A Decentralized Public Key Infrastructure with Identity Retention

Conner Fromknecht (conner@mit.edu), Dragos Velicanu (velicanu@mit.edu),
Sophia Yakoubov (sonka89@mit.edu)

November 11, 2014

Abstract

Public key infrastructures (PKIs) enable users to look up and verify one another's public keys based on identities. Current approaches to PKIs are vulnerable because they do not offer sufficiently strong guarantees of *identity retention*; that is, they do not effectively prevent one user from registering a public key under another's already-registered identity. In this paper, we leverage the consistency guarantees provided by cryptocurrencies such as Bitcoin and Namecoin to build a PKI that ensures identity retention. Our system, called Certcoin, has no central authority and thus requires the use of secure distributed dictionary data structures to provide efficient support for key lookup.

1 Introduction

PKI A public key infrastructure (PKI) is responsible for facilitating the authentication and distribution of public keys. It maintains a database of $(\mathsf{id}, \mathsf{pk})$ pairs, where id represents an identity, and pk represents a corresponding public key. It has a number of security goals, such as *accurate registration*, which is the inability of a user to register an identity that does not belong to him or her, and *identity retention*, which is the inability of a user to impersonate an identity already registered to someone else. In this paper, we propose a new PKI with a focus on offering identity retention. One can argue that identity retention is the more desirable security property, since identities which are lucrative impersonation targets have usually been around for a while, and are thus already registered. Moreover, in some applications of PKI, accurate registration may not be relevant at all, because identities are allocated on a first-come first-serve basis (as is the case, for example, in DNS).

Common Approaches to PKI Currently, the most commonly employed approaches to public key infrastructures (PKIs) fall into two categories: Certificate Authorities (CAs) [17] and decentralized networks of peer-to-peer certification, often referred to as Webs of Trust [21]. Both of these are far from infallible when it comes to identity retention.

Typically the more common choice in practice, a Certificate Authority (CA) acts a trusted third party that is responsible for distributing and managing digital certificates for a network of users. The use of these trusted third parties creates single points of failure in the PKI. There have been numerous recent incidents showing that too much trust is being placed in CAs. CAs have been hacked (e.g. the DigiNotar incident [14], wherein a certificate for 'google.com' was issued erroneously, compromising Google's identity retention), and have even accidentally issued subordinate root certificates to customers, enabling the customers to act as CAs themselves (e.g. the TrustWave incident [10]). Additionally, while the CA system is centralized enough to introduce single points of

failure, it is not centralized enough to ensure consistency. Since there exist multiple CAs, they may certify different public keys corresponding to the same identity, thus violating identity retention.

The second major PKI used in practice is the PGP Web of Trust. In this system, authentication is entirely decentralized; users are able to designate others as trustworthy by signing their public keys. A user thus accumulates a certificate containing his public key and digital signatures from entities that have deemed him trustworthy. The certificate is trusted by another user if she is able to verify that the certificate contains the signature of someone she trusts [21]. This system benefits from its distributed nature because it removes any single point of failure. However, PGP does not offer identity retention, because much like in the case of CAs there is no guarantee of consistency, and nothing prevents multiple users from creating public keys for the same identity.

Our Contributions In this paper, we describe Certcoin, a new completely decentralized PKI that leverages the consistency offered by blockchain-based cryptocurrencies such as Namecoin [2] to provide a stronger identity retention guarantee than any of the PKIs used in practice. Instead of having to trust a third party as in the CA system or a small set of fellow users as in the PGP system, Certcoin only requires that users trust that the *majority* of other users are not malicious. There have been other works similarly leveraging blockchain systems, such as the work of Garman, Green and Miers on the topic of anonymous credentials [15]. However, to our knowledge, the use of a blockchain for the instantiation of a full-fledged PKI has never been fully explored. We detail how Certcoin can efficiently support each PKI functionality. Because of the decentralized nature of this PKI, there is no single authority who can maintain a local dictionary data structure for efficient public key lookup. We therefore separate the functionality of verifying a known public key from that of looking up a new public key, and leverage secure distributed data structures to support each of them efficiently. In particular, we use cryptographic accumulators to facilitate fast public key verification, and distributed hash tables to facilitate fast public key lookup.

Alternative Approaches A different approach to designing better PKI systems is introducing transparency into the workings of the CAs. The Google Certificate Transparency project [3] does exactly this. They propose maintaining public, append-only logs on a number (e.g. ≤ 1000) of independent servers world-wide. Each such log server might be run by a separate CA, for instance. These logs would then be monitored for suspicious certificates by other (publicly-run) servers, and audited for consistent behavior by lightweight auditor software which can be run by anyone. This would ensure that the owner of a domain would be able to see all certificates issued for his domain, and thus would be able to spot any erroneous certificates, ensuring identity retention. However, while this approach gives transparency, it does not eliminate central points of failure, leaving room for improvement.

2 Background

Building Block: Namecoin Namecoin is a cryptocurrency designed to act as a decentralized domain name server (DNS) for .bit addresses. It supports the registration and update of domains: registering a domain costs 0.01 units of Namecoin, while updates are free. Namecoin already provides the infrastructure to be used as a PKI. Public keys can be included as 'auxiliary information'. We could use other cryptocurrencies in a similar way; however, Namecoin seems most suitable for our needs. We intend for Certcoin to be built on top of Namecoin.

