Efficient Data Structures for Certificate Transparency

Abhishek Singh

Efficient Data Structures for Certificate Transparency

DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Technology
in
Computer Science

by

Abhishek Singh

[Roll No: CS-1423]

under the guidance of

Dr. Sushmita Ruj

Assistant Professor
Cryptology and Security Research Unit

Indian Statistical Institute
Kolkata-700108, India

July 2016

To my family and friends

CERTIFICATE

This is to certify that the dissertation entitled **"Efficient Data Structures for Certificate Transparency"** submitted by **Abhishek Singh** to Indian Statistical Institute, Kolkata, in partial fulfillment for the award of the degree of **Master of Technology in Computer Science** is a bonafide record of work carried out by him under my supervision and guidance. The dissertation has fulfilled all the requirements as per the regulations of this institute and, in my opinion, has reached the standard needed for submission.

Sushmita Ruj
Assistant Professor,
Cryptology and Security Research Unit,
Indian Statistical Institute,
Kolkata-700108, INDIA.

Acknowledgments

I would like to show my highest gratitude to my advisor, *Sushmita Ruj*, Cryptology and Security Research Unit, Indian Statistical Institute, Kolkata, for her guidance and continuous support and encouragement. She has literally taught me how to do good research, and motivated me with great insights and innovative ideas.

I would also like to thank *Binanda Sengupta*, Senior Research Fellow, Indian Statistical Institute, Kolkata, for his valuable suggestions and discussions.

My deepest thanks to all the teachers of Indian Statistical Institute, for their valuable suggestions and discussions which added an important dimension to my research work.

Finally, I am very much thankful to my parents and family for their everlasting supports.

Last but not the least, I would like to thank all of my friends for their help and support. I thank all those, whom I have missed out from the above list.

Abhishek Singh
Indian Statistical Institute
Kolkata - 700108 , India.

Abstract

Browsers can detect malicious websites that are provisioned with forged or fake TLS/SSL certificates. However, they are not so good at detecting malicious websites if they are provisioned with mistakenly issued certificates or certificates that have been issued by a certificate authority (CA) which is compromised. Google proposed certificate transparency which is an open framework to monitor and audit certificates in real time. However, the size of a proof is logarithmic in the number of certificates. This large proof size consumes a lot of bandwidth. Apart from this drawback, revocation is not handled. In NDSS 2014, Ryan extended certificate transparency to handle efficient revocation of a certificate. However, the size of a proof still remains logarithmic in the number of certificates.

We have developed and extended the concept of certificate transparency introduced by Google and its enhanced version proposed by Ryan. We have introduced bilinear-map accumulators (in the context of certificate transparency) in order to provide proofs of constant size irrespective of the number of certificates. Our scheme has many desirable properties like efficient revocation, constant size proofs, low verification cost and update costs comparable to the existing schemes. We provide proofs of security and evaluate the performance of our scheme.

Keywords: *Certificate transparency, Revocation, Bilinear-map accumulator*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

In public key encryption schemes, the sender encrypts a message using the receiver's public key to produce a ciphertext, and the receiver decrypts the ciphertext to obtain the message using her private key. On the other hand, in digital signature schemes, a verifier uses the public key of the signer in order to check whether a signature on a given message is valid. Thus, in public key cryptography, a user should be able to verify the authenticity of the public keys of other users. Suppose, for example, a user logs in into her bank account through a web browser, and this web session is made secure by using the public key of the concerned bank. If the web browser uses the public key of some attacker instead of the bank's public key, then all the (possibly sensitive) information along with the login credentials may be known to the attacker who can misuse them later.

One solution to prevent such attacks is to rely on a trusted entity called *certificate authority* (CA) that issues digital certificates showing the association of public keys with the users. The certificate authority signs each of these certificates using its private key. When a user wants to communicate with a server, she receives the server's certificate (signed by an appropriate CA). The process of verifying the "signed certificate" is done by the user's software (typically a web browser) that maintains an internal list of popular CAs and their public keys. It uses the appropriate public key to verify the signature on the server's certificate. However, this CA model suffers from the following two major problems [33]. Firstly, if the CA is untrusted, then it may issue certificates which certify the ownership of fake public keys that could be created by an attacker or by the CA itself. So, the CA must be trustworthy. On the other hand, if the private key of a certificate owner is compromised, then the CA must revoke the corresponding certificate before its expiration date.

Over the past few years there have been numerous instances of issuing fake certificates by compromised CAs. In March 2011, in an attack on a Comodo reseller, fake certificates were issued for mail.google.com, www.google.com, login.yahoo.com,

login.skype.com, login.live.com, and addons.mozilla.org [32]. Comodo suggested that the attack originated from an Iranian IP address. In July 2011, an attacker with access to system of DigiNotar, a Dutch CA, improperly issued a certificates for all domains of Google [11, 1]. It was claimed that as many as 250 false certificates for an unknown number of domains were released. For weeks the rogue certificate had been abused in a large scale Man-In-The-Middle (MITM) attack on approximately 300,000 users that were almost exclusively located in the Islamic Republic of Iran. Eventually, the Dutch CA's certificates were revoked and the CA was shut down. More recently, a large U.S.-based CA (TrustWave) admitted that it issued subordinate root certificates to one of its customers so the customer could monitor traffic on their internal network. Later, that customer used subordinate root certificates to create fake SSL certificates. Although Trustwave has revoked the certificate and stated that it will no longer issue subordinate root certificates to customers.

In order to overcome these problems, researchers have come up with various solutions. Techniques like certificate pinning [24, 27] and crowd-sourcing [3, 4, 17, 35] have been proposed to restrict the browser to obtain certificates from the verified CAs only. All the approaches discussed above are centralized where a certificate authority acts as a trusted third party responsible for managing digital certificates for a network of users. An alternative approach to the problem of public authentication of public key information is the Web of Trust [13] (WoT as coined by Phil Zimmerman) that uses self-signed certificates and third party attestations of those certificates. The WoT is entirely decentralized in that a user signs the public keys of other users (whom she trusts) and designate them as trustworthy. However, it is difficult for a new (or remote) user to join the network as she has to meet with someone in person to get her public key signed for the first time. Moreover, the WoT does not deal with key revocation. Certcoin [19] is another decentralized public key infrastructure (PKI) based on Bitcoin [30].

Certificate transparency (CT) [25, 21], a technique proposed by Google, aims to make certificate issuance transparent by efficiently detecting fake certificates issued by malicious certificate authorities. To achieve transparency, a public append-only log structure is maintained containing all the certificates. Domain owners can obtain proofs that their certificates are recorded in the log structure appropriately. Then, they provide the certificate along with a proof to their clients so that the clients can be convinced about the authenticity of the received certificate. Google's CT scheme provides two basic proofs: *proof of presence* (that is, the issued certificate is present in the log structure) and *proof of extension* (that is, the log structure is maintained in an append-only mode). However, certificate transparency by Google does not handle revocation of a certificate. Ryan [33] extends Google's scheme to support certificate revocation efficiently and provides two more proofs: *proof of currency* (that is, the issued certificate is current or active) and *proof of absence of a user* (that is, no certificates have been issued for a particular user). In both of the CT schemes described above, the size of a proof, the computation time (and verification time) of

the proof are logarithmic in the number of certificates present in the log structure.

1.2 Our Contribution

Our contributions are summarized as follows.

- We have proposed an efficient and secure structure for CT, which is an improvement over existing schemes. We have designed a certificate transparency scheme (using bilinear-map accumulators and binary trees) that supports all the proofs found in the previous works. For the existing proofs, the parameters in our scheme are comparable to those proposed in the earlier schemes.
- In addition to the proof of currency, we have introduced another proof (*proof of absence of a certificate*) related to certificate revocation. Both of these proofs are of constant size, and verification cost is also constant for them.
- To prove the security of our scheme, we have defined the security model for certificate transparency, and we have shown that our scheme is secure in this model. To the best of our knowledge, in this work, the security model for a certificate transparency scheme is defined formally for the first time.
- We have also provided the performance evaluation of our scheme. We defer the detailed performance analysis of our construction in Section 5.3. For a quick review, we summarize the comparison of our construction with the existing schemes for certificate transparency in Table 5.3 in Section 5.3.

1.3 Thesis Outline

The rest of the thesis is organized as follows. In Chapter 2 and 3, we briefly discuss about the preliminaries and background related to our work. Chapter 4, describes the detailed construction of our scheme along with the data structures used in the construction. In Chapter 5, we provide the security analysis and performance analysis our scheme. In the concluding Chapter 6, we summarize the work done and future directions related to our work.

Chapter 2

Preliminaries

2.1 Notation

We take λ to be the security parameter. An algorithm $\mathcal{A}(1^\lambda)$ is a probabilistic polynomial-time algorithm when its running time is polynomial in λ and its output y is a random variable which depends on the internal coin tosses of $\mathcal{A}$. An element a chosen uniformly at random from a set S is denoted as $a \xleftarrow{R} S$. A function $f : \mathbb{N} \to \mathbb{R}$ is called negligible in λ if for all positive integers c and for all sufficiently large λ, we have $f(\lambda) < \frac{1}{\lambda^c}$.

