CertCoin:
A NameCoin Based Decentralized Authentication System
6.857 Class Project

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

May 14, 2014

1 Abstract

Authentication is vital to all forms of remote communication. A lack of authentication opens the door to man-in-the-middle attacks, which, if performed at a key moment, may subvert the entire interaction. Current approaches to authentication on the internet include certificate authorities and webs of trust. Both of those approaches have significant drawbacks: the former relies upon trusted third parties, introducing a central point of failure, and the latter has a high barrier to entry. We propose Certcoin, an alternative, public and decentralized authentication scheme. The core idea of Certcoin is maintaining a public ledger of domains and their associated public keys. We describe the Certcoin scheme, as well as several optimizations to make Certcoin more accessible to devices with limited storage capacity, such as smartphones. Our optimizations use tools such as *cryptographic accumulators* and *distributed hash tables*.

2 Existing Approaches to Authentication

Prior to the advent of public-key cryptography in 1976, most secure message transfer was accomplished by means of symmetric encryption protocols that relied on both parties having previously shared some secret key K. Since it is highly unlikely that an adversary would be able to guess the secret key, authentication in these schemes is trivial. Any individual encrypting messages with K is considered to be an authenticated user. However, exchanging keys before communication is impractical for dynamic environments such as the Internet, where large numbers of entities must have the ability to join and leave the network at will, and to securely communicate with any other entity on the network. Having each pair of entities establish their own shared key quickly becomes infeasible as the number of entities increases [11].

Public key cryptography addresses this scalability issue; however, it makes authentication much less trivial. In traditional public key systems, users retrieve the public key of the intended recipient of their message, encrypt under that key, and sign the message with their own signing key. The recipient is then the only one able to decrypt the message, since he is the only holder the secret decryption key. This setup eliminates the need for *a priori* key agreement, but introduces the problem of needing to authenticate the sender, since all entities are now equally capable of producing the encryption. He is able to verify the public key of the sender based on the digital signature, but a public key alone cannot be used to determine the sender's identity [21].

The notion of a Public Key Infrastructure (PKI) was born from this necessity to verify the identity of a message sender. A PKI typically issues certificates attesting to the public key of each entity. In addition, a PKI has the ability to revoke certificates in the event that a public key becomes compromised, whether it be by loss of the secret key or implication of the associated individual as malicious. Currently, the two most commonly employed approaches fall into two categories: centralized, trusted third-party Certificate Authorities (CA), and decentralized networks of peer-to-peer certification, often referred to as Webs of Trust (as coined by Phil Zimmerman, the creator of PGP).

2.1 Certificate Authorities

Typically the more common choice in practice, a Certificate Authority (CA) acts a trusted third party that is responsible for delivering and managing digital certificates for a network of users. In order to do so, the CA must have some sort of registration process whereby a user's identity is verified, they are assigned a Distinguished Name (DN, e.g. Google or Facebook), and their public keys are recorded along with their DN. These records contain an expiration date, as well as an indication of the key's purpose, whether it be for encrypting data or verifying a signature. The CA then serves two purposes: it facilitates verification that a user holds a certain public key, and it facilitates the look-up of public keys corresponding to users. Verification is done by means of a *certificate* issued to each user; that certificate is a statement of the form "user x holds public key PK", signed by the CA. Look-up is done by allowing any user to request the public keys of any other individual. The requestor can then verify that the individual holds the corresponding secret key by executing a zero-knowledge proof of knowledge with that individual. The CA ultimately acts as an agent of trust for the network, since anyone trusting in the CA should also trust the certificates it provides [16].

Many entities may wish to have their secret keys securely backed up in the event that a secret key

is lost; the process by which this is done is referred to as key recovery, and is a feature offered by many of today's CAs. The loss of a decryption key could result in the permanent loss of sensitive or otherwise important data. On the other hand, the loss of a signing key is relatively insignificant. Messages signed in the past can still be verified with the public key (assuming it wasn't revoked), and the user can simply obtain a new key pair to continue signing. Additionally, backing up a signing key with the CA would allow the CA to impersonate the key owner. Thus, it makes most sense for a CA to back up decryption keys, but not signing keys [13]. This makes it necessary for every user to create *two* key pairs - one for encryption and the other for signatures - instead of using the same key pair for both functionalities.

Recently, there have been numerous incidents showing that too much trust is being placed in CAs. CAs have been hacked (e.g. the DigiNotar incident), and have even issued subordinate root certificates to customers (e.g. the TrustWave incident), allowing the customer to themselves issue certificates.

One approach to preventing such incidents is introducing transparency into the workings of the CAs. The Google Certificate Transparency project [3] attempts to do 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 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.

2.2 Web of Trust

The second major PKI used in practice is the PGP Web of Trust (WoT). In this system, authentication is entirely decentralized; users are able designate others as trustworthy by signing their public keys. By doing so, a user accumulates a certificate containing his public key and digital signatures from entities that have deemed him trustworthy. The certificate is then trusted by a party if they are able to verify that the certificate contains the signature of someone he or she trusts [21].