We will not go into the details of the design of cryptocurrencies such as Bitcoin and Namecoin. Both implement what is essentially a public ledger by leveraging proofs of work and financial incentives.

Throughout this paper, we treat the Namecoin blockchain as a public, permanent ledger, to which information can be "posted". We refer to the entity writing the posted information to the ledger as the "block miner". We note that it is possible for a malicious, colluding majority to subvert Namecoin, and thus Certcoin; we are counting on the nonexistence of such a majority.

Building Block: Cryptographic Primitives Certcoin leverages a number of cryptographic primitives, such as *digital signatures*. There are three digital signature algorithms:

- $\mathsf{keygen}(1^k) \to (\mathsf{sk}, \mathsf{pk})$ generates a secret key together with a corresponding public key given a security parameter k.
- $\mathsf{sig}(\mathsf{sk}, \mu) \to \sigma$ produces a digital signature σ on the message μ using the secret key sk.
- $\mathsf{ver}(\mathsf{pk}, \sigma, \mu) \to b \in \{0, 1\}$ determines whether or not σ is a valid signature on μ under the secret key corresponding to the public key pk.

Digital signatures should be *existentially unforgeable* [16], meaning that without knowledge of the secret key sk corresponding to the public key pk, no probabilistic polynomial-time adversary $\mathbb{A}$ should be able to produce a signature-message pair (σ, μ) such that $\mathsf{ver}(\mathsf{pk}, \sigma, \mu) = 1$ with probability non-negligible in the security parameter k. The adversary $\mathbb{A}$ should fail even if he or she has already seen a polynomial number of valid signatures on messages of his or her choice not equal to μ.

Certcoin also makes use of *cryptographic accumulators* and *distributed hash tables*, described in Sections 5 and 6.

3 Our Scheme: Overview

Any PKI has a number of functionalities. These include:

1. Registering an identity with a corresponding public key (Section 4.1),
2. Updating the public key corresponding to a previously-registered identity (Section 4.2),
3. Looking up a public key corresponding to a given identity (Sections 4.3 and 6),
4. Verifying that a given public key corresponds to a given identity, perhaps in a way more efficient than performing a lookup (Sections 4.4 and 5), and
5. Revoking the public key corresponding to an identity (Section 4.5).

We present our scheme as a sequence of versions. Version 0 is conceptually the simplest, but involves a number of very expensive operations. Versions 1 and 2 improve upon the efficiency of Version 0 at the price of some additional complexity.

In Version 0 (Section 4), registrations, updates and revocations are handled simply by posting the appropriate information (e.g. the identity id and key pk in question) to the blockchain. Looking up the public key pk corresponding to an identity id is handled by traversing the blockchain and locating the latest value of the desired key, and verifying that pk does indeed correspond to id is handled by performing a lookup and ascertaining that the retrieved public key value matches pk. In Version 0, both lookup and verification operations take time and space linear in the number of registered identities.

In order to support public key verifications in a more efficient way, in Version 1 (Section 5) we introduce the use a *cryptographic accumulator*, which is a space-efficient data structure that supports the verification of set membership. The Certcoin accumulator contains the set of all current identity - public key pairs $(\mathsf{id}, \mathsf{pk})$, and can be used to verify that a given $(\mathsf{id}, \mathsf{pk})$ pair is current. This reduces the time and space needed for verification from linear to logarithmic in the number of registered identities.

In order to support fast public key lookup queries, in Version 2 (Section 6) we introduce the use a *distributed hash table* (DHT). This reduces the time needed for a public key lookup from linear to constant. Once a key is retrieved by means of this DHT, it can be verified using the accumulator. In order to support the verification of negative responses - that is, that a given identity does not have a current public key - we use an additional accumulator in Version 2 which requires more memory than the one introduced in Version 1.

Throughout the rest of this paper, we give more detail about how each of the three proposed versions of Certcoin support all of the PKI functionalities.

4 Certcoin Version 0: The Basics

All functionalities in Version 0 of Certcoin are supported by means of blockchain posts and blockchain traversals. Despite the apparent simplicity of the mechanisms used in Version 0, it does contain some subtleties. All of those are detailed in this section.

4.1 Registering a New Identity with a Corresponding Public Key

When a domain is initially registered in Namecoin, the transaction contains information about two public keys which will be associated with the site bought. The first public key belongs to the *online* key pair, while the second belongs to the *offline* key pair.

In Namecoin, the *online* key pair is used to authenticate messages to and from the server hosting the domain. In Certcoin, this key pair is used to authenticate an identity; a signature under this key is a required part of authentication.

The *offline* secret key is stored securely offline, so that it is not vulnerable to many of the threats that the online key might face as a result of being stored on a device connected to the internet. In Namecoin (and by extension in Certcoin), this secret key is used to revoke old keys and sign new keys in case of a key compromise. The use of the offline secret key will be explained in further detail in Section 4.5.

The registration procedure is described in full detail in Figure 1.[1] Note that we consider key generation to be outside the scope of Certcoin: we assume that users do this locally, store the produced secret keys, and register the produced public keys.

Extension: Hieirarchical Identities It is often valuable to be able to support identity hierarchies such as "google.employee1", where the fact that an entity holds this identity proves that they are in fact employed at Google. This can be achieved by requiring the entity that registered the identity "google" to sign any identity of the form "google.*" upon its registration, where "*" is an arbitrary suffix. This signature would need to be posted to the blockchain as part of the registration tuple of "google.*" in order for the registration to be accepted.