2.2 Merkle Hash Tree

A Merkle hash tree [29] is a binary tree where each leaf-node stores a data item. The label of each leaf-node is the data item stored in the node itself. A collision-resistant hash function h is used to label the intermediate nodes of the tree. The label of a intermediate node v is the output of h computed on the labels of the children nodes of v. A Merkle hash tree is used as a standard tool for efficient memory-checking. Figure 2.1 shows a Merkle hash tree containing the data items $\{d_1, d_2, \ldots, d_8\}$ stored at the leaf-nodes. Consequently, the labels of the intermediate nodes are computed using the hash function h.

The hash value of the root node A (the root digest) is made public. The proof showing that a data item d is present in the tree consists of the data item d and the labels of the nodes along the *associated path* (the sequence of siblings of the node containing the data item d). For example, a proof showing that d_3 is present in the tree consists of $\{d_3, (d_4, l_D, l_C)\}$, where d_4, l_D and l_C are the labels of the nodes K, D and C, respectively. Given such a proof, a verifier computes the hash value of the root. The verifier outputs `accept` if the computed hash value matches with the public root digest; it outputs `reject`, otherwise. The size of a proof is logarithmic in the number of data items stored in the leaf-nodes of the tree.

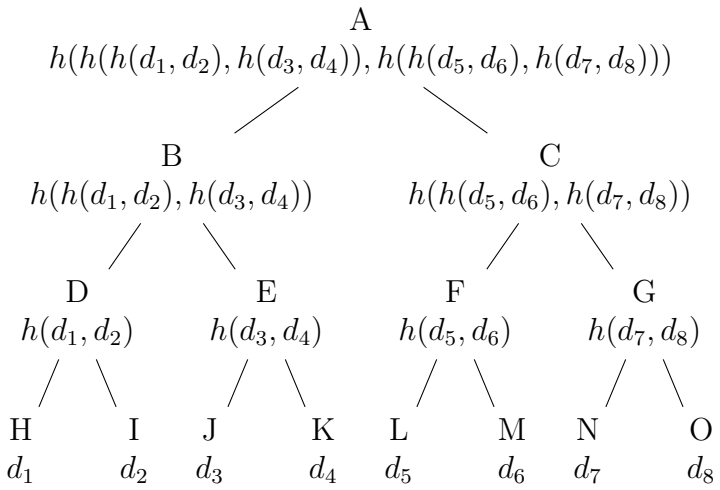


Figure 2.1: A Merkle hash tree containing data items $\{d_1, d_2, \ldots, d_8\}$.

Due to the collision-resistance property of h, it is impossible (except with some probability negligible in the security parameter λ) to add a new data item in the Merkle hash tree without changing the root digest of the tree.

2.3 Bilinear Maps

Let G_1, G_2 and G_T be multiplicative cyclic groups of prime order p. Let g_1 and g_2 be a generator of G_1 and G_2 respectively. A bilinear map is a function $e : G_1 \times G_2 \rightarrow G_T$. The bilinear map e has the following properties:

- Bilinearity: for all $u \in G_1$, $v \in G_2$, $a \in \mathbb{Z}_p$, $b \in \mathbb{Z}_p$, we have $e(u^a, v^b) = e(u, v)^{ab}$
- Non-degeneracy: e is non-degenerate, that is, $e(g_1, g_2) \neq 1$.

Furthermore, properties (1) and (2) imply that $e(u_1 \cdot u_2, v) = e(u_1, v) \cdot e(u_2, v)$ for all $u_1, u_2 \in G_1$, $v \in G_2$.

If $G_1 = G_2 = G$, the bilinear map is symmetric; otherwise, asymmetric. Unless otherwise mentioned, we consider only the bilinear maps which are efficiently computable and symmetric. Let g be a generator of G.

2.4 Assumptions

Definition 1 (Strong RSA assumption) *Given an RSA modulus N and a random element $x \in \mathbb{Z}_N$, it is hard (i.e., it can be done with probability that is* `negl`*(k), which is negligible in the security parameter k) for a computationally bounded adversary $\mathcal{A}$ to find $y > 1$ and a such that $a^y = x \bmod N$.*

Definition 2 (q-strong Diffie Hellman assumption) *Let $G =< g >$ be a cyclic group of prime order p and $\kappa \in \mathbb{Z}_p^*$. Under the q-strong Diffie-Hellman assumption, any probabilistic polynomial-time algorithm A that is given set $\{g^{\kappa^i} : 0 \leq i \leq q\}$, finds a pair $(x, g^{\frac{1}{x+\kappa}}) \in \mathbb{Z}_p^* \times G$ with at most $O(1/p)$ probability, where the probability is over the random choice of $\kappa \in \mathbb{Z}_p^*$ and the random bits chosen by A.*

2.5 Accumulators

A cryptographic accumulator is a one-way membership function. It answers a query to check whether an element is a member of a set X without revealing the individual members of X. An accumulator scheme was introduced by Benaloh and de Mare [9] and further developed by Baric and Pfitzmann [5]. Both of these constructions are based on RSA exponentiation functions (secure under the *strong RSA* assumption) and provides a constant size membership witness for any element in X with respect to the accumulation value denoted by $A(X)$ [16]. Universal accumulators [26] are designed to provide both membership and non-membership witnesses of constant size. Camenisch and Lysyanskaya [12] proposed a dynamic accumulator in which elements can be efficiently added into or removed from the accumulator. Whenever an element is inserted or deleted from X, then this results in an update on $A(X)$ and the membership (and non-membership) witnesses.

2.5.1 RSA Accumulators

Prime Representatives: Prime representatives provides the solution whenever it is necessary to map general elements to prime numbers. In particular, one can map a k-bit element e_i to a $3k$-bit prime element x_i using *two-universal hash functions.*

Definition 3 *H is a two-universal family (set) of hash functions from M to N if, for all x, y in M such that $x \neq y$,*

$$Pr_{h \in H}[h(x) = h(y)] \leq= \frac{1}{|N|}$$

similar to pairwise independence.

In our case, set M is the set of $3k$-bit boolean vectors, N is the set of k-bit boolean vectors, and the two-universal hash function used,

$$h(x) = Fx$$

where F is $k \times 3k$ boolean matrix. This system has more than one solution, i.e., one k-bit element is mapped to more than one $3k$-bit elements. We are interested in finding only one solution which is prime.

The RSA accumulator: It provides an efficient technique to produce a short (computational) proof that a certain element is a member of a set of elements. Let $S = \{e_1, e_2, \ldots, e_n\}$ be the set of elements. Each element e is represented by k-bit element. Let N be a k'-bit RSA modulus ($k' > 3k$), namely $N = pq$, where p and q are strong primes that are suitably large, e.g., $p, q > 2^{\frac{3}{2}k}$. It can represent S compactly and securely with an *accumulation* value which is an k'-bit integer,

$$A(S) = g^{r(e_1)r(e_2)\ldots r(e_n)} \bmod N$$

where g is relative prime to N and $r(e_i)$ is a $3k$-bit prime representative. RSA modulus N, the exponentiation base g and the two-universal hash function h comprise the public key pk. Each element e_i in set S has membership witness as,

$$A_{e_i} = g^{\prod_{e_j \in S: e_j \neq e_i} r(e_j)} \bmod N$$

Then verifier can verify the membership of element e_i in S by computing $A_{e_i}^{r(e_i)}$ and checking that it is equal to publically known accumulation value $A(S)$.

It has a property that any computationally bounded adversary $\mathcal{A}$ cannot find another set $S' \neq S$ such that $A(S') = A(S)$, unless $\mathcal{A}$ breaks the *strong RSA assumption* (see Definition 1 in Section 2.4) [6].

2.5.2 Bilinear-Map Accumulator

Nguyen [31] constructed the first dynamic (but not universal) accumulator based on bilinear maps. Later, Damgård and Triandopoulos [15] extended the work of Nguyen in order to provide both membership and non-membership witnesses. This scheme is proved secure under the *q-strong Diffie-Hellman* assumption. We briefly describe this scheme proposed by Damgård and Triandopoulos [15] as follows.

Let an algorithm BLSetup(1^λ) output (p, g, G, G_T, e) as the parameters of a bilinear map, where g is a generator of G. Given a set $X = \{x_1, x_2, \ldots, x_n\}$, an accumulation function $f_s(X) : 2^{\mathbb{Z}_p^*} \to G$ gives the accumulation value $A(X)$ defined as $f_s(X) = A(X) = g^{(x_1+s)(x_2+s)\ldots(x_n+s)}$, where $s \xleftarrow{R} \mathbb{Z}_p^*$ is the secret trapdoor information. The set $\{g^{s^i} | 0 \leqslant i \leqslant q\}$ is public, where q is an upper bound on $|X|$.

