{A}$ returns allow, C returns success in the hybrid world. In the ideal world, the client C returns success as well, because the simulator $\mathcal{S}$, as it can be seen in Step 3 of Figure 8, returns allow to $\mathcal{F}_{ns}$, playing the role of C, $\mathcal{F}_{TP}$, $\mathcal{F}_{UDB}$. Finally, $\mathcal{F}_{ns}$, verifying that id is not registered, sends success to C.

Reg1(b): *The identity id is currently registered:* In the hybrid world, an *honest* C sends $(sid, \mathsf{RetrieveDB})$ to $\mathcal{F}_{UDB}$ and when C receives $DBstate$ checks that id is registered and returns fail. In the ideal world, $\mathcal{S}$, simulating C and $\mathcal{F}_{UDB}$ according to Step 3 of Figure 8, checks that there is a register record for the identity id which has not been revoked yet and therefore returns fail to $\mathcal{F}_{ns}$. Next, $\mathcal{F}_{ns}$ returns fail to C.

A *corrupted* client C in the hybrid world may try to convince $\mathcal{F}_{TP}$ that id is **not** registered. Then, C should either provide a non-membership witness W_1 for (id, j, a), for some $j \geq 2$, such that $\mathsf{VerifyNonMem}(pk_2, (id, j, a), W_1, c_2) = 1$ and a membership witness for W_2 for $(id, j-1, d)$ such that $\mathsf{VerifyMem}(pk_2, (id, j-1, d), W_2, c_2) = 1$, or, C should provide a valid non-membership witness W_1 for $(id, 1, a)$. We show that, by the security of the accumulator c_2, such an attack takes place only with negligible probability. Recall that since id is currently registered, it holds either that (1) there is $\ell \geq 2$ such that $(id, \ell, a) \in X_2$ and $(id, \ell, d) \notin X_2$, where X_2 is set accumulated in c_2, or, (2) $(id, 1, a) \in X_2$. Starting with (1), we consider the cases where $1 < j \leq \ell$, and $j > \ell$. If $1 < j \leq \ell$, then $(id, j, a) \in X_2$ and $(id, j-1, d) \in X_2$. By the security of the accumulator c_2, C can produce a valid non-membership witness W_1 for (id, j, a) only with negligible probability. If $j > \ell$, then $(id, j, a) \notin X_2$ and $(id, j-1, d) \notin X_2$. By the security of the accumulator c_2, C cannot produce a valid membership witness W_2 for $(id, j-1, d)$. For the case (2), similarly, C can produce a valid non-membership witness W_1 for $(id, 1, a)$ only with negligible probability. Hence, if a corrupted client C sends $(sid, \mathsf{Register}, id, pk, j, W_1, W_2)$ to $\mathcal{F}_{TP}$, following the reasoning described above, $\mathcal{F}_{TP}$ will return $(\mathsf{fail}, state)$ (except with negligible probability). Therefore, C returns fail in the hybrid world. Consistently to the hybrid world, in the ideal world, C would also return fail. In detail, the simulator $\mathcal{S}$, according to Step 8, waits for the actions of $\mathcal{A}$, i.e., the corrupted client C. Then, $\mathcal{S}$ simulates $\mathcal{F}_{TP}$ in the eyes of the corrupted client C and since $\mathcal{F}_{TP}$ returns $(\mathsf{fail}, state)$, $\mathcal{S}$ sends fail to $\mathcal{F}_{ns}$. Then, $\mathcal{F}_{ns}$ sends fail to C and C returns fail.