[1] Much like in Namecoin, we intend for registration to be the only operation which has a non-zero monetary cost.

Registration:

1. The identity owner posts
   $(\mathsf{id}, \mathsf{register}, \mathsf{online}, \mathsf{values} = (\mathsf{pk}_n, \sigma_n))$ and
   $(\mathsf{id}, \mathsf{register}, \mathsf{offline}, \mathsf{values} = (\mathsf{pk}_f, \sigma_f))$
   to the blockchain, where:

   - id is an identity,
   - pk_n is an online public key,
   - pk_f is an offline public key, and
   - σ_n and σ_f are two digital signatures: $\sigma_n = \mathsf{sig}(\mathsf{sk}_n, \mathsf{id})$ and $\sigma_f = \mathsf{sig}(\mathsf{sk}_f, \mathsf{id})$. These signatures demonstrate that the identity owner is able to sign with sk_n and sk_f, respectively.

2. The block miner preforms the following verifications:

   - Check that id has never previous been registered (by iterating through the block chain),
   - Check that $\mathsf{ver}(\mathsf{pk}_n, \sigma_n, \mathsf{id}) = 1$, and
   - Check that $\mathsf{ver}(\mathsf{pk}_f, \sigma_f, \mathsf{id}) = 1$.

   If any of these verifications fail, the block miner omits the posted tuples from the block. Otherwise, he or she includes them.

3. The identity owner stores the id bid of the block his identity is published in.

4. Each recipient of the mined block performs the following verifications:

   - Unless the recipient is also the identity owner, check that id is not any of the identities held by the recipient.
   - Check that $\mathsf{ver}(\mathsf{pk}_n, \sigma_n, \mathsf{id}) = 1$, and
   - Check that $\mathsf{ver}(\mathsf{pk}_f, \sigma_f, \mathsf{id}) = 1$.

   If any of the verifications fail, discard the received block.

   The first verification is necessary to ensure that the block miner did not cheat in checking that id has not previously been registered. If the first verification fails (that is, the identity being registered already belongs to the block recipient), the block recipient should post "ERROR: $(\mathsf{id}, \mathsf{pk}_n, \mathsf{pk}_f)$ invalid, identity already claimed in earlier block bid^*: $(\mathsf{id}, \mathsf{pk}_n^*)$".

   Upon receiving this post, all parties verify that id already corresponds to pk_n^* and was claimed in block bid^*, and discard the block with the new registration.

Figure 1: (V0) Registering an identity with a corresponding public key

4.2 Updating a Public Key Corresponding to a Domain

Changing a public key pk^{old} (of either the online or $\mathsf{offline}$ type) to a new public key pk^{new} of the same type is done by posting the identity in question together with the old and new public keys to the blockchain. We leverage digital signatures here to ensure that a new public key can only be

posted by the holder of the secret key corresponding to the old public key. This update transaction will only be processed if the signature verifies with the old public key pk^{old}.

Update:

1. The identity owner posts $(\mathsf{id}, \mathsf{update}, \mathsf{keytype}, \mathsf{values} = (\mathsf{pk}^{old}, \mathsf{pk}^{new}, \sigma_1, \sigma_2, \mathsf{aux}))$ to the bulletin board, where:

   - id is an identity.
   - $\mathsf{keytype}$ is the type of the key (either online or $\mathsf{offline}$).
   - pk^{old} is the old public key.
   - pk^{new} is the new public key which is to replace pk^{old}.
   - Signature $\sigma_1 = \mathsf{sig}(\mathsf{sk}^{old}, (\mathsf{id}, \mathsf{pk}^{new}))$ is a digital signature of the identity together with the new public key, signed by the old secret key. This proves that the identity owner knows the secret key sk^{old} corresponding to the old public key pk^{old}, and that pk^{new} is the intended new public key for id.
   - Signature $\sigma_2 = \mathsf{sig}(\mathsf{sk}^{new}, \mathsf{id})$ is a digital signature of the identity together signed by the new secret key. This proves that the identity owner knows the secret key sk^{new} corresponding to the new public key pk^{new}.

   Note that if, as described in Section 4.5, the update in question addresses a key compromise, σ_1 may be composed of two signatures, one under each of the two secret keys sk_n and sk_f. In this case the situation is described in the auxiliary message aux, and the verification operations are modified accordingly. We do not describe this in detail in the interest of brevity.

2. The block miner does the following:

   - Verify that pk^{old} corresponds to id as described in Section 4.4.
   - Check that $\mathsf{ver}(\mathsf{pk}^{old}, \sigma_1, (\mathsf{id}, \mathsf{pk}^{new})) = 1$.
   - Check that $\mathsf{ver}(\mathsf{pk}^{new}, \sigma_2, \mathsf{id}) = 1$.

   If any of these verifications fail, the block miner omits this post from the block. If all of the verifications succeed, he or she includes it.

3. Each recipient of the mined block performs the same verifications as the block miner. If any of these verifications fail, the recipient discards the received block.

Figure 2: (V0) Updating the public key corresponding to an identity

The update procedure is described in full detail in Figure 2. Note that while for the sake of simplicity here we describe the update operation in the context of a single key pair $(\mathsf{sk}, \mathsf{pk})$, Section 4.5 describes how multiple key-pairs (the online key-pair $(\mathsf{sk}_n, \mathsf{pk}_n)$ and the $\mathsf{offline}$ key-pair $(\mathsf{sk}_f, \mathsf{pk}_f)$) should be leveraged in the case of a key compromise.