For any $x \in X$, the membership witness is defined as $w_x = g^{\prod_{x_j \in X: x_j \neq x}(x_j+s)}$. The verifier can verify the membership witness by checking the equation

$$e(w_x, g^x \cdot g^s) \stackrel{?}{=} e(A(X), g).$$

For any $y \notin X$, the non-membership witness is defined as $\hat{w}_y = (w_y, v_y)$, where $v_y = -\prod_{x \in X}(x - y) \bmod p$ and $w_y = g^{\frac{(\prod_{x \in X}(x+s)) + v_y}{y+s}}$. Now, the verifier can verify the non-membership witness by checking the equation

$$e(w_y, g^y \cdot g^s) \stackrel{?}{=} e(A(X) \cdot g^{v_y}, g).$$

Under the q-Strong Diffie-Hellman assumption (see Definition 2 in Section 2.4), any probabilistic polynomial-time algorithm $\mathcal{B}(1^\lambda)$, given any set X ($|X| \leqslant q$) and set $\{g^{s^i} | 0 \leqslant i \leqslant q\}$, finds a fake non-membership witness of a member of X or a fake membership witness of a non-member of X with respect to $A(X)$ only with a probability negligible in λ (measured over the random choice of $s \in \mathbb{Z}_p^*$ and the internal coin tosses of $\mathcal{B}$) [15, 31].

Chapter 3

Related Work

3.1 Certificate Transparency

Certificate transparency (CT) [25, 21] is a technique proposed by Google in order to efficiently detect fake public keys issued by malicious certificate authorities. A detailed description of this open framework is given in [21]. We provide a brief overview of the same. The framework consists of the following main components (see Figure 3.1) .

- **Certificate Log:** All the certificates that have been issued by certificate authorities are stored in an append-only log structure. The log structure is maintained as a Merkle hash tree (see Section 2.2). This enables the log maintainer to provide two types of verifiable cryptographic proofs: (a) proof of presence (that is, the issued certificate is present in the log structure) and (b) proof of extension (that is, the log structure is maintained in an append-only mode).

- **Monitors:** Monitors are publicly run servers that contact all of the log maintainers periodically and watch for suspicious certificates (illegitimate or unauthorized certificates, unusual certificate extensions, or certificates with strange permissions). Monitors also verify that all logged certificates are visible in the log structure.

- **Auditors:** Auditors are lightweight software components that perform the following two functions. Firstly, they can verify that logs are behaving correctly and are cryptographically consistent. Secondly, they can verify that a particular certificate is recorded in a log appropriately. An auditor may be an integral component of a browser's TLS client, a standalone service, or a secondary function of a monitor.

We may consider a certificate as a signed pair (u, pk_u), where u is a user and pk_u is a public key of u. We denote the certificate by $cert(u, pk_u)$. The log maintainer maintains certificates issued by CAs in a Merkle hash tree that stores the certificates

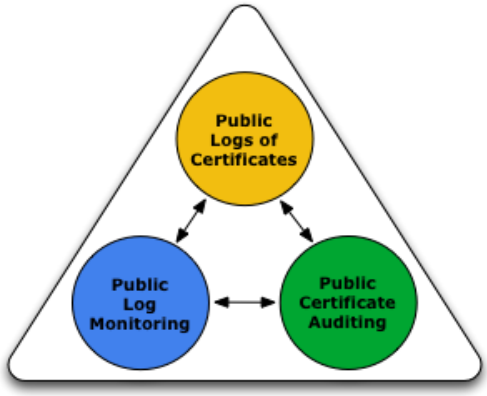


Figure 3.1: Three components of Certificate Transparency

as the leaf-nodes in chronological order (left-to-right). Whenever a new certificates is added, it is appended to the right of the tree. User's browser accepts the certificate only if it is accompanied by a proof of the presence of the certificate in the log. For proof of extension, the monitor submits two hash values (computed at different time) of the log to the CA. The CA returns a proof that one of them is an extension of the other. Both the proof of presence and the proof of extension can be done in time/space complexity of $O(\log n)$, where n is the total number of certificates.

3.2 Enhanced Certificate Transparency

Enhanced certificate transparency (ECT) by Ryan [33] proposes an idea to address the revocation problem that was left open by Google. It provides transparent certificate revocation. It also reduces reliance on trusted parties by designing the monitoring role so that it can be distributed among the browsers.

In this extension, Ryan introduced two proofs: (a) proof of currency (that is, the certificate is issued and not revoked) and (b) proof of absence of a user (that is, the CA has not issued any certificate for a particular user). Both of these proofs are logarithmic in the number of certificates. The proof of extension remains the same as that of Google's certificate transparency.

3.3 Accountable Key Infrastructure

Accountable Key Infrastructure (AKI) [22], integrating an architecture for key revocation of all entities (e.g., CAs, domains) with an architecture for accountability of all infrastructure parties through checks-and-balances. Through checks-and-balances among independent entities, AKI reduces the amount of trust placed in any one in-

frastructure component (e.g., CA) and successfully distributes trust over multiple parties. It detects misbehaving entities by maintaining a public log of certificates. A domain in AKI can define which and how many CAs are required to update the certificate. To enable recovery, certificates can be updated through another set of CAs.

In their approach of check-and-balances, they assume a set of entities that do not collude: CAs, public log servers, and validators. It heavily relies on third party (called validators) to ensure that log is properly maintained.

3.4 Attack Resilient Public-Key Infrastructure

Attack Resilient Public-Key Infrastructure (ARPKI) [8], a public-key infrastructure that ensures that certificate-related operations, such as certificate issuance, update, revocation, and validation, are transparent and accountable. ARPKI is inspired by AKI's design and employs some of its concepts. A client in ARPKI can designate n different service providers (e.g. CAs and log maintainers), and only needs to contact one CA to register her certificate. Each of the designated service providers will monitor the behaviour of other designated service providers. As a result, it offers resilience against impersonation attacks that involve $n - 1$ compromised entities.

The involvement of all n designated service providers in certificate registration, confirmation and update cause considerable extra latencies and delay in client connections.

3.5 Distributed Transparent Key Infrastructure

Distributed Transparent Key Infrastructure (DTKI) [36] is an infrastructure for managing keys and certificates on the web in a way which is transparent, minimises oligopoly. It prevents attacks that use fake certificates and eliminates the need for trusted parties. There are mainly two types of logs. First, certificate log stores all valid and invalid certificates for particular set of domains. Second, mapping log stores the association between certificate logs and the domains they are responsible for.

Rather than relying on trusted parties (e.g. monitors in CT and validators in AKI) to verify the healthiness of logs and the relations between logs, DTKI uses a crowdsourcing to ensure the integrity of the log and the relations between mapping log and a certificate log, and between certificate logs.

3.6 Key Usage Detection (KUD)

Key Usage Detection protocols are used to monitor the usage of cryptographic keys (for both signing and decryption) and can detect unauthorised key usage, i.e., that

allows the device owner to detect if another party is using the device's long-term key. This concept was proposed by Ryan and Yu [37]. They achieve this by storing keys in append-only log, which the device owner can interrogate. In their scheme, log is maintained as a tree of trees.

The log is an append-only Merkle Tree T which records the entire update history. Leaves of tree T contains items which are ordered chronologically. Each leaf node is labeled by root hash value of another Merkle Tree T'. Items in T' are also stored only in leaves but ordered lexicographically by user's identity. Each leaf contains user's identity and list of certificates for different devices of the same user. It can provide proofs if/when a specific key has been compromised as a result of an attack and allow for the key to be revoked. These proofs can be proof of presence, proof of absence, proof of extension and proof of currency. All of these proofs are logarithmic in the number of users in Merkle Tree.

3.7 Key Transparency to End Users

CONIKS [28] is a transparency log scheme that aims to enable privacy-preserving transparency logging for end-user keys, for applications such as secure messaging. CONIKS uses a Merkle prefix tree to aggregate user's public keys. Position of user's key in the tree is determined by user's identity. The root of the tree is signed by the log maintainer and made public and is known as signed tree root (STR). Signed tree roots are linked to its previous by including a hash of the previous signed tree root in its computation.

CONIKS employs cryptographically primitives to make the log privacy-preserving. Private index is computed for each username by using verifiable unpredictable function which is a function that requires a private key. CONIKS allows clients to monitor their own keys and quickly detect equivocation with high probability. Log maintainer provides the membership proof of a key by providing authentication path from leaf node containing key to root node and these proofs are logarithmic in the number of users in the log.

3.8 Decentralized PKI

3.8.1 Web of Trust

Web of Trust (WoT) [13] is an alternative approach to the problem of public authentication of public key information. This term was coined by Phil Zimmerman. WoT uses self-signed certificates and third party attestations of those certificates. The WoT is entirely decentralized in that a user signs the public keys of other users (whom she trusts) and designate them as trustworthy. However, it is difficult for a new (or remote) user to join the network as she has to meet with someone in person

to get her public key signed for the first time. Moreover, the WoT does not deal with key revocation.

3.8.2 Certcoin

Certcoin [19] is another decentralized public key infrastructure (PKI) based on Bitcoin [30]. It incorporates the aspects of transparent Certificate Authorities and of the Web of Trust. Certcoin is build 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. It supports the features of a Certificate Authority including certificate creation, revocation, chaining, and recovery. Using Certcoin, user can register a domain and update public keys corresponding to a respective domain.