This system benefits from its distributed nature because it removes any central point of failure. However, it makes it difficult for new or remote users to join the network, since they must typically meet with someone in person to have their identity verified and public key signed for the first time. Also, unlike a CA, the WoT also has no way to deal with key recovery. A user is limited to choosing another user to be their "designated revoker", tasked with revoking their certificate when the private key is lost or compromised. Despite these set backs, the Web of Trust has successfully been in operation for almost a quarter century without significant modification.

2.3 Certcoin: the Best of Both Worlds

We introduce Certcoin, a system that incorporates the best aspects of transparent Certificate Authorities and of the Web of Trust. Like Google's Certificate Transparency project, Certcoin is completely public and auditable. Like the WoT, Certcoin is decentralized, and does not have a single point of failure. We provide the details of the Certcoin protocol in Section 3.

3 The Fundamental Certcoin Protocol

We propose a new, decentralized alternative public key infrastructure, based on Bitcoin. The ability to publish a public key corresponding to a given identity or domain in a reliable, permanent way would provide a trivial way to authenticate. Bitcoin and other cryptocurrencies implement exactly such a bulletin board.

We propose to build Certcoin on top of Namecoin [2] by branching the project and taking advantage of the merged mining protocol to ensure that Certcoin transactions are constructed properly in its blockchain. Namecoin is a cryptocurrency designed to act as a decentralized DNS for .bit addresses. It has three types of transactions, which support the registration and update of domains: name_new, name_firstupdate and name_update. name_new generally costs 0.01 units of Namecoin, while the others are free.

Throughout this paper, we treat the Namecoin blockchain as a public "bulletin boards", to which information can be posted in a permanent way. Certcoin transactions are posted to that bulletin board. A transaction fee is paid for every such post, much like in Bitcoin, as an incentive for block miners to include Certcoin information.

3.1 Registering a Domain with Certcoin

When a domain is initially registered in Namecoin, the transaction contains signed information about two public keys that 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.

The “online” secret key is used to authenticate messages to and from the server hosting the website. This secret key is also the primary key used to authenticate the website in Certcoin; a proof on knowledge of that key will be required as part of authentication.

The “offline” secret key is stored in a secure place offline that won’t be vulnerable to the same security threats that the online key might face as a result of being stored on a device connected to the internet. This offline key is to be used primarily to sign or revoke new keys in case of a security break or key compromise. This will be explained in further detail in Section 4.2.

When a new key is to created for the domain, it will be signed with the old key of the same type. This way, at any point, both valid keys registered to the domain can be traced back through a series of signed statements to the initial two key pairs registered to the domain.

3.2 Updating a Public Key Corresponding to a Domain

Changing the public key pk_{old} to a new public key pk_{new} (of the same online / offline type) corresponding to domain d is done by posting

$$(d, \sigma((d, pk_{new}), sk_{old}))$$

to our bulletin board, where σ is a secure digital signature algorithm. 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 pk_{old}.

3.3 Verifying / Looking Up a Public Key

Verifying that a public key pk (of either the offline or online type) corresponds to a domain d involves traversing the block chain, and checking that:

- the domain d has been registered exactly once,
- all signatures in subsequent updates to the public key corresponding to d verify with d's previous public key,
- the last update resulted in pk being the public key corresponding to d, and
- the user claiming to be the owner of domain d knows the secret key sk corresponding to pk, using a zero-knowledge proof of knowledge.

Looking up a public key pk (of either the offline or online type) corresponding to a domain d involves traversing the block chain and doing the following:

- Check that the domain d has been registered exactly once.
- Check that all signatures in subsequent updates to the public key corresponding to d verify with d's previous public key.
- Find the last update to the public key corresponding to d, and store the public key pk that update resulted in.
- Check that the user claiming to be the owner of domain d knows the secret key sk corresponding to pk, using a zero-knowledge proof of knowledge. If the zero-knowledge proof of knowledge is valid, then return pk. Else, return $\perp$.

4 Key Recovery and Revocation

A fully functional certificate system must provide some functionality for revoking and recovering keys. A certain secret key may be compromised or stolen, or simply expire, in which case the owner of the key would need to revoke the old key and instate a new key. Similarly, a password could be forgotten and certain methods of recovery would be necessary. In a traditional certificate authority system, one could simply call customer support to achieve either of these objectives; however, in our decentralized system, a mechanism for revocation and recovery must be built into the machinery from the start.

4.1 Key Recovery

When a user creates a secret / public key pair for a domain, 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. A user who really does not want to trust anyone but themselves can still achieve these security

goals by naming their "friends" to be several Certcoin accounts they themselves created. A similar technique is used by the Bitcoin wallet management platform Armory [12], where the key to the wallet is secret shared to enable recovery.

4.2 Key Revocation

In traditional public key infrastructures, certificates either expire at a certain age or are revoked either by being added to a Certificate Revocation List (CRL) [10] or by means of the Online Certificate Status Protocol (OCSP) [14].