4.3 Looking Up a Public Key Corresponding to a Given Identity

Lookup is an operation that takes in an identity id, and a public key type keytype which should be either online or offline. It outputs the current public key of type keytype corresponding to id, or it outputs $\perp$ if there isn't one.

In order to perform a lookup in Version 0 of Certcoin, a user traverses the entire blockchain. He or she verifies that the identity id was registered exactly once, and that id was not revoked. He or she then settles on the key yielded by the registration and the following sequence of updates. Note that the user does *not* need to verify all of the signatures included in the updates, because he or she already verified those signatures when first processing the blocks in which they were posted.

The lookup procedure is described in full detail in Figure 6.

Lookup: The user performing the lookup operation traverses the block chain, taking note of each posted tuple of the form $(\mathsf{id}, \mathsf{commandtype}, \mathsf{keytype}, \mathsf{values})$, where values are the values (including signatures) included in the tuple. For each post, the verifier does the following:

- If commandtype = register:
  - Parse $(\mathsf{pk}^*, \sigma) = \mathsf{values}$.
  - If this is not the very first post of the form $(\mathsf{id}, \mathsf{commandtype}, \mathsf{keytype}, \mathsf{values})$, return $\perp$.
  - Otherwise, set $\mathsf{currentpublickey} = \mathsf{pk}^*$.
- If commandtype = update:
  - Parse $(\mathsf{pk}^{old}, \mathsf{pk}^{new}, \sigma_1, \sigma_2, \mathsf{aux}) = \mathsf{values}$.
  - Set $\mathsf{currentpublickey} = \mathsf{pk}^{new}$.
- If commandtype = revoke:
  - return $\perp$.

  Once all of the posts of the form $(\mathsf{id}, \mathsf{commandtype}, \mathsf{keytype}, \mathsf{values})$ have been processed, if $\perp$ has not already been returned, return currentpublickey.

Figure 3: (V0) Looking up a public key corresponding to a given identity

4.4 Verifying that a Given Public Key Corresponds to a Given Identity

Verification is an operation that takes in an identity id, a public key type keytype which should be online or offline, and a public key pk. It outputs either TRUE or FALSE, depending on whether pk is really the public key of type keytype currently corresponding to identity id.

In order to perform a verification in Version 0 of Certcoin, a user simply performs a lookup of $(\mathsf{id}, \mathsf{keytype})$ as described in Section 4.3. If the lookup returns pk, the verification operation returns TRUE. Otherwise, the verification operation returns FALSE.

4.5 Dealing with Key Compromise, and Revoking the Public Key Corresponding to an Identity

In traditional public key infrastructures, certificates either expire at a certain age or are revoked by being added to a Certificate Revocation List (CRL) [11] or by means of the Online Certificate Status Protocol (OCSP) [12].

In Certcoin, an owner of an identity id can revoke their public key simply by posting $(\mathsf{id}, \mathsf{revoke}, \mathsf{keytype}, \mathsf{pk}_n, \mathsf{pk}_f, \sigma_n, \sigma_f)$ to the blockchain, where σ_n and σ_f are signatures on $(\mathsf{id}, \mathsf{revoke}, \mathsf{keytype})$ under the online secret key sk_n and offline secret key sk_f, respectively.

However, unless both the online and the offline secret keys are simultaneously compromised, revocation is not necessarily the only course of action available to Certcoin users. Instead, an identity owner can update his or her compromised key by posting an update signed by both the online and offline secret keys to the blockchain. Depending on the signatures included, posts carry different weights. Any statement signed by both secret keys outweighs any statement signed by only one. For instance, if the online secret key sk_n is compromised but the offline secret key sk_f is not, the identity owner can generate a new online key pair, and update pk_n by issuing an update signed by both keys. Even if an adversary issues a similar update signed only by the online secret key sk_n which he has stolen, the identity owner's update will supercede the adversary's update. If both keys are compromised, the identity owner must revoke his identity and all associated public keys; the identity owner will never be able to reclaim his or her identity id securely, but at least he or she can make sure that no adversary will successfully impersonate id.

Note that in order to address secret key compromises in the manner described above, compromises must be detected first. If a compromise goes undetected for a significant period of time, an adversary has time to impersonate the legitimate identity owner, and even if in retrospect the impersonation is detected and revoked, the adversary may have already done some damage.

4.6 Recovering the Secret Key Corresponding to an Identity

When a user creates a key pair for an identity, Certcoin requires them to set up a recovery system where the secret key is secret shared (e.g. using the Shamir secret sharing paradigm [22]) among at least three trusted "friends", with a threshold of at least two for reconstruction. Furthermore, for improved security, each trusted friend should not know the identities of the others. This cannot be enforced; a user who really does not want to trust anyone but themselves can still satisfy these criteria by naming their "friends" to be several Certcoin accounts they themselves created. A similar technique is used by the Bitcoin wallet management platform Armory [9], where the key to the wallet is secret shared to enable recovery.

5 Certcoin Version 1: Efficient Verification

One major challenge in deploying Version 0 of Certcoin would be the necessity for every Certcoin user to store the entire blockchain. The Bitcoin blockchain is currently at 16GB, and it appears to be growing at a rate of approximately 1GB a month [1]. The Certcoin blockchain, once it enters popular use, would also be fairly large. Version 0 of Certcoin requires any device or entity performing verification to have a large storage capacity in order to verify that a given identity id and public key pk do indeed correspond to one another. However, that is not always possible; for instance, a browser on a smartphone might not have that much storage available to it.