Chapter 4

Proposed Scheme

4.1 Our Construction

In our extension of certificate transparency, each certificate issued by a (possibly malicious) certificate authority (CA) is associated with proofs showing the validity of that particular certificate. The certificates issued by various CAs are stored in a public (and append-only) log structure that is maintained by the log maintainer. This log maintainer that maintains the public log of all certificates issued by certificate authorities is known as the *certificate prover* (CP). We use the terms "log maintainer" and "certificate prover" interchangeably in this work. Anyone can submit certificates to a log, although certificate authorities will likely be the foremost submitters. Likewise, anyone can query a log for a cryptographic proof, which can be used to verify that the log is behaving properly or verify that a particular certificate has been logged.

The number of log servers doesn't have to be large (say, much less than a thousand worldwide), and each could be operated independently by a CA, an ISP, or any other interested party. In general, the certificate prover is a server which has enough resources (in terms of storage capacity and computing power) to store all the certificates and compute the proofs relevant to certificate transparency efficiently. We define certificate transparency as follows.

Definition 4 *A certificate transparency scheme consists of the following algorithms.*

- Setup(1^λ)*: Given the security parameter λ, the certificate prover (CP) generates the private key (sk), public parameters (PP), and instantiates the log structure.*
- Insert(c, sk, PP)*: The certificate prover (CP) adds a new certificate c in the log structure where $c = cert(u, pk_u)$ and updates the public parameters.*
- Revoke(c, sk, PP)*: The certificate prover (CP) removes a certificate c from the log structure where $c = cert(u, pk_u)$ and updates the public parameters.*
- Query(PP)*: This algorithm is run by the auditor, and it returns a query Q.*

- ProofGeneration(Q, PP)*: On input a query* Q *and the public parameters* PP*, the certificate prover (CP) returns a proof* $\Pi(Q)$.
- Verify(Q, Π, PP)*: On input a query* Q*, a proof* Π *and the public parameters* PP*, the auditor outputs either* `accept` *or* `reject`.

4.1.1 Data Structures Used in Our Construction

In our construction, the public log structure maintained by the CP is organized by using the following tree data structures: *chronTree*, *searchTree* and *accTree*. Let n be the total number of certificates present in the log structure, t be the total number of users and m be the total number of active certificates present in the log structure. Clearly, $n \geqslant t \geqslant m$. Notations used in our construction are described in Table 4.1. We describe the data structures used in our construction as follows.

Symbol	Definition
$digCT$	Root hash value of the chronTree
$digST$	Root hash value of the searchTree
$A(X)$	Accumulation value of set X in accumulator
n	Total number of certificates in the log structure
t	Total number of users
m	Total number of active certificates
$cert(u, pk_u)$	Certificate of user u having public key pk_u

Table 4.1: Notations used in our construction.

- **chronTree**. The chronTree is a Merkle hash tree (see Section 2.2) where certificates are stored as the leaf-nodes of the tree. The certificates are arranged in the chronological order in left-to-right manner. When a new certificate $c = cert(u, pk_u)$ is issued by a CA, it is added to the right of the chronTree. When a certificate $c = cert(u, pk_u)$ is revoked, another certificate $c' = cert(u, \texttt{null})$ is added to the right of the chronTree. A collision-resistant function h is used to compute the hash values corresponding to the nodes of the chronTree. The hash value of the root node (the root digest) of the chronTree is denoted by $digCT$. The structure of the chronTree is shown in Figure 4.1. The leaf-nodes of the chronTree contain items of the form $x = (c, A, digST)$, where A is the accumulation value of the updated set X after inserting the certificate c or revoking the active certificate of the corresponding user.
- **searchTree**. This tree is organized as a modified binary search tree where data items corresponding to the users are stored in the lexicographic order (of the users). Here, a data item corresponding to a user u is of the form $(u, List(pk_u))$, where $List(pk_u)$ is the list of N most recent public keys of the user u. In other

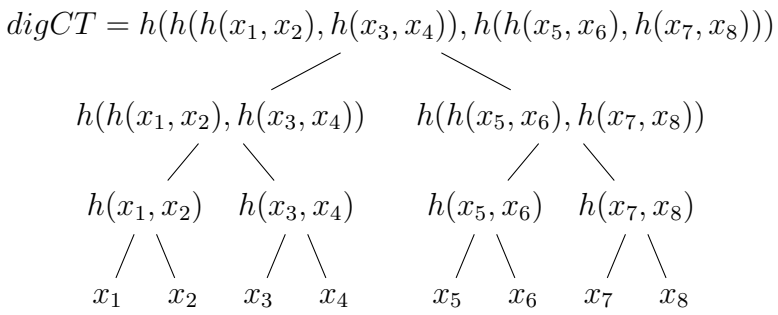


Figure 4.1: An example of the chronTree.

words, the last certificate in the list is the current public key of the user, and other keys are already revoked. The value of N is taken to be constant, and the list is maintained in a first-in-first-out (FIFO) fashion. The data items are stored in leaf-nodes as well as in non-leaf nodes such that an in-order traversal of the searchTree provides the lexicographic ordering of the users. The collision-resistant function h is used to compute the hash values corresponding to the nodes of the searchTree. The hash value of a node is computed on the data item (of that node) and the hash values of its children. This hash value is also stored in the node along with the data item. The hash value of the root node (the root digest) of the searchTree is denoted by $digST$. The value of $digST$ is linked to a leaf-node of the chronTree. The structure of the searchTree is illustrated in Figure 4.2. An example is showing searchTree containing data items d_i of the form $(u_i, List(pk_{u_i}))$, where $List(pk_u)$ is the list of N most recent public keys of the user u and $u_1, u_2, \ldots, u_9$ are in lexicographic order.

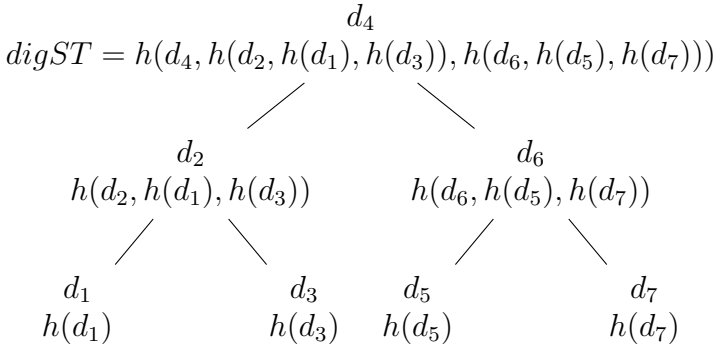


Figure 4.2: An example of the searchTree.

- **accTree**. This tree is organized as a binary search tree in which active certificates are stored in the lexicographic order of the users. Let X be the set of active (or current) certificates for different users. In our construction, the set

X is implemented as accTree. Each node in the accTree contains a certificate $c = cert(u, pk_u) \in X$ and the corresponding membership (in X) witness w_c of c. The accumulation value $A(X)$ is linked to a leaf-node of the chronTree. Figure 4.3 shows the structure of the accTree.

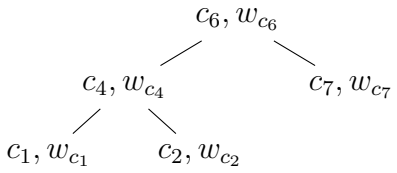


Figure 4.3: An example of the accTree.

4.1.2 Detailed Construction

In this section, we describe our construction in details. Our construction involves the following algorithms to achieve certificate transparency.

- Setup(1^λ): The Setup algorithm runs BLSetup(1^λ) to output (p, g, G, G_T, e) as the parameters of a bilinear map, where g is a generator of G. Let X be the set of active certificates issued by a certificate authority (CA), that is,

$$X = \{cert(u_i, pk_{u_i})\},$$

where pk_{u_i} is the active public key issued by the CA for the user u_i. The Setup algorithm selects a random element $s \xleftarrow{R} \mathbb{Z}_p^*$ as the secret trapdoor information. The set $\{g^{s^i} | 0 \leqslant i \leqslant q\}$ is made public, where q is an upper bound on $|X|$. The accumulation function $f_s(X) : 2^{\mathbb{Z}_p^*} \to G$ gives the accumulation value $A(X)$ defined as

$$f_s(X) = A = g^{\prod_{x_i \in X}(x_i + s)}.$$

The algorithm constructs a chronTree by inserting certificates in the chronological (left-to-right) order and returns $digCT$ as the root digest of the chronTree. It also constructs a searchTree by inserting users in the lexicographic order along with other relevant data associated with each user and returns $digST$ as the root digest of the searchTree. The algorithm constructs an accTree by inserting (only) the active certificates represented as set X along with their membership (in X) witnesses for different users. For each active certificate $c \in X$, the membership witness w_c is computed as

$$w_c = g^{\prod_{x_j \in X : x_j \neq c}(x_j + s)} = A^{\frac{1}{(c+s)}}.$$

These data structures are discussed in Section 4.1.1.

We note that a *collision-resistant* hash function h is used to compute the hash values in the searchTree and the chronTree.

Finally, $(p, g, G, G_T, e, \{g^{s^i} | 0 \leqslant i \leqslant q\}, h, A, digCT, digST)$ are set as the public parameters PP, and the secret key is the trapdoor value s.