Certcoin can establish a *lifetime* value; each public key will have an associated timestamp, and if a public key was created more than a lifetime ago, it should rejected based on age alone. The public key can also be manually revoked with a reason code, much like in a traditional PKI. Reason codes can be one of "Unspecified" , "Key Compromise" , "CA Compromise" , "Affiliation Changed" , "Superseded" , or "Cessation of Operation".

At all times, each domain has two associated secret keys - one online, and one offline. Different states of compromise can be responded to in separate ways. We make distinctions between the adversary gaining access to a key, and the adversary stealing a key (i.e., taking access to the key away from the domain owner).

- **If the adversary obtains access to only one secret key:** the real domain owner still has access to both of his secret keys, and can use both keys to sign statetments revoking the sole compropised key and any other keys it may have tried to sign. Next, the domain owner will update the compromised key using the current two key pairs. Lastly, both the old secret keys will be used to sign a statement invalidating all future operations of the compromised key. This way, the real domain owner is again the sole owner of two uncompromised keys, and the adversary only has access to an invalidated key. All these statements are posted as transactions in the blockchain, and are publicly verifiable. Due to the fact that many statements can appear in one block, all new keys signed by a key which is being revoked in the same block (and not signed by its sister key) are automatically invalid regardless of their timestamp within the block. This is done so that the adversary can't sneak in a new key that it controls right as the old one is being revoked.

- **If the adversary steals the online secret key:** The domain owner loses all access to the online secret key, and the adversary manages to prevent the domain owner from recovering the key through the key recovery system. All is not lost, because the offline secret key always has veto power over any signatures produced by the online secret key. The domain owner can use the offline secret key to sign a statement invalidating all keys signed by the compromised online key, and revoking the compromised key itself. Next, the domain owner can use the offline secret key to create a new online secret key. He then returns the offline secret key to secure offline storage.

- **If the adversary obtains access to both the online and offline keys:** The domain owner still has access to both keys, he can use both to sign a statement invalidating all present and future statements signed by these keys, since there is now no way to distinguish the adversary from the real domain owner.

- **If the adversary steals both secret keys:** The domain owner no longer has access to his keys, and there is no way to address such a situation within the Certcoin protocol.

Note that in order to address secret key compromises in the manners described above, compromises must be detected first.

5 Cutting Down on Storage: Accumulators for Key Verification

One major challenge in deploying 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]. A naive implementation of Certcoin would require any device or entity performing verification to have a large storage capacity. However, that is not always possible; for instance, a browser on a smartphone might not have that much storage available to it.

We can do slightly better by only maintaining the current state of the registered domains and keys (i.e., a map from domain to current key). This still requires storage space that is linear in the number of registered domains, but it only requires constant time per check, unlike a traversal of the entire blockchain. We would also need to store the last few blocks of the block chain, so as to be able to verify further blocks using the Bitcoin protocol [20]. The rest of the blocks would be thrown away, after any public key updates they contain are verified, and after their contents are used to update the stored state. However, storage space linear in the number of registered domains is still wildly impractical - a browser on a smart phone would not be able to support this.

We propose the use of *accumulators* [15] to lower the Certcoin storage requirements. An accumulator is a digital object used for testing membership in a set. The accumulator would store tuples of the form (d, pk) or of the form (d, pk, exp), where d is a domain, pk is a public key, and exp is an optional expiration date. The accumulator could then be used to determine whether pk is registered to d; knowledge of the corresponding secret key sk would still need to be proven via a zero-knowledge proof. In this section, we describe cryptographic accumulators and how they could be used in Certcoin.

5.1 Cryptographic Accumulator Definitions

The use of cryptographic accumulators as a decentralized alternative to digital signatures was first described in 1994 by Benaloh and de Mare [5]. An accumulator is a constant-sized representation of a set of elements. 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:

- $\mathsf{Gen}(1^k) \rightarrow a$ generates the initial value of the empty accumulator, as well as any additional parameters, given the security parameter k.
- $\mathsf{Add}(a, y) \rightarrow (a', w)$ 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.
- $\mathsf{WitAdd}(w, y) \rightarrow w'$ 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'.
- $\mathsf{Ver}(a, y, w) \rightarrow \{0, 1\}$ 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.

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 $[y_1, \ldots, y_{l_1}]$, $[y_1, \ldots, y_{l_2}]$,

$$\begin{aligned} \Pr[a \leftarrow \mathsf{Gen}(1^k); \\ (a, w_{new}) \leftarrow \mathsf{Add}(a, y_i) \text{ for } i \in [1, \ldots, l_1]; \\ (a, w) \leftarrow \mathsf{Add}(a, y); \\ ((a, w_{new}), w) \leftarrow (\mathsf{Add}(a, y_i), \mathsf{WitAdd}(w, y_i)) \text{ for } i \in [1, \ldots, l] : \\ \mathsf{Ver}(a, y, w) = 1] = 1 \end{aligned} \tag{1}$$

- **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,

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

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

Some additional properties 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 [9] introduced the notion of *dynamic accumulators*, which support deletion of elements from the accumulator by means of a deletion algorithm Del, and a witness update algorithm WitDel.
- **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 [7] introduced the notion of *strong accumulators*, which are accumulators that 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 described in Section 5.2.1. The Merkle Hash Tree accumulator described in Section 5.2.2 is an example of a strong accumulator.