We propose the use of *accumulators* [13] in Version 1 to lower the Certcoin storage and running time requirements for verification, making them logarithmic as opposed to linear in the number of identities. An accumulator is a data structure used for testing membership in a set in a more space-efficient way than simply storing all of the elements of the set. The accumulator would store tuples of the form (id, keytype, pk), where id is an identity, keytype is a public key type (either online or offline), and pk is a public key. One can then verify that an identity id and public key pk of type keytype correspond to one another simply by querying the accumulator on (id, keytype, pk). In this section, we describe cryptographic accumulators and how they can be used in Certcoin.

Accumulators The use of cryptographic accumulators as a decentralized alternative to digital signatures was first described in 1994 by Benaloh and de Mare [4]. An accumulator is a compact representation of a set of elements, which supports membership queries. Upon the addition of an element into the accumulator, a witness is generated that can then be used to prove that the element in question has been added. More formally, an accumulator scheme consists of four polynomial-time algorithms:

- $\mathsf{AccGen}(1^k) \to a$ generates the initial value of the empty accumulator, as well as any additional parameters, given the security parameter k.
- $\mathsf{AccAdd}(a, y) \to (a', w)$ takes in the current state of the accumulator a and the value to be added y, and returns the new state of the accumulator a' as well as the corresponding witness w.
- $\mathsf{AccWitAdd}(w, y) \to w'$ takes in the current state of a witness w and the new value y being added to the accumulator, and returns an updated witness w'.
- $\mathsf{AccVer}(a, y, w) \to \{0, 1\}$ takes in the current state of the accumulator a, the value y whose membership in a is being checked, and the witness w that y is in a, and returns 1 if y appears to be in a, and 0 otherwise.

The security properties of accumulators are described in Appendix 9. The following are some additional properties which might be desired from accumulators.

- **Compactness:** A desirable property of accumulators is that they remain small, no matter how many items are added to them. An accumulator is *compact* if its size is constant (i.e., independent of the number of elements it contains). Some accumulators grow logarithmically with the number of elements they contain.
- **Dynamism:** In 2002, Jan Camenisch and Anna Lysyanskaya [8] introduced the notion of *dynamic accumulators*, which support deletion of elements from the accumulator by means of a deletion algorithm AccRem, and a witness update algorithm AccWitRem.
- **Universality:** In 2007, Jiangtao Li, Ninghui Li and Rui Xue [18] introduced the notion of *universal accumulators*, which are accumulators supporting non-membership proofs in addition to membership proofs.
- **Strength:** In 2008, Philippe Camacho, Alejandro Hevia, Marcos Kiwi and Roberto Opazo [6] introduced the notion of *strong accumulators*, which do not assume that the accumulator manager is trusted. Strong accumulators cannot use trapdoor information in the creation or maintenance of the accumulator, as done in the RSA accumulator [4]. The Merkle Hash Tree accumulator [6] described later in this section is an example of a strong accumulator.

- **Public Checkability:** An accumulator is *publicly checkable* if the correctness of every operation - e.g. an accumulator update - can be publicly verified. All of the accumulators mentioned in this section are publicly checkable.

Accumulator Constructions There are many known accumulator constructions, including the RSA construction, the Bilinear Map construction, and the Merkle Hash Tree construction. In addition to these accumulators, we also consider the Bloom Filter, which, while technically not a cryptographic accumulator, can also be used for efficient membership testing. The table in Figure 4 summarizes some properties of all of these constructions.

Accumulator	Accumulator Size	Witness Size	Witness -Free?	Dynamic?	Universal?	Strong?	Publicly Checkable?
RSA [4]	$O(1)$	$O(1)$	no	yes	yes [18]	no	yes
Bilinear Map [7]	$O(1)$	$O(1)$	no	no	no	no	yes
Merkle [6]	$O(\log(n))^2$	$O(\log(n))$	no	yes	yes[3]	yes	yes
Bloom Filter [5]	$O(n)$	N/A	yes	no	N/A	yes	yes

Figure 4: Various Accumulators and their Properties

Using Accumulators in Certcoin There are two possible ways to integrate the use of accumulators into our blockchain-based PKI: either each user can maintain their own accumulator, or there can be a single accumulator maintained in the blockchain.

If each user maintains their own accumulator locally, witnesses cannot be used. This is because if each accumulator holder issues each public key holder a witness, all this accomplishes is shifting the storage burden from the accumulator holders to the key holders, as well as creating a large amount of communication overhead. The witness-free requirement naturally suggests the use of Bloom Filters. Assuming that no more than 10^9 certificates exist at any given point in time, only $10^{10}b = 1.16G$ of storage is required to guarantee negligible probability of false positives. This is ten bits per certificate - much less than storing an outright list, but still too much for lightweight clients like a smartphone browser.

Because of this issue, we propose storing one global accumulator in the blockchain. Each time a public key is created, updated or revoked, the block miner who processes this transaction will update the accumulator, and include in the block the updated accumulator value, the information added to the accumulator, and any corresponding witnesses. Because of the public checkability property of all of the accumulators under consideration, users can check that the updated accumulator correctly incorporates the new values, and that the witnesses were correctly computed.