Reg2: $\mathcal{A}$ has sent $(sid, \mathsf{ChangeDBstate}, DBstate')$ to $\mathcal{F}_{UDB}$ before $\mathcal{Z}$ sends $(sid, \mathsf{Register}, id, pk)$, such that $DBstate' \neq DBstate$ and the set X_2' derived by $DBstate'$ is different from the set X_2 accumulated in c_2[2]. We consider the following subcases:

Reg2(a): *The identity id is not registered but the last record including id in $DBstate'$ is of the form* $(\mathsf{Register}, id, pk, j, W_1, W_2, W_3)$*:* In the hybrid world, an *honest* C sends $(sid, \mathsf{RetrieveDB})$ to $\mathcal{F}_{UDB}$ and when C receives $DBstate'$, checks that id is registered and returns fail. In the ideal world, $\mathcal{S}$, simulating C and

[2] As we explained in Section 5, a set X_2' is derived by $DBstate'$ in the following way: For any record of the form $(\mathsf{Register}, id, pk, i, W_1, W_2, W_3)$, (id, i, a) is added to X_2 and for any record of the form $(\mathsf{Revoke}, id, pk, i)$, (id, i, d) is added to X_2.

$\mathcal{F}_{UDB}$, sends fail to $\mathcal{F}_{ns}$, which returns fail to C. A *corrupted* C may send $(\mathsf{Register}, id, pk', j', W_1', W_2')$ to $\mathcal{F}_{TP}$. In that case, if $\mathcal{F}_{TP}$ returns $(\mathsf{fail}, state)$, then C will return fail. In the ideal world, $\mathcal{S}$, simulating $\mathcal{F}_{TP}$, returns fail to $\mathcal{F}_{ns}$, which returns fail to C. If $\mathcal{F}_{TP}$ returns $((c_1', W_1'), (c_2', W_2'), state)$ and $\mathcal{F}_{UDB}$ returns success to C after receiving a message of the form $(sid, \mathsf{Post}, \cdot)$, then C outputs success. Respectively, in the ideal world $\mathcal{S}$, simulating $\mathcal{F}_{TP}$ and $\mathcal{F}_{UDB}$, returns allow to $\mathcal{F}_{ns}$, which first checks that id is not registered, adds the pair (id, pk') and sends success to C. In both cases, C returns consistent outputs in the hybrid and ideal world.

Reg2(b): *The identity id is not registered and the last record including id in $DBstate'$ is of the form* $(\mathsf{Revoke}, id, pk, j)$*, or there is no record for id:* The reasoning in this subcase is the same with Reg2(a), except that an honest client interacts with $\mathcal{F}_{TP}$ after receiving $DBstate'$ from $\mathcal{F}_{UDB}$.

Reg2(c): *The identity id is registered and the last record including id in $DBstate'$ is of the form* $(\mathsf{Revoke}, id, pk, j)$*, or there is no record for id:* In the hybrid world, an *honest* client C sends $(sid, \mathsf{RetrieveDB})$ to $\mathcal{F}_{UDB}$. Upon receiving $DBstate'$ from $\mathcal{F}_{UDB}$, she runs Steps 3a, 3b of the protocol π (Figure 7) and computes W_1, a non-membership witness for $(id, j+1, a)$ and W_2, a membership witness for (id, j, d) or sets $W_2 = \perp$ for the case where there is no record for id in $DBstate'$. Then, C sends $(sid, \mathsf{Register}, id, pk, j+1, W_1, W_2)$ to $\mathcal{F}_{TP}$. Assuming that an honest client C, given the accumulated set $X_2' \neq X_2$ could produce a valid non-membership witness W_1 for $(id, j+1, a)$ and a valid membership witness W_2 for (id, j, d) with non-negligible probability following the corresponding non-membership and membership generation algorithms, then, the security of the accumulator c_2 would break, using the same arguments as in the case Reg1(b). Therefore, $\mathcal{F}_{TP}$ returns $(\mathsf{fail}, state)$ to C and C returns fail in the hybrid world. The client C also returns fail in the ideal world, since $\mathcal{S}$ simulates $\mathcal{F}_{TP}$ and C and returns fail to $\mathcal{F}_{ns}$.

A *corrupted* C, in the hybrid world may send $(\mathsf{Register}, id, pk^*, \ell, W_1^*, W_2^*)$ to convince $\mathcal{F}_{TP}$ that id is not registered. Following the same analysis as in the subcase Reg1(b), we can conclude that C returns the same output both in the hybrid and ideal world.