- Insert(c, sk, PP): When a new certificate $c = cert(u, pk_u)$ is issued by a CA, then it asks the log maintainer (or the certificate prover) to insert the certificate in the log structure. The new certificate c is added to the log structure (accumulator, searchTree and chronTree) as follows. The public parameters PP are updated accordingly.

  - Adding c to accTree: Compute the new accumulation value A' (corresponding to the new set $X' = X \cup \{c\}$) as
  $$A' = A^{(c+s)}.$$
  The membership witness for c is A. For each $i \in X$, the updated membership witness is computed as
  $$w_i' = w_i^{(c+s)}.$$
  The accTree is updated accordingly.
  - Adding c to searchTree: Search for the node corresponding to the user u, if it is present, then append the new public key pk_u to the associated list of public keys for u. Otherwise, create a new node for u with the list containing only the public key pk_u and insert it in the searchTree maintaining the lexicographic order. Consequently, the root digest of the searchTree is updated as $digST'$.
  - Adding c to chronTree: Add a new node containing $(c, A', digST')$ to the right of the existing chronTree. The new root digest of the chronTree is updated as $digCT'$.

- Revoke(c, sk, PP): Let $c = cert(u, pk_u)$ be the certificate to be revoked. Then, the following operations are performed on the log structure, and the public parameters PP are updated accordingly.

  - Removing c from accTree: Compute the new accumulation value A' (corresponding to the new set $X' = X \backslash \{c\}$) as
  $$A' = A^{\frac{1}{(c+s)}}.$$
  Remove the node corresponding to the user u of certificate c from the accTree. For each $i \in X'$, the updated membership witness is computed as
  $$w_i' = w_i^{\frac{1}{(c+s)}}.$$
  The accTree is updated accordingly.

  - There are no changes in the searchTree for the revocation of c. Therefore, the value of $digST$, the root digest of the searchTree, remains the same.
  - Adding a new node for the user u to chronTree: Add a new node containing $(c', A', digST)$ to the right of the existing chronTree where $c' = cert(u, \texttt{null})$. The new root digest of the chronTree is updated as $digCT'$.

- Query(PP): This algorithm is run by an auditor to output a query Q. The type of the query Q is based on the type of the proof associated with Q. The types of proofs are as follows.

  - Proof of presence of a certificate (Type 1): The query Q asks for a proof of whether a certificate $c = cert(u, pk_u)$ is present in the log structure.
  - Proof of absence of a certificate (Type 2): The query Q asks for a proof of whether a certificate $c = cert(u, pk_u)$ is absent in the set of active certificates, that is, $c \notin X$.
  - Proof of absence of a user (Type 3): The query Q asks for a proof of whether a user u is absent, that is, there are no certificates for u in the log structure.
  - Proof of extension (Type 4): The query Q asks for a proof of whether the chronTree corresponding to $digCT'$ is an extension of the chronTree corresponding to $digCT$.
  - Proof of currency (Type 5): The query Q asks for a proof of whether pk_u is the current public key of the user u, that is, whether the certificate $c = cert(u, pk_u)$ is present in the set of active certificates ($c \in X$).

- ProofGeneration(Q, PP): Upon receiving the query Q, the certificate prover (CP) generates the corresponding proof $\Pi(Q)$ as follows.

  - Proof of presence of a certificate (Type 1): Search for the certificate c in the searchTree. If a node for the user u is present in the searchTree, define h_1 and h_2 to be the hash values of the children of the node (they are taken to be `null` if the node is a leaf-node). Send the following sequences of data items and hash values as the proof Π.

    Let the sequence of data items of the nodes along the search path be

    $$dataseq_{type1} = (d_1, d_2, d_3, \ldots, d_r)$$

    for some $r \in \mathbb{N}$, where d_1 is the data item corresponding to the node for the user u, d_r is the data item corresponding to the root node, and other data items correspond to the other intermediate nodes in the search path.

Let the sequence of hash values of the nodes in the *associated path* (the path containing the siblings of the nodes along the search path mentioned above) along with h_1 and h_2 be

$$hashseq_{type1} = (h_1, h_2, h(v_1), h(v_2), h(v_3), \ldots).$$

– Proof of absence of a certificate (Type 2): Search for the certificate c in the accTree. If there is no node for the user u in the accTree, then send the non-membership (not in X) witness $\hat{w}_c = (w_c, v_c)$ of certificate c, where

$$v_c = -\prod_{x \in X}(x - c) \bmod p \in \mathbb{Z}_p^* \quad \text{and} \quad w_c = g^{\frac{(\prod_{x \in X}(x+s))+v_c}{c+s}} \in G.$$

– Proof of absence of a user u (Type 3): The proof is similar to the proof of Type 1. Find the nodes in the searchTree corresponding to the users u_1 and u_2 such that they were the neighbor (in the lexicographic ordering) nodes of the node corresponding to u if u were present in the searchTree, that is, $u_1 \leqslant u \leqslant u_2$ lexicographically. These nodes can be found by searching for the user u in the searchTree, and the search ends at some leaf-node in the searchTree. The nodes corresponding to u_1 and u_2 reside on this search path itself, and one of them is the leaf-node (where the search ends).

Let the sequence of data items of the nodes along the search path be

$$dataseq_{type3} = (d_1, d_2, \ldots, d_{r'})$$

for some $r' \in \mathbb{N}$, where d_1 is the data item corresponding to the leaf-node, $d_{r'}$ is the data item corresponding to the root node, and other data items correspond to the other intermediate nodes in the search path.

Let the sequence of hash values of the nodes in the *associated path* (the path containing the sibling nodes of the nodes along the search path) be

$$hashseq_{type3} = (h(v_1), h(v_2), \ldots).$$

Finally, send $(dataseq_{type3}, hashseq_{type3})$ as the proof Π.

– Proof of extension (Type 4): Compare the chronTree structures corresponding to both $digCT$ and $digCT'$ and send one hash value per level of the latest chronTree as a proof Π.

If the chronTree corresponding to $digCT'$ is an extension of the chronTree corresponding to $digCT$, then the latter chronTree is a subtree of the earlier chronTree. The proof $\Pi = (h_1, h_2, \ldots)$ is the sequence of hash values of the nodes required to compute the current root digest $digCT'$ from the previous root digest $digCT$. Here, h_1 is the hash value of the sibling node of node v whose hash value is $digCT$ (that is, v is the root of the previous chronTree), h_2 is the hash value of the sibling node of parent node of v, and so on.

  - Proof of currency (Type 5): Search for the certificate c in the accTree. If c is present in the node for the user u in the accTree, then send the membership (in X) witness w_c stored at that node.

- Verify(Q, Π, PP): Given the query Q and the corresponding proof Π, the auditor verifies the proof in the following way depending on the type of the proof.

  - Proof of presence of a certificate (Type 1): Given the sequence of data items $dataseq_{type1} = (d_1, d_2, d_3, \ldots, d_r)$ and the sequence of hash values $hashseq_{type1} = (h_1, h_2, h(v_1), h(v_2), h(v_3), \ldots)$, the auditor verifies whether

$$h(\cdots h(d_3, h(d_2, h(d_1, h_1, h_2), h(v_1)), h(v_2)) \ldots) \stackrel{?}{=} digST$$

    and outputs `accept` if the equation holds; it outputs `reject`, otherwise.

  - Proof of absence of a certificate (Type 2): Given the non-membership (in X) witness $\hat{w}_c = (w_c, v_c)$ for a certificate c, the value of g^s (included in PP) and the accumulation value $A = f_s(X)$, the auditor verifies whether

$$e(w_c, g^c \cdot g^s) \stackrel{?}{=} e(A \cdot g^{v_c}, g)$$

    and outputs `accept` if the equation holds; it outputs `reject`, otherwise.

  - Proof of absence of a user (Type 3): Given the two sequences $dataseq_{type3} = (d_1, d_2, \ldots, d_{r'})$ and $hashseq_{type3} = (h(v_1), h(v_2), \ldots)$, the auditor verifies whether

$$h(\cdots h(d_3, h(d_2, h(d_1), h(v_1)), h(v_2)) \ldots) \stackrel{?}{=} digST$$

    and outputs `accept` if the equation holds; it outputs `reject`, otherwise.

  - Proof of extension (Type 4): Given the proof $\Pi = (h_1, h_2, \ldots)$, the auditor verifies whether

$$h(\ldots(h(h(digCT, h_1), h_2) \ldots) \stackrel{?}{=} digCT'$$

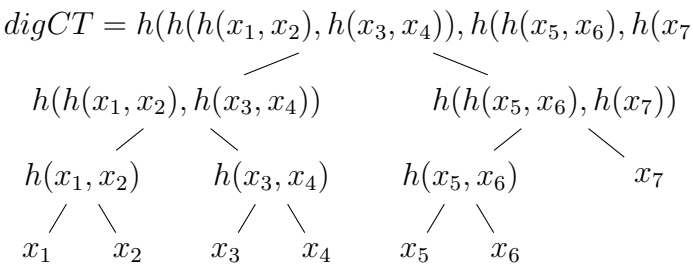
    and outputs `accept` if the equation holds; it outputs `reject`, otherwise.