5.2 Cryptographic Accumulator Constructions

There are several known accumulator constructions, including the RSA construction, the Bilinear Map construction, and the Merkle Hash Tree construction. Since the properties of the Bilinear Map construction are similar to those of the RSA construction, we do not provide details about it in this paper. In addition to these accumulators, we also consider the Bloom Filter, which, while technically not a cryptographic accumulator since it does not leverage witnesses, can also be used for efficient membership testing. The table in Figure 1 summarizes some properties of all of these constructions.

Accumulator	Accumulator Size	Witness Size	Witness -Free?	Dynamic?	Universal?	Strong?
RSA [5]	$O(1)$	$O(1)$	no	yes	yes [18]	no
Bilinear Map [8]	$O(1)$	$O(1)$	no	no	no	no
Merkle [7]	$O(\log(n))$	$O(\log(n))$	no	yes	yes[1]	yes
Bloom Filter [6]	$O(n)$	N/A	yes	no	N/A	yes

Figure 1: Various Accumulators and their Properties

5.2.1 The RSA Accumulator

Benaloh and de Mare [5] proposed constructing accumulators from *quasi-commutative hash functions*, which are hash functions $h : X \times Y \to X$ such that, for all $x \in X$ and $y_1, y_2 \in Y$, it holds that

$$h(h(x, y_1), y_2) = h(h(x, y_2), y_1).$$

Since order does not matter in the application of this hash function, we let $h(x, \{y_1, y_2, \ldots, y_n\})$ denote $h(h(\ldots h(h(x, y_1), y_2) \ldots), y_n)$.

An accumulator can be built by starting with a fixed x, and applying h repeatedly as values y_i are added to the set. If h is one-way, then membership of a value y_i in the set can be tested in the accumulator $a = h(x, \{y_1, y_2, \ldots, y_n\})$ given a witness $w = h(x, \{y_j\}_{j \neq i})$ by checking that $h(w, y_i) = a$. This is clearly correct: if the value y_i was legitimately added to a, a valid $w = h(x, \{y_j\}_{j \neq i})$ would have been maintained. It is also sound; since h is one-way, the values w and y_i would be hard to obtain given only a.

Exponentiation in $\mathbb{Z}_n^*$ is one-way and quasi-commutative, and can be used to construct a cryptographic accumulator as follows:

Let v be a collision-resistant hash function.

- $\mathsf{Gen}(1^k)$:
  - choose a large integer $n = pq$ where p and q are both safe primes
  - select a generator g of $\mathbb{Z}_n^*$
  - return $x = g$
- $\mathsf{Add}(a, y)$:
  - set $w = a$
  - set $a' = a^{v(y)} \mod n$
  - return (a', w)
- $\mathsf{WitAdd}(w, y)$:
  - return $w' = w^{v(y)} \mod n$
- $\mathsf{Ver}(a, y, w)$:
  - if $w^{v(y)} = a$: return 1
  - else: return 0

The use of v is necessary to prevent the obvious witness fabrication attack for $y = y_i y_j$, where y_i and y_j have been legitimately added to the accumulator.

Note that this construction is only secure if the factorization of n is not known to the adversary; that is, the factorization of n is a *trap door*. However, this trap door is only necessary to create the accumulator, not to use the accumulator; the Add, WitAdd and Ver algorithms can all be carried out without knowing the factorization of n. So, while the RSA accumulator is not a strong accumulator because Gen uses trap-door information, it can be made strong by performing Gen in a secure fashion (possibly via multi-party computation), and only then giving the resulting accumulator to the accumulator manager.

Jan Camenisch and Anna Lysyanskaya [9] give a way to make the RSA accumulator dynamic. They describe a deletion algorithm Del and an additional witness update algorithm WitDel, as follows:

- $\mathsf{Del}(a, y)$:
  - compute $y' = y^{-1} \mod \phi(n)$, where $\phi(n) = (p-1)(q-1)$
  - set $a' = a^{y'} \mod n$
  - return a'
- $\mathsf{WitDel}(w, y)$:
  - compute $y' = y^{-1} \mod \phi(n)$, where $\phi(n) = (p-1)(q-1)$
  - set $w' = w^{y'} \mod n$
  - return w'

Note that Del and WitDel require the knowledge of the trap door, just like Gen.

5.2.2 The Merkle Hash Tree Accumulator

Jiangtao Li, Ninghui Li and Rui Xue [18] proposed constructing accumulators in a way similar to the building of Merkle Hash Trees. The construction given in their paper is made more complicated, and somewhat less efficient, by the fact that they designed it to be universal. Because we do not think that universality is a very important property for Certcoin accumulators, we informally present a slightly different, simpler version of the Merkle Hash Tree construction.

Let h be a collision-resistant hash function. This accumulator maintains a list of $\log(n)$ balanced Merkle Hash Tree roots, 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.