Reg2(d): *The identity id is registered but the last record including id in $DBstate'$ is of the form* $(\mathsf{Register}, id, pk, i+1, W_1, W_2, W_3)$*:* Following the arguments of Reg2(c) with the only difference that an honest client returns fail after receiving $DBstate'$ from $\mathcal{F}_{UDB}$, we conclude that C returns consistent outputs in the hybrid and ideal world.

Reg3: *$\mathcal{A}$ has sent $(sid, \mathsf{ChangeDBstate}, DBstate')$ to $\mathcal{F}_{UDB}$ before $(sid, \mathsf{Register}, id, pk)$ sent by $\mathcal{Z}$, such that $DBstate' \neq DBstate$ but the set X_2' derived by $DBstate'$ is the same with X_2 accumulated in c_2.* In this case, a similar reasoning with Reg1 can be followed, where the adversary has not sent a message which changes the contents stored by $\mathcal{F}_{UDB}$. This happens because the accumulated set remains the same and thus an honest client is able to compute correctly the witnesses W_1, W_2.

$\mathcal{Z}$ sends $(sid, \mathsf{Revoke}, id, pk, \mathsf{aux})$ to a client C: Similarly to the previous case, we examine the output of the client C in the hybrid and ideal world by distinguishing different cases depending on whether the adversary has tampered with the contents stored by $\mathcal{F}_{UDB}$. Note that in all the cases we describe below, we assume that $R(pk, \mathsf{aux}) = 1$, otherwise, it can be easily observed that a client C returns fail both in the hybrid and ideal world.

Rev1: $\mathcal{A}$ has **not** sent $(sid, \mathsf{ChangeDBstate}, DBstate')$ to $\mathcal{F}_{UDB}$ before $(sid, \mathsf{Revoke}, id, pk, \mathsf{aux})$ is sent to C. We consider two different subcases:

Rev1(a): (id, pk) *is registered:* In the hybrid world, an *honest* C sends $(sid, \mathsf{RetrieveDB})$ to $\mathcal{F}_{UDB}$. If $\mathcal{A}$ returns allow, then $\mathcal{F}_{UDB}$ sends $DBstate$ to C. C computes the witnesses W_1', W_2', W_3' following the steps 4a, 4b of the protocol π in Figure 7. Recall that W_1' is a membership witness for (id, i, a), W_2' is a non-membership witness for (id, i, d) and W_3' is a membership witness for (id, pk, i, a). Then, C sends $(sid, \mathsf{Revoke}, id, pk, i, W_1', W_2', W_3', \mathsf{aux})$ to $\mathcal{F}_{TP}$. If $\mathcal{A}$ returns allow and since $R(pk, \mathsf{aux}) = 1$, then $\mathcal{F}_{TP}$ returns $((c_1, W_1''), (c_2, W_2''), state)$. Then C sends $(sid, \mathsf{Post}, (\mathsf{Revoke}, id, pk, i))$ to $\mathcal{F}_{UDB}$ and if $\mathcal{A}$ returns allow, $\mathcal{F}_{UDB}$ returns $\mathsf{success}$ to C. In the ideal world, $\mathcal{S}$, upon receiving $(sid, \mathsf{Revoke}, id, pk, \mathsf{aux})$ simulates $C, \mathcal{F}_{TP}$ and $\mathcal{F}_{UDB}$ as described in Step 4 of Figure 8. f $\mathcal{A}$ returns allow as response in all cases it is requested, $\mathcal{S}$ sends allow to $\mathcal{F}_{ns}$. Then, $\mathcal{F}_{ns}$ checks that $R(pk, \mathsf{aux}) = 1$ and $(id, pk) \in X$, it deletes the pair (id, pk) and sends $\mathsf{success}$ to C. Thus, an honest client C returns $\mathsf{success}$ both in the hybrid and ideal world.