  - Proof of currency (Type 5): Given the membership (in X) witness w_c for a certificate c, the value of g^s (included in PP) and the accumulation value $A = f_s(X)$, the auditor verifies whether

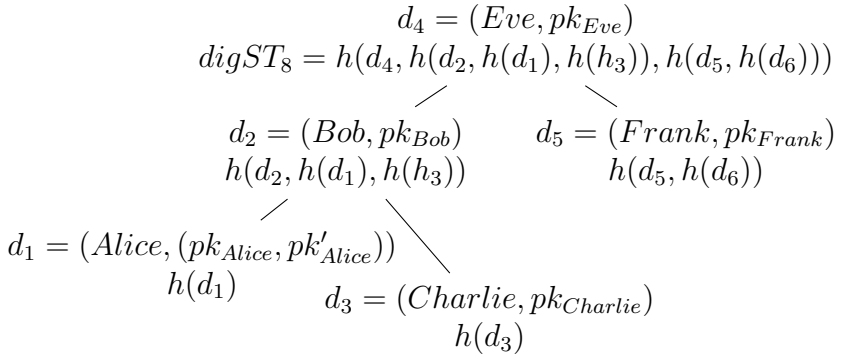
$$e(w_c, g^c \cdot g^s) \stackrel{?}{=} e(A, g)$$

    and outputs `accept` if the equation holds; it outputs `reject`, otherwise.

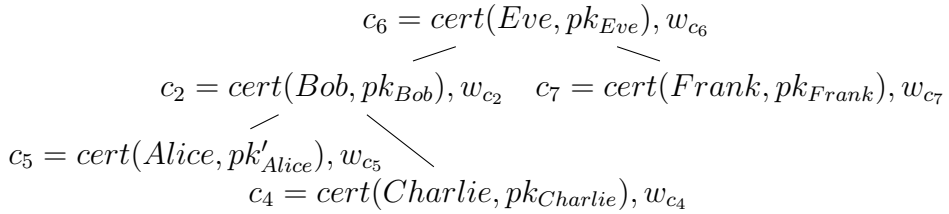
We show a complete chronTree, searchTree and accTree in Figure 4.4, and then update it with one revoked certificate and two new certificates, resulting in new chronTree, searchTree and accTree in Figure 4.5.



(a) The chronTree stores the certificates c_i in the chronological order (the higher the value of i, the more recent the certificate c_i is). The certificates $c_1 = cert(Alice, pk_{Alice})$, $c_2 = cert(Bob, pk_{Bob})$, $c_3 = cert(Alice, \texttt{null})$, $c_4 = cert(Charlie, pk_{Charlie})$, $c_5 = cert(Alice, pk'_{Alice})$, $c_6 = cert(Eve, pk_{Eve})$ and $c_7 = cert(Frank, pk_{Frank})$ are stored in the order they are issued (or revoked). When the certificate c_1 is revoked, another certificate c_3 is inserted in the chronTree having `null` as the public key of *Alice*. The leaf-nodes of the chronTree contain items of the form $x = (c, A, digST)$, where A is the accumulation value of the updated set X after inserting the certificate c or revoking the active certificate of the corresponding user.

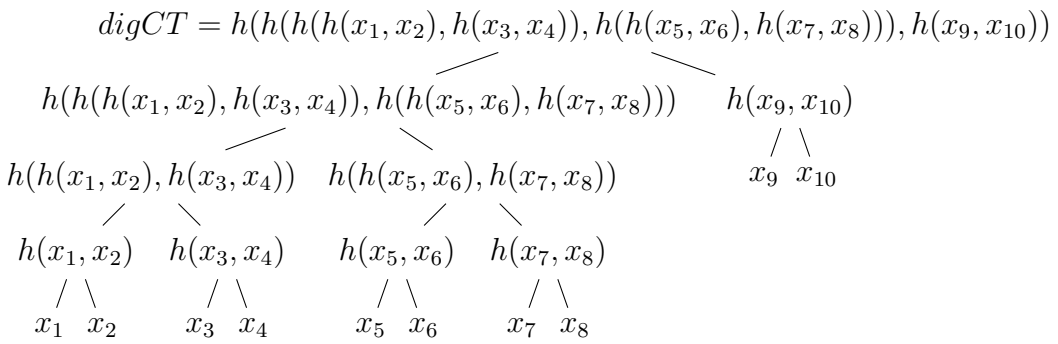


(b) The searchTree stores the certificates in the lexicographic order of the users.

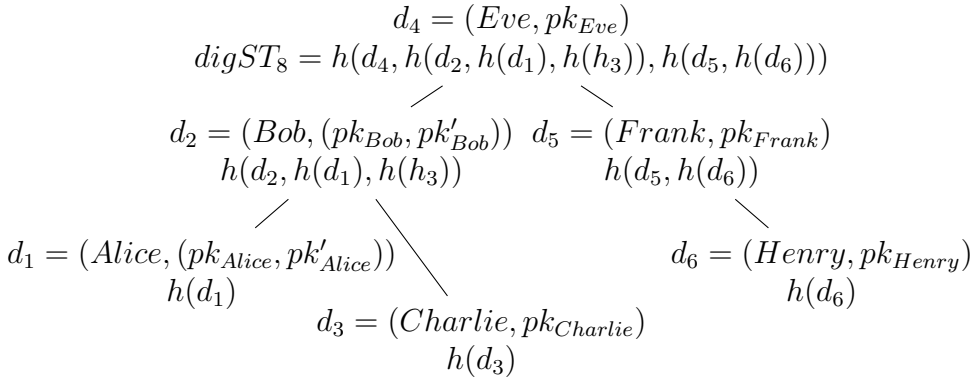


(c) The accTree stores the elements of X (the set of active or current certificates) and their corresponding membership (in X) witnesses.

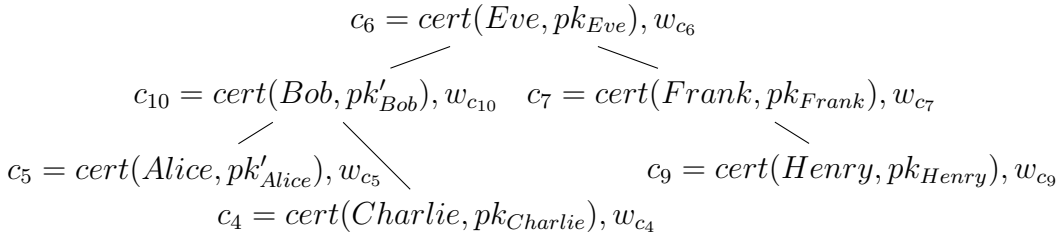
Figure 4.4: The structures of chronTree, searchTree and accTree used in our scheme.



(a) Certificate c_2 is revoked and another certificate $c_8 = cert(Bob, \texttt{null})$ is inserted in chronTree having `null` as the public key of *Bob*. Then certificates $c_9 = cert(Henry, pk_{Henry})$ and $c_{10} = cert(Bob, pk'_{Bob})$ corresponding to $Henry$ and *Bob* respectively are inserted.



(b) New node for $Henry$ is created and its public key is stored. For *Bob*, its new public key is inserted in the list of public keys corresponding to $Bob's$ node in searchTree.



(c) Revoked certificate $cert(Bob, pk_{Bob})$ is removed from accTree and new certificates c_9 and c_{10} are inserted.

Figure 4.5: The updated structures of chronTree, searchTree and accTree after revoking certificate $c_2 = cert(Bob, pk_{Bob})$ and inserting new certificates $c_9 = cert(Henry, pk_{Henry})$ and $c_{10} = cert(Bob, pk'_{Bob})$.

Chapter 5

Security and Performance Analysis

5.1 Notations

Symbol	Definition
λ	Security Parameter
$\|data\|$	Size of data item stored in each node of the searchTree
$\mathcal{U}$	Space of user-identifiers
N	Maximum size of the list of public keys corresponding to any user in searchTree
pk_{size}	Size of a public key
$hash_{in}$	Size of the message input to the hash function h in searchTree

Table 5.1: Notations used for analysis.

5.2 Security Analysis

We assume that auditor and monitor are honest while log maintainer and certificate authority (CA) are dishonest. We formalize the security analysis of our scheme in following lemma.