If a single accumulator is stored and maintained in the block chain, then it is important that the accumulator be strong, so that parties instantiating the accumulator cannot cheat. We propose using the Merkle Hash Tree accumulator, which is strong as well as dynamic, thus directly supporting key revocations. However, the Merkle Hash Tree accumulator is not compact. The accumulator and the associated witnesses will be of logarithmic instead of constant size. Because the number of public keys a PKI manages is likely bounded by a constant times the world's population, this is not terribly problematic. Assuming the world's population is twice what it is now, its logarithm will still not exceed 34.

[3]For the author's modification of the [6] scheme.

[3]This comes at some additional memory cost.

Details of The Merkle Hash Tree Accumulator In this section, we describe an approach based on, but slightly different form, the [6] construction. Let h be a collision-resistant hash function. A Merkle hash tree accumulator can maintain a list of $\log(n)$ balanced Merkle hash tree roots r_i, at most one with associated tree depth i for $i \in [1, \ldots, \log(n)]$ where n is the number of elements in the accumulator. Let $a = [r_1, \ldots, r_{\log(n)}]$. A witness for y consists of d and elements $e_1, \ldots, e_{d-1}$ such that $h(h(\ldots(h(h(y)||e_1)||e_2)\ldots)||e_{d-1}) = r_d$, where $||$ denotes concatenation. In other words, a witness is a Merkle hash tree path to one of the $\log(n)$ tree roots stored in the accumulator.

This accumulator can be made dynamic by introducing another publicly checkable algorithm $\mathsf{Del}(a, y, w)$ that replaces the node in the tree corresponding to y with $\perp$, and updates the corresponding Merkle Hash Tree path. WitDel can be implemented by updating the witness Merkle Hash Tree path to make it consistent with the new tree, which can be done given the deleted element witness. By keeping a list of deletions and their corresponding Merkle Hash Tree paths, the space vacated by deletions can even be reclaimed in subsequent additions.

Figure 5 describes the maintenance of an accumulator during every Certcoin operation, as well as verification by means of the accumulator.

6 Certcoin Version 2: Efficient Lookup

While Version 1 of Certcoin makes the verification of a public key more time- and space-efficient, the lookup of a public key corresponding to a given identity id would still require a traversal of the entire blockchain. In order to be a practical PKI, it is clear that Certcoin must support public key lookup more efficiently. We propose the use of an authenticated distributed hash table (DHT), effectively creating a resilient, decentralized key-server maintained by the Certcoin users. Certcoin's distribution mechanism exploits the efficiency of the Kademlia DHT [19] to create a self-sustaining key-delivery service. The Certcoin DHT will have an entry corresponding to each identity-keytype $(\mathsf{id}, \mathsf{keytype})$ pair, containing the corresponding public key pk and accumulator witness w, the use of which is described in Section 5. The DHT nodes will need to update the stored witness values whenever the accumulator is updated.

In standard form, Kademlia is unauthenticated and thus susceptible to poisoning, routing, and Sybil attacks. Fortunately, we can easily protect against poisoning and routing by requiring Kademlia nodes to authenticate all of their communication using Certcoin's authentication protocol, described in 7. Authentication also discourages Sybil attacks, since there is some small cost associated with registering an identity, and authentication requires that any node participating in the DHT hold a valid identity.

Additionally, we need to give our nodes incentive to participate in the DHT. In order to prevent nodes from simply removing themselves from the DHT network and leaving the rest of the network to distribute its keys, we require any DHT node K answering a key lookup query to respond only if it is able to receive a response to a heartbeat message from the node I associated with the key. This heartbeat message should be signed, so that K cannot skip the heartbeat check without detection. Nodes have an incentive to respond to the heartbeat messages, since otherwise users will not be able to retrieve their public key.

Note that having retrieved the public key pk and witness w corresponding to $(\mathsf{id}, \mathsf{keytype})$, a user can verify that the public key pk is correct using the Version 1 verification procedure described in Section 5. However, we do not currently have the infrastructure in place for a user to verify negative responses. In other words, if the lookup operation returned $\perp$, a user cannot verify that id really

Accumulator maintenance:
(Note that this accumulator maintenance is performed *in addition* to the registration, update and revocation steps described in Section 4, not instead of them.)

1. Accumulator creation:
   - The block miner creates the accumulator: $a \leftarrow \mathsf{AccGen}(1^k)$. He or she includes a in the newly mined block.
   - All block recipients verify that a was created correctly. If it was, they store a; otherwise, they discard the block.
2. Registration of $(\mathsf{id}, \mathsf{keytype}, \mathsf{pk})$:
   - The block miner adds $(\mathsf{id}, \mathsf{keytype}, \mathsf{pk})$ to the accumulator, thus registering the public key: $a, w \leftarrow \mathsf{AccAdd}(a, (\mathsf{id}, \mathsf{keytype}, \mathsf{pk}))$.
   - The block miner includes $(\mathsf{id}, \mathsf{keytype}, \mathsf{pk}, a, w)$ in the newly mined block.
   - All block recipients verify that a was updated correctly. If it was, they update their stored accumulator values with a, and update their stored witnesses: $w \leftarrow \mathsf{AccWitAdd}(w, (\mathsf{id}, \mathsf{keytype}, \mathsf{pk}))$. Otherwise, they discard the block.
   - The identity owner verifies that w is correct, and stores it.
3. During update of $(\mathsf{id}, \mathsf{keytype}, \mathsf{pk}^{old}, \mathsf{pk}^{new})$ using witness w:
   - Carry out the accumulator operations described in Item 4 for $(\mathsf{id}, \mathsf{keytype}, \mathsf{pk}^{old})$ using w.
   - Carry out the accumulator operations described in Item 2 for $(\mathsf{id}, \mathsf{keytype}, \mathsf{pk}^{new})$.
4. During revocation of $(\mathsf{id}, \mathsf{keytype}, \mathsf{pk})$ using witness w:
   - The block miner verifies that $(\mathsf{id}, \mathsf{keytype}, \mathsf{pk})$ is in the accumulator using the procedure described below. If verification fails, the block miner aborts.
   - The block miner removes $(\mathsf{id}, \mathsf{keytype}, \mathsf{pk})$ from the accumulator, thus revoking the public key: $a \leftarrow \mathsf{AccRem}(a, (\mathsf{id}, \mathsf{keytype}, \mathsf{pk}))$
   - The block miner includes a in the newly mined block.
   - All block recipients verify that a was updated correctly. If it was, they update their stored accumulator values with a, and update their stored witnesses: $w \leftarrow \mathsf{AccWitRem}(w, (\mathsf{id}, \mathsf{keytype}, \mathsf{pk}))$. Otherwise, they discard the block.