Rev1(b): (id, pk) *is not registered:* In the hybrid world, an *honest* C sends $(sid, \mathsf{RetrieveDB})$ to $\mathcal{F}_{UDB}$ and if $\mathcal{A}$ returns allow, $\mathcal{F}_{UDB}$ returns $DBstate$ to C. C checks if the last record related to id is a $\mathsf{Register}$ record and, since this does not hold, C returns fail. In the ideal world, $\mathcal{S}$, simulating C and $\mathcal{F}_{UDB}$, sends fail to $\mathcal{F}_{ns}$ which, in turn, sends fail to C. As a result, C returns fail in the ideal world as well.

A *corrupted* C, in the hybrid world, may try to convince $\mathcal{F}_{TP}$ that (id, pk) is *currently* registered. We will consider three cases. First, we will consider the case where the identity id has never been registered, second, the case where id is not currently registered but id has been registered in the past, and third, the case where a pair (id, pk') is currently registered, where $pk' \neq pk$.

• In the first case, where the identity id has never been registered, C should find i and compute a membership witness W_1' for (id, i, a) and a non-membership witness W_2' for (id, i, d). However, since id has never been registered, $(id, i, a) \notin X_2$ and $(id, i, d) \notin X_2$. By the security of the accumulator c_2, an adversary can find a membership witness W_1' for $(id, i, a) \notin X_2$ only with negligible probability. Therefore, $\mathcal{F}_{TP}$ will return fail and C returns fail in the hybrid world except with negligible probability.

• We proceed with the second case, where id has been registered in the past. Similarly to the first case, C should find i and compute a membership witness W_1' for (id, i, a) and a non-membership witness W_2' for (id, i, d). Assume that id has been registered in the past ℓ times, and C chooses $i \in \{1, \ldots, \ell\}$. Then, C

has to compute a membership witness W_1' for (id, i, a) and a non-membership witness W_2' for (id, i, d). However, we have that $(id, i, a) \in X_2$ and $(id, i, d) \in X_2$ since id is not currently registered. By the security of the accumulator c_2, C can compute a non-membership witness for $(id, i, d) \in X_2$ only with negligible probability, and therefore C returns fail. If C chooses $i > \ell$, following the same arguments with the first case, C returns fail except with negligible probability (by the security of the accumulator c_2).

• In the third case, since (id, pk') is registered, there is ℓ such that $(id, \ell, a) \in X_2, (id, \ell, d) \notin X_2$, $(id, pk', \ell, a) \in X_1$ and $(id, pk', i, d) \notin X_1$. Supposing that C chooses ℓ then C, has to compute a membership witness W_1' for $(id, \ell, a) \in X_1$, a non-membership witness W_2' for $(id, \ell, d) \notin X_2$ and a membership witness W_3' for $(id, pk, \ell, a) \in X_1$ [3]. By the security of the accumulator c_1, C cannot compute a valid membership witness for $(id, pk, \ell, a) \notin X_1$. Therefore, C returns fail, except with negligible probability. If C chooses $i < \ell$ or $i > \ell$, following similar arguments with the first and the second case, by the security of the accumulator c_2, we conclude that C returns fail except with negligible probability. In the ideal world, in all the aforementioned cases, $\mathcal{S}$, simulates $\mathcal{F}_{TP}, \mathcal{F}_{UDB}$ and since $\mathcal{F}_{TP}$ returns (fail, *state*), $\mathcal{S}$ sends fail to $\mathcal{F}_{ns}$, which , in turn, sends fail to C.

Rev2: $\mathcal{A}$ has sent $(sid, \mathsf{ChangeDBstate}, DBstate')$ to $\mathcal{F}_{UDB}$ before $\mathcal{Z}$ sends $(sid, \mathsf{Revoke}, id, pk, \mathsf{aux})$, such that $DBstate' \neq DBstate$ at least one of the accumulated sets $X_i'(i \in \{1, 2\})$ derived by $DBstate'$ is different from the corresponding one derived by $DBstate$: Considering the way X_1', X_2' are computed given $DBstate'$ (see Section 5), there are two possible scenarios, (1) $X_2' \neq X_2$ and $X_1' \neq X_1$ and (2) $X_2' = X_2$ and $X_1' \neq X_1$. For both scenarios, we have the following subcases:

Rev2(a): (id, pk) *is registered*: In the hybrid world, an *honest* C sends $(sid, \mathsf{RetrieveDB})$ to $\mathcal{F}_{UDB}$ and if $\mathcal{A}$ returns allow, then C computes the witnesses W_1', W_2', W_3' according to Steps 4a, 4b of Figure 7, and sends $(\mathsf{Revoke}, id, pk, j, W_1', W_2', W_3')$ to $\mathcal{F}_{TP}$. If $\mathcal{F}_{TP}$ returns fail, then C returns fail, otherwise, if $\mathcal{A}$ returns allow after C sending $(sid, \mathsf{Post}, (\mathsf{Revoke}, id, pk, j))$ to $\mathcal{F}_{UDB}$, C returns success in the hybrid world. In the ideal world, $\mathcal{S}$, simulating $C, \mathcal{F}_{TP}, \mathcal{F}_{UDB}$, if $\mathcal{F}_{TP}$ returns fail, then $\mathcal{S}$ returns fail to $\mathcal{F}_{ns}$ and $\mathcal{F}_{ns}$ returns fail to C. Even in the case where $\mathcal{F}_{TP}$ returns success which means that $\mathcal{S}$, simulating $\mathcal{F}_{TP}$, will return allow to $\mathcal{F}_{ns}$. Since (id, pk) is registered, $\mathcal{F}_{ns}$ will delete the pair (id, pk) and it will return success to the client C. Therefore, a client C returns consistent outputs both in the hybrid and ideal world.

Rev2(b): (id, pk) *is not registered*: In the hybrid world, an *honest* C sends $(sid, \mathsf{RetrieveDB})$ to $\mathcal{F}_{UDB}$. If $\mathcal{A}$ returns allow, $\mathcal{F}_{UDB}$ returns $DBstate'$ and then C checks whether her registration record [4] exists in $DBstate'$. If there does not exist such a record, C returns fail. If the ideal world, $\mathcal{S}$ simulates C and $\mathcal{F}_{TP}$ and therefore $\mathcal{S}$ returns fail to $\mathcal{F}_{ns}$ and therefore C returns fail in the ideal world as well. If

[3] Note that even if (id, pk) had been registered in the past, X_1 would contain an element (id, pk, j, a) but for $j \neq \ell$.

[4] Note that in our protocol π the clients have state, and thus they store the last registration record.

there exists such a record, then C follows Steps 4a, 4b of Figure 7, computes the witnesses W_1', W_2', W_3' and sends $(\mathsf{Revoke}, id, pk, j, W_1', W_2', W_3')$ to $\mathcal{F}_{TP}$. By the security of the accumulators c_1, c_2, $\mathcal{F}_{TP}$ returns $(\mathsf{fail}, state)$ except with negligible probability. In more detail, if an honest client given $DBstate'$ could convince $\mathcal{F}_{TP}$ that (id, pk) is currently registered, then following the same arguments as in the case Rev1(b), the accumulators' security would break. Therefore, C returns fail in the hybrid world. In the ideal world, since $\mathcal{S}$ simulates $C, \mathcal{F}_{TP}$ sends fail to $\mathcal{F}_{ns}$ which returns fail to C. Consequently, C also returns fail in the ideal world.

A *corrupted* client C may try to convince $\mathcal{F}_{TP}$ that (id, pk) is currently registered (irrespectively of what $\mathcal{F}_{UDB}$ has sent), by sending $(\mathsf{Revoke}, id, pk, j^*, W_1^*, W_2^*, W_3^*)$ to $\mathcal{F}_{TP}$. By the security of the accumulators c_1, c_2, following the same arguments as in Rev1(b), this happens only with negligible probability. Therefore, C returns fail both in the hybrid and ideal world.

Rev3: $\mathcal{A}$ has sent $(sid, \mathsf{ChangeDBstate}, DBstate')$ until $(sid, \mathsf{Revoke}, id, pk, \mathsf{aux})$ sent by $\mathcal{Z}$, such that $DBstate' \neq DBstate$ but the sets X_1', X_2' derived by $DBstate'$ are the same as X_1, X_2: In this case, we follow a similar analysis with Rev1, since an honest client can compute correctly the witnesses W_1', W_2', W_3'.