Lemma 1 *Let A be an honest auditor and certificate authority (CA) issues a fake certificate, $c = cert(u, pk'_u)$, for a domain owner u. If it is not logged in public log, then A will reject the certificate.*

Proof:

The above lemma holds because A accepts the certificate $c = cert(u, pk'_u)$ only when it is accompained by a proof of presence of the certificate in the log. As the certificate c is not present in the log, therefore, the log maintainer will fails to provide proof of presence of the certificate. □

Lemma 2 *Let A be an auditor and D be an domain owner. Certificate authority (CA) issues a fake certificate, $c = cert(u, pk'_u)$, for a domain owner u. If it is present in public log, then an monitor will be able to see this mis-issue immediately and will report this problem.*

Proof:

The above lemma holds because when an auditor, A, asks for the certificate for a domain owner, u, then auditor will get the fake certificate $c = cert(u, pk'_u)$ along with the proof of presence of the certificate in the log. As certificate c is present in the log, then the domain owner u while monitoring the log will see that new certificate is issued for his domain which is not requested by him. Therefore, domain owner u will report this problems and asks CA for revocation of the fake certificate c. □

Lemma 3 *Let a certificate authority (CA) colludes with the log maintainer. The CA issues a fake certificate, $c = cert(u, pk'_u)$, for a domain owner u. If the certificate c is not present in the log and the log maintainer provides a proof of currency of a certificate c, then an auditor, A, will able to see that the proof is not correct.*

Proof:

According to our construction, the trapdoor information s used in the accumulator is secret with the log maintainer (see Section 2.5.2). As the log maintainer colludes with CA, then the log maintainer can use different trapdoor information s' to generate the fake membership witness, w_c, for the certificate, $c = cert(u, pk'_u)$ which is not present in the log. When an auditor, A, gets the certificate c along with proof of currency, w_c, then will try to verify the membership witness by checking the equation

$$e(w_c, g^c \cdot g^s) \stackrel{?}{=} e(A(X), g).$$

As bilinear-map accumulator used in our construction is secure, then the value $g^{s'}$ will not be same as publically known g^s. Therefore, the above equation will not satisfy and proves that the proof of currency is not correct. Hence, above lemma holds and an auditor A will report this problem and say that the log maintainer is not acting correctly. □

Lemma 4 *Let a certificate authority (CA) is dishonest and log maintainer or certificate prover (CP) is honest. The CA issues a fake certificate, $c = cert(u, pk'_u)$, for a domain owner u. If certificate c is not present in the log, then certificate authority will fails to provide a valid proof Π of any type except with some probability negligible in the security parameter λ.*

Proof:

To prove the above lemma, we define the security model and the corresponding security proof:

Security Model We define the security game between the challenger (acting as the Certificate Prover or CP) and the probabilistic polynomial-time adversary (acting as the Certificate Authority or CA) as follows.

- The challenger executes the Setup algorithm to generate the secret information s for the bilinear-map accumulator and the public parameters PP. The public parameters are made available to the adversary.
- Given the public parameters PP, the adversary chooses a sequence of requests (of its choice) defined by $\{(\texttt{reqtype}_i, \texttt{metadata}_i)\}$ for $1 \leqslant i \leqslant q$ (q is polynomial in the security parameter λ). The type of each of these requests, defined by `reqtype`, is an insertion (or revocation) of a certificate c or a query Q of any of the five types described above. The relevant information for each of these requests is stored in the corresponding `metadata`. If the request is for an insertion (or revocation) of a certificate, the challenger performs the necessary changes in the log structure and publishes the updated public parameters PP. If the request is a query Q, the challenger generates the proof corresponding to that particular Q and sends it to the adversary.

Let PP be the final public parameters at the end of the security game mentioned above. The adversary generates a proof Π of one of the five types. The adversary wins the game if the proof Π is not provided by the challenger in the request phase and $\text{Verify}(Q, \Pi, PP) = \texttt{accept}$.

Security Proof Based on the security game described above, we show that an adversary in our scheme cannot win the game except with some probability negligible in the security parameter λ. To be more precise, the adversary cannot produce a valid proof of one of the following types unless it is provided by the challenger itself in the request phase.

- Proof of presence of a certificate (Type 1): A proof of this type consists of two sequences $(dataseq_{type1}, hashseq_{type1})$ in the searchTree. Since the hash function h involved in the computation of the root digest of the searchTree is *collision-resistant* and the root digest is public, the adversary fails to provide such a valid pair of sequences, except with some probability negligible in λ.
- Proof of absence of a certificate (Type 2): A proof of this type is a witness of non-membership in the accumulator set X. Since the bilinear-map accumulator used in our scheme is secure, the adversary cannot forge a proof Π showing that a certificate $c = cert(u, pk_u)$ is absent in the set X (where actually $c \in X$).
- Proof of absence of a user (Type 3): The proof is similar to the proof of Type 1. A proof consists of two sequences $(dataseq_{type3}, hashseq_{type3})$ in the searchTree. Since the hash function h involved in the computation of the root digest of the

searchTree is *collision-resistant* and the root digest is public, the adversary fails to forge a proof of Type 3, except with some probability negligible in λ.

- Proof of extension (Type 4): The proof of extension involves the computation of the *collision-resistant* hash function h per layer of the chronTree and checking whether the final hash value is equal to $digCT$. Due to the collision-resistance property of h, the adversary cannot forge a proof for extension.

- Proof of currency (Type 5): A proof of this type is a witness of membership in the accumulator set X. Since the bilinear-map accumulator used in our scheme is secure, the adversary cannot forge a proof Π showing that a certificate $c = cert(u, pk_u)$ is present in the set X (where actually $c \notin X$).

□

5.3 Performance Analysis

5.3.1 Asymptotic Analysis

Let n be the total number of certificates in the log structure, t be the total number of users and m be the total number of active certificates (the size of the accumulator set X).

Cost of Insertion and Revocation In the chronTree, the cost of the insertion or revocation of a certificate is $O(\log n)$ as the new leaf-node is to be inserted to the right of the chronTree and the new root digest $digCT$ is to be computed. In the searchTree, the cost of the insertion or revocation of a certificate is $O(\log t)$ since searching for the node corresponding to the particular user (and computing the updated root digest $digST$) takes $O(\log t)$ time. In the accTree (maintained for the accumulator X), an insertion or revocation takes $O(m)$ time as the new membership witness is to be updated (and stored) for each element of the set X. Although the cost of revocation is $O(m)$ for the bilinear-map accumulator, we get constant size proofs (with constant verification cost) related to revocation transparency (discussed in the following sections). However, this trade-off is justified as revocation is done by the powerful log server (CP), whereas the proof is verified by a lightweight auditor (or user). In Table 5.2, we have summarized the cost of insertion and revocation in all the three trees maintained by the log maintainer.

Parameters of a Proof For each type of proof, we consider the following parameters: the size of the proof, the computation cost for the proof and the verification time for the proof.

	Insertion	**Revocation**
chronTree	O($\log n$)	O($\log n$)
searchTree	O($\log t$)	O($\log t$)
accTree	O(m)	O(m)

Table 5.2: Cost of Insertion and Revocation

- Proof of presence of a certificate (Type 1): The proof consists of the sequences ($dataseq_{type1}$, $hashseq_{type1}$) in the searchTree. The size of each of these sequences, the time taken to generate them and the time taken to verify them with respect to the root digest $digST$ — all are $O(\log t)$.

- Proof of absence of a certificate (Type 2): The size of a non-membership (in X) witness is $O(1)$. The computation time of a proof is $O(m)$ as the calculation of the non-membership witness requires $O(m)$ multiplications in $\mathbb{Z}_p^*$. The verification time for a proof is $O(1)$ as it involves only two pairing operations.

- Proof of absence of a user (Type 3): The proof is similar to the proof of Type 1. Here, the proof consists of the sequences ($dataseq_{type3}$, $hashseq_{type3}$) in the searchTree. Thus, the size of a proof, the time taken to generate it and the time taken to verify it with respect to the root digest $digST$ — all are $O(\log t)$.

- Proof of extension (Type 4): The proof consists of the sequence $(h_1, h_2, \ldots)$ in the chronTree. The size of the sequence, the time taken to generate it and the time taken to verify it with respect to the root digest $digCT$ — all are $O(\log n)$.

- Proof of currency (Type 5): The size of a membership (in X) witness is $O(1)$. The search for a node containing the certificate c in the accTree takes $O(\log m)$ time, and it takes $O(1)$ time to retrieve the precomputed witness w_c stored at that node. Thus, the computation time of a proof is $O(\log m)$. The verification time for a proof is $O(1)$ as it involves only two pairing operations.

We compare our construction with the existing schemes for certificate transparency based on the parameters described above. The comparison is summarized in Table 5.3. This comparison is based on the parameters: the size of a proof, the cost for computing a proof and the cost for verifying a proof. † For a proof of presence, a certificate is searched in the list of the most recent N public keys of the corresponding user u. This list is maintained in a node (associated with u) of the searchTree. Instead, if we want to search the certificate in the chronTree (which includes all the historical certificates), then the value of this parameter is exactly the same as that for Google [25].

Method	Parameters	Proof of Presence (Type 1)	Proof of Absence of a Certificate (Type 2)	Proof of Absence of a User (Type 3)	Proof of Extension (Type 4)	Proof of Currency (Type 5)
Google [25]	Proof Size	O(log n)	-	-	O(log n)	-
	Cost of Proof Computation	O(n)	-	-	O(log n)	-
	Cost of Proof Verification	O(log n)	-	-	O(log n)	-
Ryan [33]	Proof Size	-	-	O(log t)	O(log n)	O(log t)
	Cost of Proof Computation	-	-	O(log t)	O(log n)	O(log t)
	Cost of Proof Verification	-	-	O(log t)	O(log n)	O(log t)
Our Construction	Proof Size	O(log t)†	O(1)	O(log t)	O(log n)	O(1)
	Cost of Proof Computation	O(log t)†	O(m)	O(log t)	O(log n)	O(log m)
	Cost of Proof Verification	O(log t)†	O(1)	O(log t)	O(log n)	O(1)

Table 5.3: Comparison of our scheme with the existing certificate transparency schemes.