- $\mathsf{Gen}(1^k)$:
  - set $a = [\perp]$
- $\mathsf{Add}(a, y)$: Modify the set of Merkle Tree roots in the accumulator to include y, while maintaining the property that there are at most $\log(n)$ roots, and all of the associated trees are balanced. More formally,
  - set $w = []$
  - set $i = 1$
  - set $e = h(y)$
  - while $a[i] \neq \perp$:
    - if a has fewer than $i + 1$ elements: append $\perp$ to the end of a
    - set $e = h(e||a[i])$
    - append $a[i]$ to the end of w
    - set $a[i] = \perp$
    - set $i = i + 1$
  - set $a[i] = e$
  - return (a, w)
- $\mathsf{WitAdd}(a_{old}, w, y)$: informally, execute Add, and modify the witness path w accordingly. This will only involve appending an element to w. (Note that unlike the RSA construction, Add takes in the state of the accumulator before the addition.)
- $\mathsf{Ver}(a, y, w)$:
  - parse $[d, e_1, \ldots, e_{d-1}] = w$
  - if $h(h(\ldots(h(h(y)||e_1)||e_2)\ldots)||e_{d-1}) = a[d]$: return 1
  - else: return 0

Note that the Gen and Add functions described above are *publicly checkable*. Gen is simply by looking at the added element and the accumulator state before and after an addition, anyone can verify that addition was performed correctly.

Note also that 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!

5.2.3 The Bloom Filter

An alternative accumulator to consider is the Bloom Filter. It is not a typical cryptographic accumulator, in that it does not use witnesses, and the probability of erroneous verification goes up with the number of elements that are added. A Bloom Filter works as follows:

- $\mathsf{Gen}(1^k)$:
  - instantiate $\mathsf{BF} = 0^m$
  - select k hash functions $h_1, \ldots, h_k$
- $\mathsf{Add}(y)$:
  - for $i \in \{1, \ldots, k\}$, set $\mathsf{BF}[h_i(y)] = 1$
- $\mathsf{Ver}(\mathsf{BF}, y)$:
  - if for $i \in \{1, \ldots, k\}$, $\mathsf{BF}[h_i(y)] = 1$: return 1
  - else: return 0

Note that bloom filters have no false negatives, but they do have false positives. The probability of a false positive in the testing of membership in a Bloom Filter is:

$$p = (1 - (1 - \frac{1}{m})^{kn})^k$$

Where m is the number of bits used, n is the number of elements added, and k is the number of hash functions.

Figure 2 shows the probability of false positives in Bloom Filter checks for various values of m and n, with the optimal k already taken into account. The number of bits required for the bloom filter to remain secure is linear in the number n of elements added; however, only 10 bits per element are required, while the elements themselves can be arbitrarily large.

5.3 Using Accumulators in Certcoin

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

If each user maintains their own accumulator locally, witnesses cannot be used; if each Certcoin verifier issues each public key holder a witness, that simply shifts the storage burden from the