$\mathcal{Z}$ sends $(sid, \mathsf{Retrieve}, id)$ to a client C: We consider the following cases:

Ret1: $\mathcal{A}$ has not sent $(sid, \mathsf{ChangeDBstate}, DBstate')$ to $\mathcal{F}_{UDB}$ before $\mathcal{Z}$ sends $(sid, \mathsf{Retrieve}, id)$ to C. We consider the following subcases:

Ret1(a): *There is (id, pk) which is currently registered*: In the hybrid world, an *honest* client C sends $(sid, \mathsf{RetrieveDB})$ to $\mathcal{F}_{UDB}$. If $\mathcal{A}$ returns allow, then, $\mathcal{F}_{UDB}$ sends $DBstate$ to C who follows Step 5a of protocol π in Figure 7 and computes the witnesses W_1', W_2', W_3'. Then, C sends $(sid, \mathsf{Retrievestate})$ to $\mathcal{F}_{TP}$. If $\mathcal{A}$ returns allow, then C runs Step 5b and returns pk as the retrieved public key. In the ideal world, since $\mathcal{S}$ simulates $C, \mathcal{F}_{TP}$ and $\mathcal{F}_{UDB}$, $\mathcal{S}$ returns allow to $\mathcal{F}_{ns}$. Then $\mathcal{F}_{ns}$ checks if there is a public key registered under the identity id, and since (id, pk) is currently registered, $\mathcal{F}_{ns}$ returns pk to C. Therefore, C returns consistent outputs both in the hybrid and ideal world.

Ret1(b): *id is not currently registered:* In the hybrid world, an *honest* client C sends $(sid, \mathsf{RetrieveDB})$ to $\mathcal{F}_{UDB}$, If $\mathcal{A}$ returns allow, then, $\mathcal{F}_{UDB}$ sends $DBstate$ to C. Then C checks if the last record related to id is a $\mathsf{Register}$ record. As this does not hold, C returns fail in the hybrid world. In the ideal world, $\mathcal{S}$, according to Step 5 of Figure 8, will return fail to $\mathcal{F}_{ns}$ which will return fail to C. Therefore, C returns fail in the ideal world as well.

A *corrupted* client C, may still send $(sid, \mathsf{Retrievestate})$ to $\mathcal{F}_{TP}$, however, C has no advantage in influencing the output of $\mathcal{F}_{TP}$, and therefore C returns $\perp$, both in the hybrid and ideal world.

Ret2: $\mathcal{A}$ has sent $(sid, \mathsf{ChangeDBstate}, DBstate')$ to $\mathcal{F}_{UDB}$ before $\mathcal{Z}$ sends $(sid, \mathsf{Retrieve}, id)$ to C such that the accumulated set X_2' derived by $DBstate'$ is different than X_2. We consider the following subcases:

Ret2(a): *There is (id, pk) which is currently registered and the last record related to id is a register record:* In the hybrid world, an *honest* client C sends $(sid, \mathsf{RetrieveDB})$ to $\mathcal{F}_{UDB}$. If $\mathcal{A}$ returns allow, then, $\mathcal{F}_{UDB}$ sends $DBstate$ to C who, following Step 5a of protocol π in Figure 7, computes the witnesses W_1', W_2', W_3'. Then, C sends $(sid, \mathsf{Retrievestate})$ to $\mathcal{F}_{TP}$. If $\mathcal{A}$ returns allow, then C runs Step 5b of protocol π. Even if all algorithms return 1, and C returns pk in the hybrid world then $\mathcal{S}$ will return allow to $\mathcal{F}_{ns}$ and since (id, pk) is registered C will return pk in the ideal world. If at least one algorithm returns 0, then C returns $\perp$ in the hybrid world. In the ideal world, $\mathcal{S}$, returns fail to $\mathcal{F}_{ns}$, which returns fail to C.