Verification that a given public key pk of type $\mathsf{keytype}$ corresponds to a given identity id is now very simple given the accumulator a (extracted from the latest Certcoin block) and the corresponding witness w (provided by the party whose identity is being verified):

- Given $(\mathsf{id}, \mathsf{keytype}, \mathsf{pk})$, accumulator a and the witness w, verify that $\mathsf{AccVer}(a, (\mathsf{id}, \mathsf{keytype}, \mathsf{pk}), w) = 1$.

Figure 5: (V1) Accumulator maintenance assuming use of the Merkle Hash Tree accumulator, and the Certcoin Version 1 procedure for verifying that a given public key and identity correspond

does not currently have an associated key of type keytype. In order to support such verifications, we will use an additional strong universal accumulator [18]. In this accumulator a_n, we will store tuples of the form $(\mathsf{id}, \mathsf{keytype})$; a proof that an identity id does not have a current public key of type keytype is then simply a proof that $(\mathsf{id}, \mathsf{keytype})$ is not in a_n. In order to update accumulator a_n, storage linear in the number of entries is required. However, the accumulator itself is constant in size, and witnesses are logarithmic in size, so verifications can still be performed space-efficiently. We are not too worried about the fact that entities performing the updates will need a substantial amount of storage. These entities will be the blockchain miners, and it is reasonable to assume that they are not smartphones, and have sufficient storage available. We do not describe the maintenance of accumulator a_n here, in the interest of brevity. This maintenance closely parallels that of the accumulator a, which is described in Figure 5.

Lookup:

1. The querying party Q sends a query of the form $(\mathsf{id}, \mathsf{keytype}, \mathsf{h})$, where:
   - id is the identity the public key associated with which he or she wishes to retrieve,
   - keytype is the key type (either online or offline), and
   - h is a random heartbeat message.
2. The DHT node K responsible for key queries for id determines whether id has an associated public key of type keytype. If it does not:
   - K sends $(\perp, w)$ to Q, where $\perp$ is a null value and w is an accumulator witness for a_n that proves that $(\mathsf{id}, \mathsf{keytype})$ is not in a_n.
   - Q verifies that $(\mathsf{id}, \mathsf{keytype})$ is not in a_n using w.

   If it does:
   - K sends sends h to the node I with identity id.
   - I sends $\sigma = \mathsf{sig}(\mathsf{sk}_n^I, \mathsf{h})$ to K, where sk_n^I is I's online secret key.
   - K acquires (pk_n^I, w) from its database, where pk_n^I is I's online public key, and w is an accumulator witness for a as described in Section 5.
   - K checks that $\mathsf{ver}(\mathsf{pk}_n^I, \sigma, \mathsf{h}) = 1$, and if this holds, sends $(\mathsf{pk}_n^I, w, \sigma)$ to Q.
   - Q does the following:
     - Verifies (using the witness w in the protocol described in Section 5) that pk_n^I corresponds to id, and
     - Verifies that $\mathsf{ver}(\mathsf{pk}_n^I, \sigma, \mathsf{h}) = 1$.

Figure 6: (V2) Looking up the public key corresponding to an identity

Performance critical applications should make use of Kademlia's support for parallel asynchronous queries by requesting the same public key from a handful of different nodes in the DHT. This way, they will experience lower expected latencies, since at least one of their queries is more likely to be routed along an un-congested path.

Future Work Note that while the nodes in the DHT are not able to mislead users making lookup queries, they are able to deny service by ignoring lookup requests. One interesting extension of Certcoin might be a reputation-based (or financial incentive-based) system which would discourage nodes from doing so.

7 Authentication: Putting it All Together

We have described in Sections 4.4 and 4.3 (for Version 0) and Sections 5 and 6 (for Versions 1 and 2) how one can verify and look up public keys corresponding to given identities. Now, we can take a look at the big picture: that is, if Alice is communicating with Bob, how can she leverage the Certoin infrastructure to prove to Bob that she is, indeed, Alice? In this, Certcoin does not differ from any other PKI.