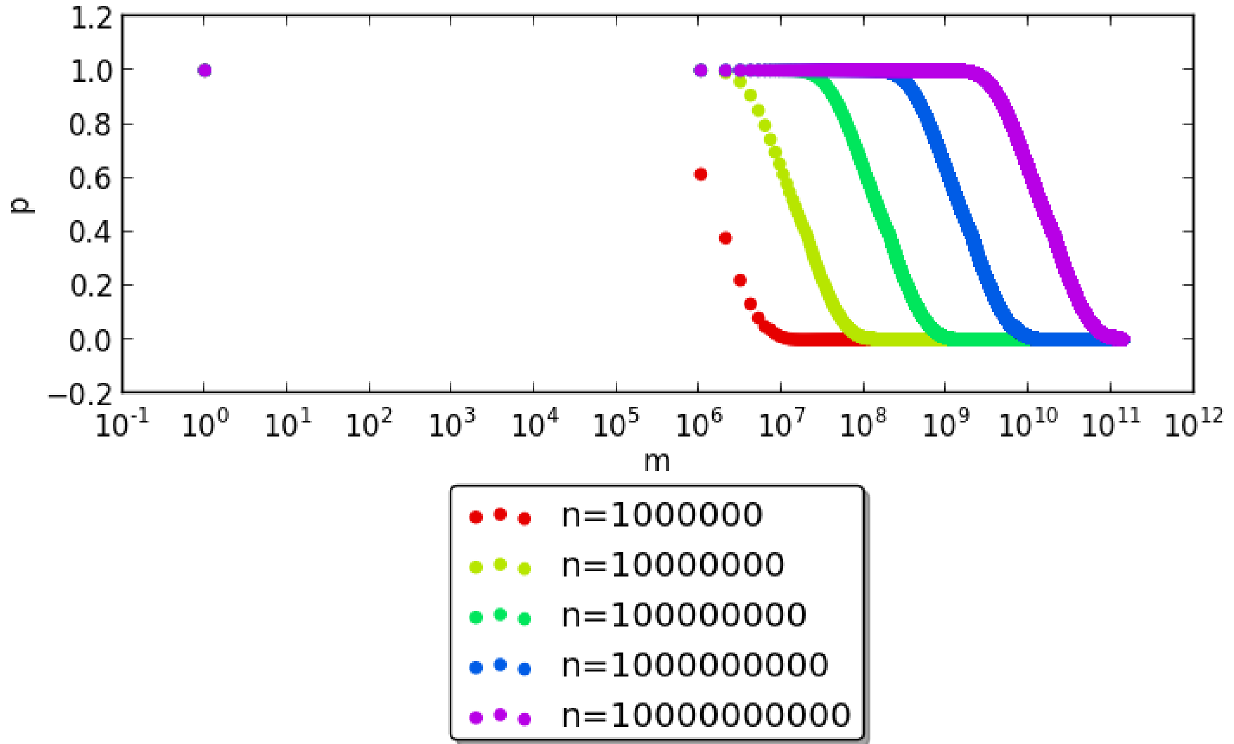

Figure 2: Probability of False Positives in a Bloom Filter Check

verifiers to the key holders. 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.

However, there is one major issue with locally stored accumulators: the validity of a new entry cannot be checked. Because the accumulator has no look-up feature, a duplicate entry for a single domain cannot be screened.

Because of this issue, we propose storing one global accumulator on the blockchain. We assume that entities participating in mining blocks are not computationally bounded; so, we can assume that they have the entire blockchain on hand, and can check that all broadcast values are either public keys for previously unregistered domains or are valid updates to registered domains before adding them to the accumulator.

Each time a public key is created or updated, the tuple to be added to the accumulator is broadcast, much like a transaction in Bitcoin. A single accumulator can be stored and maintained in the block chain; each time a Certcoin user is the one to mine a block, he will update the accumulator and include its updated value in the block (in addition to all new values he incorporated into it). All Certcoin users can check that the updated accumulator correctly incorporates the new values, and the individuals who broadcast the values can then compute their witnesses themselves, since all modifications to the accumulator are public and thus locally repeatable.

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 or updating the accumulator cannot cheat. So, we can either use the RSA accumulator with multi-party computation to secure the instantiation process, or we can use the Merkle Hash Tree accumulator.

- **RSA** The RSA accumulator, discussed in Section 5.2.1, is not strong because it is instantiated with a trapdoor: the factorization of n. However, this can be circumvented by using *multi-party computation* to instantiate the accumulator. If we use a t-threshold multi-party computation scheme, the accumulator will remain secure as long as fewer than t parties involved in the instantiation are malicious. For $t = \frac{1}{3}$, a polylogarithmic set of parties such that fewer than $\frac{1}{3}$ of them are malicious can be chosen by using the scalable leader election algorithm of [17]. This protocol involves n participating parties each talking to at most $polylog(n)$ other parties in $polylog(n)$ rounds, to elect a committee of size $O(\log(n))$. It guarantees that with overwhelming probability, the elected committee will have fewer than $\frac{1}{3}$ malicious parties if the set of participating parties had fewer that $\frac{1}{3}$ malicious parties. Once the committee is elected, it can perform multi-party computation (say, the protocol of [4] which is secure against $t \leq \frac{1}{3}$ malicious adversaries) to instantiate the accumulator.

  Note that using the dynamic accumulators from [9] requires knowledge of the trapdoor for performing deletions. Performing multi-party computation for each deletion is impractical. Instead, we propose implementing deletions by keeping a blacklist of revoked public keys. Periodically, the accumulator would be recomputed from scratch, taking the blacklist into account. However, we aim to keep the recomputations as infrequent as we can.

- **Merkle Hash Tree** The Merkle Hash Tree accumulator, discussed in Section 5.2.2, is strong and can thus be used with no modifications. Deletions can also be implemented efficiently via the original protocol. However, the Merkle Hash Tree accumulator is not compact; the accumulator will be of logarithmic size instead of constant size, making it slightly less easy to store.

We believe that the Merkle Hash Tree accumulator is the best bet for Certcoin, even though it is not compact.

6 Cutting Down on Storage: Distributed Hash Tables for Key Lookup

Cryptographic accumulators address the problem of verifying a public key without incurring a large storage overhead; now, we move to the problem of retrieving a public key with similar limitations. We first note that Certcoin's design creates a clear separation between authentication and distribution; domain purchases and public keys are stored in a blockchain, creating a publicly verifiable source for credentials. Unfortunately, blockchains do not inherently support key-value retrieval, which makes them a poor medium for facilitating dynamic queries. Traditional CAs typically also serve as self-authenticated keyservers, allowing a network to query it to retrieve other users' public keys. In order to be a practical PKI, it is clear Certcoin must also provide its own interface for efficiently retrieving public keys for a chosen identity. To accomplish this, our solution proposes the use of an authenticated DHT, effectively creating a resilient, decentralized keyserver maintained by the collection of purchased domains. Hence, Certcoin's distribution mechanism exploits some of the unique properties of a Kademlia DHT [19] to create a self-sustaining key-delivery service.