Ret2(b): *There is (id, pk) which is currently registered but the last record related to id is a revoke record:* The arguments for this subcase are the same as Ret2(a), except that an honest client C, checking that the last record related to id is a revoke record, will return fail.

Ret2(c): *id is not registered but the last record related to id is a register record*: In the hybrid world, an *honest* client C sends $(sid, \mathsf{RetrieveDB})$ to $\mathcal{F}_{UDB}$. If $\mathcal{A}$ returns allow, then, $\mathcal{F}_{UDB}$ sends $DBstate$ to C who, following Step 5a of protocol π in Figure 7, computes the witnesses W_1', W_2', W_3'. Then, C sends $(sid, \mathsf{Retrievestate})$ to $\mathcal{F}_{TP}$. If $\mathcal{A}$ returns allow, then C runs Step 5b of protocol π. By the security of the accumulators c_1, c_2, at least one of the verification algorithms will return 0. Therefore, C returns fail. In the ideal world, $\mathcal{S}$, sends fail to $\mathcal{F}_{ns}$ which returns fail to C.

Ret2(d): *id is not registered but the last record is a revoke record*: For this subcase we can follow a similar reasoning to Ret2(c), with the difference that an honest client C, returns fail when she checks that the last record related to id is a revoke record.

We omit the cases where an environment $\mathcal{Z}$ sends $(sid, \mathsf{VerifyID}, id)$ and $(sid, \mathsf{VerifyMapping}, id, pk)$ to a client C since the analysis is similar to the case where $\mathcal{Z}$ sends $(sid, \mathsf{Retrieve}, id)$. This can be easily observed by the description of Steps 6, 7 of the protocol π (Figure 7). This completes our proof. □

B Proof of Lemma 4.1

Lemma 4.1 ([30])**.** *Let U be a 2-universal hash function family from $\{0,1\}^{3k}$ to $\{0,1\}^k$. Then, for all but a 2^{-k} fraction of functions $f \in U$, for any $y \in \{0,1\}^k$, a fraction of at least $1/ck$ elements in $f^{-1}(y)$ are primes where c is some small constant.*

Lemma 4.1 is proven using a sequence of lemmas. We follow the proof given in the appendix of the paper [47].

Lemma B.1. *Let $U = \{f : \{0,1\}^m \to \{0,1\}^k\}$ be a 2-universal hash function family. For any $A \subseteq \{0,1\}^m$, for all but a $O(\frac{2^{2k}}{|A|})$-fraction of $f \in U$, it holds that for any $z \in \{0,1\}^k$, $\frac{|f^{-1}(z) \cap A|}{|f^{-1}(z)|} > \frac{|A|}{2^{m+1}}$.*

Lemma B.1 is proven by first showing Lemma B.2 and then Lemma B.3.

Lemma B.2. *Let $A \subseteq \{0,1\}^m$ and $z \in \{0,1\}^k$. We say that $f \in U$ is (A,z)-balanced if $\frac{2}{3} \cdot \frac{|A|}{2^k} \leq |f^{-1}(z) \cap A| \leq \frac{4}{3} \cdot \frac{|A|}{2^k}$. It holds that $\mathsf{Pr}_{f \in U}[f$ **is not** (A,z)-balanced$] \leq \frac{9 \cdot 2^k}{|A|}$.*

Proof. Assume that $A = \{a_1, \ldots, a_\ell\}$ and that we choose $f \in U$ uniformly at random. Since U is a 2-universal hash function family (Definition 3.2) we have that $\mathsf{Pr}[f(a_i) = z] = \frac{1}{2^k}$.