Alice sends Bob a message of the form $(\mathsf{id}^A, \mathsf{pk}_n^A, w)$, where id^A is Alice's identity, pk_n^A is Alice's online public key, and w is an accumulator witness as described in Section 5. Bob verifies that pk_n^A does indeed correspond to id^A. He then sends Alice a random challenge message h; if Alice can respond with $\sigma = \mathsf{sig}(\mathsf{sk}_n^A, \mathsf{h})$ such that $\mathsf{ver}(\mathsf{pk}_n^A, \sigma, \mathsf{h}) = 1$, Bob believes that she is, indeed, the owner of id^A. So, Alice has authenticated successfully.

8 Conclusion

In conclusion, we believe that Certcoin is a viable PKI offering better identity retention guarantees than either Certificate Authorities or PGP Webs of Trust. Our construction benefits from an entirely decentralized architecture offering inherent fault tolerance, redundancy, and transparency. At the same time, Certcoin supports the expected features of a full-fledged Certificate Authority, including public key registration, update, revocation and recovery, as well as public key lookup and verification. We plan to implement Certcoin so as to demonstrate the viability of this idea.

9 Acknowledgements

We would like to thank Professor Ron Rivest. Certcoin was first designed as a final project for his Computer and Network Security class, and he has offered us encouragement and advice throughout the process of turning this into a paper.

We would also like to thank Professor Leo Reyzin for his support and patient feedback.

References

[1] Bitcoin blockchain size, http://blockchain.info/charts/blocks-size.

[2] Namecoin, https://www.namecoin.org/.

[3] Certificate transparency, http://www.certificate-transparency.org/.

[4] Josh Benaloh and Michael de Mare. One-way accumulators: A decentralized alternative to digital signatures. In *Workshop on the Theory and Application of Cryptographic Techniques*

on Advances in Cryptology, EUROCRYPT '93, pages 274–285, Secaucus, NJ, USA, 1994. Springer-Verlag New York, Inc.

[5] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[6] Philippe Camacho, Alejandro Hevia, Marcos 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*, volume 5222 of *Lecture Notes in Computer Science*, pages 471–486. Springer Berlin Heidelberg, 2008.

[7] Jan Camenisch, Markulf Kohlweiss, and Claudio Soriente. An accumulator based on bilinear maps and efficient revocation for anonymous credentials. In Stanisaw Jarecki and Gene Tsudik, editors, *Public Key Cryptography PKC 2009*, volume 5443 of *Lecture Notes in Computer Science*, pages 481–500. Springer Berlin Heidelberg, 2009.

[8] Jan Camenisch and Anna Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *CRYPTO*, pages 61–76, 2002.

[9] Armory Developers. Version: 0.90-beta (26 nov, 2013).

[10] Tom Espiner. Trustwave sold root certificate for surveillance, 2012.

[11] D. Cooper et al. Internet x.509 public key infrastructure certificate and certificate revocation list (crl) profile, 2008.

[12] S. Santesson et al. X.509 internet public key infrastructure online certificate status protocol - ocsp, 2013.

[13] Nelly Fazio and Antonio Nicolosi. Cryptographic accumulators: Definitions, constructions and applications.

[14] Dennis Fisher. Final report on diginotar hack shows total compromise of ca servers, 2012.

[15] Christina Garman, Matthew Green, and Ian Miers. Decentralized anonymous credentials. *IACR Cryptology ePrint Archive*, 2013:622, 2013.

[16] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks, 1988.

[17] Walter Goulet. Understanding the public key infrastructure behind SSL secured websites.

[18] 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*, volume 4521 of *Lecture Notes in Computer Science*, pages 253–269. Springer Berlin Heidelberg, 2007.

[19] Peter Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric.

[20] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[21] Konstantin Ryabitsev. PGP web of trust: Core concepts behind trusted communication.

[22] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, November 1979.

A Accumulator Security Properties

An accumulator is *secure* if it has the following properties:

- **Correctness:** An up-to-date witness corresponding to value y can always be used on an up-to-date accumulator to verify the membership of y.

  More formally, for all valid values y and additional sets of valid values $[y_1, \ldots, y_{l_1}]$, $[y_{l_1+1}, \ldots, y_{l_2}]$,

$$\begin{aligned} &\Pr[a \leftarrow \mathsf{AccGen}(1^k); \\ &(a, w_{new}) \leftarrow \mathsf{AccAdd}(a, y_i) \text{ for } i \in [1, \ldots, l_1]; \\ &(a, w) \leftarrow \mathsf{AccAdd}(a, y); \\ &((a, w_{new}), w) \leftarrow (\mathsf{AccAdd}(a, y_i), \mathsf{AccWitAdd}(w, y_i)) \text{ for } i \in [l_1 + 1, \ldots, l_2] : \\ &\mathsf{AccVer}(a, y, w) = 1] = 1 \end{aligned} \tag{1}$$

- **Soundness:** It is hard to fabricate a witness w for a value y that has not been added to the accumulator in such a way that the verification of y's membership succeeds.

  More formally, for any probabilistic polynomial-time adversary $\mathbb{A}$ with black-box access to Add and WitAdd oracles on a,

$$\begin{aligned} &\Pr[(y, w) \leftarrow \mathbb{A}^{\mathsf{AccAdd}, \mathsf{AccWitAdd}}(k, a); \\ &\quad y \text{ has not been added to } a : \\ &\quad \mathsf{AccVer}(a, y, w) = 1] = negl \end{aligned} \tag{2}$$

  Where y is an element $\mathbb{A}$ has not called AccAdd on, a is the state of the accumulator after the adversary made all of his calls to AccAdd, and $negl$ is a negligible function in the security parameter.