In standard form, Kademlia is unauthenticated and thus susceptible to poisoning, routing, and Sybil attacks. However, below we will describe methods of prevention for each, leveraging the authentication data stored in Certcoin's blockchain. Of course, we want any party to have access to Certcoin's public keys, whether they have purchased a domain or not we do not. Therefore, we choose not to authenticate key lookup requests. The read-only nature of these requests guarantees that an outside adversary cannot exploit this RPC to alter key-value pairs. Hence, the remainder of this section will exclude this operation from authentication concerns.

6.1 Design of the Kademlia Distributed Hash Table

Many of the desirable properties that Kademlia offers are the result of its symmetric routing protocol which introduces minimal communication overhead in order to update routing tables. The protocol is designed such that any packet sent over the network contains useful information about that status of viable nodes and routes. Thus, Kademlia will automatically route around failures just by continued communication from other nodes and querying from users.

In addition, Kademlia leverages the fact that node failure is inversely related to uptime by attempting to maintain the oldest active nodes in its routing table, thus guaranteeing that the routes to the most reliable nodes are propagated into other routing tables in the network as described above. It then becomes the best interest of a node in the DHT to remain online, that way its presence is maintained in an ever growing number of other routing tables. Since response time in the DHT is dependent on the number hops required to reach a destination, an unreliable node will witness slower query responses for clients wishing to access its public keys. This is enforced by Certcoin's modification to Kademlia's key retrieval protocol discussed further in 6.2.3.

Every node maintains k possible routing values for each possible path. Assuming no entry was repeated between each of those nodes, the probability of that a node cannot reach a key is roughly $(2^{-k})^k = 2^{-k^2}$; this would require the k nodes in the sender's table all to have k failures in their own routing tables without any incoming or outgoing messages of any type to any of the nodes. However, overlap is certainly bound to happen but this is offset by the fact that the network's routing tables inherently store the most reliable nodes. As mentioned previously, routing tables are updated with every message sent or received. Consequently, the network becomes more secure as more requests are made since the average window in which an adversary would have to create a hole shrinks proportionally.

Furthermore, Kademlia's routing protocol can be modified to employ caching mechanisms that distribute values based on their frequency of access. This would make more popular keys harder to target by adversaries since greater portions of the network will have replicated the value. The additional space required to achieve this is minimal and nodes would benefit from participating because they would see reduced traffic directly to the source as more nodes continue to propagate the cached value.

6.2 Kademlia for Certcoin

Certcoin's decentralized keyserver provides both the integrity and availability of public keys by making three key alterations to the standard Kademlia protocol. The first is the use of digital signatures to provide authentication and integrity to RPCs. The second is a unique Node ID assignment process which provides significant resistance Sybil attacks. Finally, we conclude with

a modified key retrieval process which creates an incentive for all nodes to support and enforce the DHT, leading to its self-sustaining nature. To participate in key distribution, each domain contributes a node to the DHT under its purchased domain name. When a node is admitted to the network it publishes its (*public key*, *witness*) pair using the hash of its corresponding domain name as the key. For simplicity, in the rest of the section we will refer to the process of retrieving both the key and witness for a given domain name simply as key retrieval.

6.2.1 Authenticated RPCs

As mentioned before, unauthenticated DHTs suffer from susceptibility to poisoning and routing attacks. Fortunately, we can easily protect against these by requiring nodes to sign Kademila's RPCs. We start by describing how to transform a standard message M originating from node u into a authenticated message M_{Auth} suitable for Certcoin's DHT as follows

$$M_{Auth} = (M|PK_u|Witness_{PK_u})|Sig_{SK_u}(M|PK_u|Witness_{PK_u}).$$

The inclusion of PK_u and $Witness_{PK_u}$ allows the receiver to immediately verify the sender's authenticity using its accumulator. It should be noted that a node should never use unauthenticated messages to update its routing, since this is exactly what leads to poisoning attacks. Thus, when a node receives an unauthenticated query from outside the DHT for a particular key, it initiates a recursive, authenticated key retrieval sequence but disregards the message for the purposes of updating its routing table. Of course, any node in the path to the destination should update their tables using the verified RPCs.

6.2.2 Node ID Assignment

Even more pressing is defending against the well-known Sybil attack, whereby an adversary can create a large number ID's, usually with the ability to target specific areas of the ID space. In applications that can afford the presence of a Certificate Authority, this is typically addressed by ensuring a one-to-one mapping of entities to ID's via a registration process. Since the replacement of a CA is the ultimate goal of Certcoin, it is clear that this is not a valid solution. Fortunately, Certcoin's integration with the blockchain already provides a registration service. The DHT then creates node IDs which correspond to H(*domain name*, *signing public key*, *block timestamp*) where H is one-way and collision-resistant.

The inclusion of a timestamp greatly reduces an adversary's ability to find valid collisions, for even collisions that hash to the same ID are only valid if there exists a corresponding block with the same timestamp and public key. Thus an adversary can explore the key space for collisions at will but is forced to purchase a domain name in order to find collisions that will be admitted by the DHT.