We define the random variable X_i, which equals 1 if $f(a_i) = z$ and 0 otherwise. Therefore, it holds that $\mathsf{Pr}[X_i = 1] = \frac{1}{2^k}$. If $X = \Sigma_{i=1}^{\ell} X_i$, then it can be easily observed that $X = |f^{-1}(z) \cap A|$. We also have that $\mu = E[X] = \ell \cdot \mathsf{Pr}[f(a_i) = z] = \ell \cdot 2^{-k}$. We will now utilize the Chebychev inequality

$$\mathsf{Pr}[|X - \mu| \geq t] \leq \frac{Var(X)}{t^2}.$$

If we set $t = \mu/3$, we have that

$$\mathsf{Pr}_{f \in U}[|X - \mu| \geq \frac{\mu}{3}] \leq \frac{9Var(X)}{\mu^2}. \tag{2}$$

Since $X_i, \ldots, X_\ell$ are pairwise independent, we have that $Var(X) = \Sigma_{i=1}^{\ell} Var(X_i)$. Therefore, $Var(X) = \Sigma_{i=1}^{\ell}\big(E[X_i^2] - E[X_i]^2\big) = \ell \frac{1}{2^k}\Big(1 - \frac{1}{2^k}\Big) \leq \ell 2^{-k}$. Hence, $\frac{9Var(X)}{\mu^2} \leq \frac{9 \cdot 2^k}{\ell}$. Therefore, by (2), it holds that

$$\frac{2}{3} \cdot \frac{|A|}{2^k} < |f^{-1}(z) \cap A| < \frac{4}{3} \cdot \frac{|A|}{2^k}, \tag{3}$$

except for $\frac{9 \cdot 2^k}{\ell}$ fraction of functions $f \in U$. □

Lemma B.3. *Let $A \subseteq \{0,1\}^m, z \in \{0,1\}^k$ and $f \in U$. We say that the pair (f,z) is "bad" for the set A if $\frac{|f^{-1}(z) \cap A|}{|f^{-1}(z)|} \leq \frac{|A|}{2^{m+1}}$. Then, for any $A \subseteq \{0,1\}^m, z \in \{0,1\}^k$,*

$$\mathsf{Pr}_{f \in U}[(f,z) \text{ is "bad" for } A] \leq \frac{18 \cdot 2^k}{|A|}. \tag{4}$$

Proof. We fix $z \in \{0,1\}^k$. Then, by Lemma B.2, if we set $A = \{0,1\}^m$, we have that

$$\frac{2}{3} \cdot 2^{m-k} < |f^{-1}(z)| < \frac{4}{3} \cdot 2^{m-k}, \tag{5}$$

except for $9 \cdot 2^{k-m}$ fraction of functions $f \in U$. Next, by Lemma B.2 for an arbitrary $A \subseteq \{0,1\}^m$, we have that

$$\frac{2}{3}|A| \cdot 2^{-k} < |f^{-1}(z) \cap A| < \frac{4}{3}|A| \cdot 2^{-k}, \tag{6}$$

except for $\frac{9 \cdot 2^k}{|A|}$ fraction of functions $f \in U$.

Combining (5), (6), it holds that

$$\frac{|f^{-1}(z) \cap A|}{|f^{-1}(z)|} > \frac{|A|}{2^{m+1}}, \tag{7}$$

except for $\frac{18 \cdot 2^k}{|A|}$ fraction of functions $f \in U$. □

By Lemma B.3, using the union bound, we can get Lemma B.1.

Proof of Lemma 4.1. Now, let A be the set of prime numbers less than 2^m and $m = 3k$, as Lemma 4.1 considers a 2-universal hash function family of functions from $\{0,1\}^{3k}$ to $\{0,1\}^k$. By the prime number theorem, we have that the number of primes which are less or equal to x, denoted as $\pi(x)$, is asymptotically $\frac{x}{\ln x}$. Making use of a non-asymptotic bound ([45]), we have that for any $x \geq 55$, $\pi(x) > \frac{x}{\ln x + 2}$. Therefore, we have that

$$|A| = \pi(2^{3k}) > \frac{2^{3k}}{\ln 2^{3k} + 2}. \tag{8}$$

By (8) and Lemma B.1, it holds that except for $(\frac{18(3k+2\log_2 e)}{2^k \log_2 e})$-fraction of functions $f \in U$,

$$\frac{|f^{-1}(z) \cap A|}{|f^{-1}(z)|} > \frac{1}{\frac{6k}{\log_2 e} + 2} \approx \frac{1}{4.16k + 2}. \tag{9}$$

□