In addition, hashing the signing public key firmly associates the ID of the entities in Kademila with the identity registered on the blockchain. This provides a form of two-factor authentication, since nodes are only admitted to the network if they have both a valid digital signature and a confirmation via the hash.

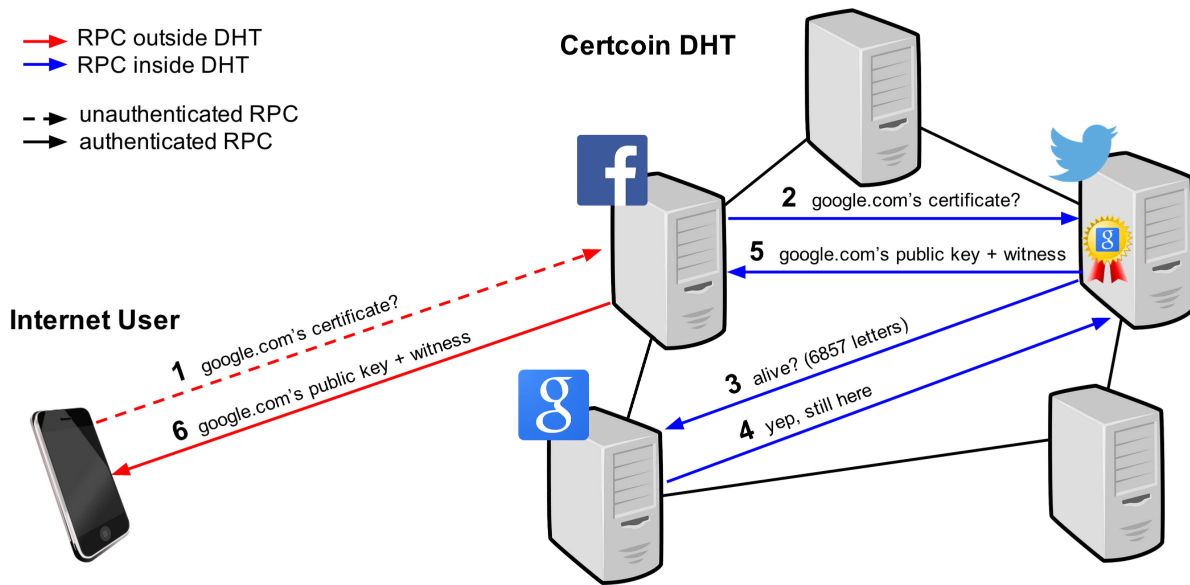

Figure 3: An example of a Certcoin's alternate key retrieval protocol, demonstrating the DHT's response to a query made by an arbitrary user.

6.2.3 Key Retrieval

In order to prevent nodes from simply removing themselves and leaving the rest of the network to distribute its keys, we require any node answering a key lookup to respond only if it is able to receive a response to a heartbeat message (with array bounds checking of course) from the hostname associated with the key. Nodes have an incentive to perform this check since failure to do so would allow an entity to leave, placing the burden of answering requests on the remaining nodes. Furthermore, servers have an obligation to respond to the heartbeat messages, otherwise users will not be able to retrieve the public key for its site.

We can further increase the security of the key-lookup procedure by requiring query responses to be delivered outside the DHT to be constructed precisely as other authenticated RPCs. This allows anyone to report a malicious node by publishing proof the incorrect responses. Once nodes become aware of this, they should immediately remove the offending node from its routing tables. In addition, any node responsible for storing values associated with the hostname should fail to honor requests for said key.

Users outside the network can also 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 and checking for consistency between the responses. In addition, performance critical applications should also opt to implement this tactic because they will experience lower expected latencies since it is more likely to be routed on an uncongested path.

7 Conclusion

In conclusion, we believe that Certcoin is a viable PKI, capable of replacing Certificate Authorities and PGP Webs of Trust. Our construction benefits from an entirely decentralized architecture offering inherent fault tolerance, redundancy, and transparency. Despite this, Certcoin supports the expected features of a full-fledged Certificate Authority including certificate creation, revocation, chaining, and recovery. Domain purchases and transfers are accomplished with simple Bitcoin transactions to incentivize miners. Certcoin employs cryptographic accumulators to maintain a constant size storage for authenticating domains, which is becoming ever more critical with recent trends in internet usage. Finally, our design addresses the need for a self-sustaining, trusted key distribution mechanism that provides efficient key retrieval, making Certcoin more practical for performance conscious applications. Moreover, Certcoin addresses many of the issues inherent to current PKIs, such as the need for a trusted third party and limited accessibility.

We plan to implement Certcoin in the future to explore further optimizations to found our results empirically and demonstrate its viability.

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] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, pages 1–10, New York, NY, USA, 1988. ACM.

[5] 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.

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

[7] 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.

[8] 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.

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

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

[11] Dawid Czagan. Symmetric and asymmetric encryption.

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

[13] Entrust. What is PKI?

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

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

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

[17] Valerie King, Jared Saia, Vishal Sanwalani, and Erik Vee. Scalable leader election. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm*, SODA '06, pages 990–999, New York, NY, USA, 2006. ACM.

[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.