5.3.2 Performance Evaluation

Bilinear Setting and Hash Function

In general setting, we take the bilinear pairing function $e : G_1 \times G_2 \to G_T$ with parameters $(p, g_1, g_2, G_1, G_2, G_T)$, where $|G_1| = |G_2| = |G_T| = p = \Theta(2^{2\lambda})$ and g_1, g_2 are generators of the groups G_1 and G_2, respectively. We take $\lambda = 128$. Practical constructions of pairings are done on elliptic (or hyperelliptic) curves defined over a finite field. We write $E(\mathbb{F}_q)$ to denote the set of points on an elliptic curve E defined over the finite field $\mathbb{F}_q$. Then G_1 is taken as a subgroup of $E(\mathbb{F}_q)$, G_2 is taken as a subgroup of $E(\mathbb{F}_{q^{k'}})$ and G_T is taken as a subgroup of $\mathbb{F}^*_{q^{k'}}$, where k' is the embedding degree [23, 20, 34]. For the value of the security parameter λ up to 128, Barreto-Naehrig (BN) curves [7, 18] are suitable for our scheme. In this setting, each of the elements of $\mathbb{Z}^*_p$ and G_1 is of size 256 bits.

We use SHA-256 as the *collision-resistant* hash function h used to compute the root digests corresponding to the chronTree and the searchTree.

Size of a Proof

We calculate the size (in bits) of each type of proof as follows.

- Proof of presence of a certificate (Type 1): The proof consists of the sequences ($dataseq_{type1}$, $hashseq_{type1}$) in the searchTree. If SHA-256 is used as the hash function h, then the size of $hashseq_{type1}$ is at most $256 \log t$ bits. Let the size of the data item stored in each node of the searchTree be denoted by $|data|$. Then, $|data| \leqslant \log |\mathcal{U}| + N \cdot pk_{size}$, where $\mathcal{U}$ is the space of user-identifiers, N is the maximum size of the list of public keys corresponding to any user and

pk_{size} is the size of a public key. Thus, the size of a proof (in bits) is at most

$$\log t(256 + \log |\mathcal{U}| + N \cdot pk_{size}) \approx \log t(256 + \log t + N \cdot pk_{size}),$$

where t is the number of users (or nodes) present in the searchTree.

- Proof of absence of a certificate (Type 2): The size of a proof is 512 bits as the size of a non-membership (in X) witness $\hat{w_c} = (w_c \in G_1, v_c \in \mathbb{Z}_p^*)$ is 512 bits.
- Proof of absence of a user (Type 3): The proof is similar to the proof of Type 1. Thus, the size of a proof is $\log t(256 + \log t + N \cdot pk_{size})$ bits, where t is the number of users (or nodes) present in the searchTree.
- Proof of extension (Type 4): The proof consists of a sequence of hash values in the chronTree. Therefore, for SHA-256 used as the hash function h, the size of a proof is at most $256 \log n$ bits, where n is the number of certificates present in the chronTree.
- Proof of currency (Type 5): The size of a proof is 256 bits as the size of a membership (in X) witness $w_c \in G_1$ is 256 bits.

In Table 5.4, we have summarized the size (in bits) of each type of proof provided by the log maintainer. The graphical comparison of the size of the proofs provided by the log maintainer in our scheme with the existing one is represented in Figure 5.1.

Proof	**Size (in bits)**
Proof of presence of a certificate	$\log t(256 + \log \lvert\mathcal{U}\rvert + N \cdot pk_{size}) \approx \log t(256 + \log t + N \cdot pk_{size})$
Proof of absence of a certificate	512
Proof of absence of a user	$\log t(256 + \log \lvert\mathcal{U}\rvert + N \cdot pk_{size}) \approx \log t(256 + \log t + N \cdot pk_{size})$
Proof of extension	$256 \log n$
Proof of currency	256

Table 5.4: Size of the proof in bits.

Cost of Verification of a Proof

For the timing analysis of the pairing operations, we use PandA, a recent software framework for *Pairings and Arithmetic* developed by Chuengsatiansup et al. [14]. The cycle-counts of a pairing operation (Ate pairing) for a 128-bit secure, Type-3 pairing framework involving a pairing-friendly BN curve is 3832644 (taken as the median of 10000 measurements on a 2.5 GHz Intel Core i5-3210M processor [14]). These many

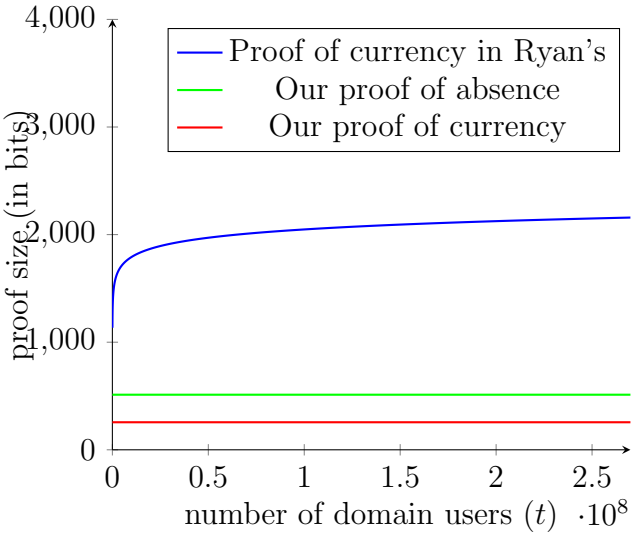


Figure 5.1: Graphical representation of the size of proofs in different schemes.

cycles take approximately 1.53 milliseconds on this processor. On the other hand, we estimate the time taken, on a 2.5 GHz Intel Core i5-3210M processor, to compute SHA-256 from *eBASH*, a benchmarking project for hash functions [10]. We calculate the cost of verification of each type of proof as follows.

- Proof of presence of a certificate (Type 1): The proof consists of the sequences $(dataseq_{type1}, hashseq_{type1})$ in the searchTree. The number of hashes performed for the verification is at most $\log t$. Each hash is performed on a message of the form (d, h_1, h_2), where d is the data item associated with a node and h_1 (or h_2) is the hash value of the left (or right) child of the node. Let the size of the data item stored in each node of the searchTree be denoted by $|data|$. Then, $|data| \leqslant \log |\mathcal{U}| + N \cdot pk_{size} \approx \log t + N \cdot pk_{size}$, where $\mathcal{U}$ is the space of user-identifiers, N is the maximum size of the list of public keys corresponding to any user, pk_{size} is the size of a public key and t is the number of users (or nodes) present in the searchTree.

  For SHA-256 used as the hash function h, the size of h_1 (or h_2) is 256 bits. Therefore, the size of the message input to the hash function h is given by $hash_{in} = (512 + |data|)$ bits. For long messages, computing a single hash value takes 12.71 cycles per byte of the message on a 2.5 GHz Intel Core i5-3210M processor [10]. Consequently, computing each hash value requires $(hash_{in} \cdot (1.59))$ cycles (approximately) that, in turn, takes around $(hash_{in} \cdot (0.64) \cdot 10^{-6})$ milliseconds on this processor. Thus, the total time required for the verification of a proof is around $(\log t \cdot hash_{in} \cdot (0.64) \cdot 10^{-6})$ milliseconds.

- Proof of absence of a certificate (Type 2): The cost of verification of a proof is 3.06 milliseconds as the verification requires two pairing operations.

- Proof of absence of a user (Type 3): The proof is similar to the proof of Type 1. Thus, the total time required for the verification of a proof is around $(\log t \cdot hash_{in} \cdot (0.64) \cdot 10^{-6})$ milliseconds.
- Proof of extension (Type 4): The proof consists of a sequence of hash values in the chronTree. If SHA-256 is used as the hash function h, then the size of the input to h is 512 bits (the hash values of two children). The total number of hashes to be performed is at most $\log n$, where n is the number of certificates present in the chronTree. For messages of length 64 bytes, computing a single hash value takes 29.94 cycles per byte of the message on a 2.5 GHz Intel Core i5-3210M processor [10]. Thus, the total time required for the verification of a proof is around $(\log n \cdot 64 \cdot (11.98) \cdot 10^{-6})$ milliseconds.
- Proof of currency (Type 5): The cost of verification of a proof is 3.06 milliseconds as the verification requires two pairing operations.

In Table 5.5, we have summarized the cost (in milliseconds) of verification of each type of proof by an auditor.

Proof	**Cost of verification (in milliseconds)**
Proof of presence of a certificate	$(\log t \cdot hash_{in} \cdot (0.64) \cdot 10^{-6})$
Proof of absence of a certificate	3.06
Proof of absence of a user	$(\log t \cdot hash_{in} \cdot (0.64) \cdot 10^{-6})$
Proof of extension	$(\log n \cdot 64 \cdot (11.98) \cdot 10^{-6})$
Proof of currency	3.06

Table 5.5: Cost of verification of proofs at an auditor side in milliseconds.

Chapter 6

Future Work and Conclusion

We have developed a scheme which is an extended version of the existing certificate transparency schemes. Apart from handling the existing proofs efficiently, we have proposed a new proof showing the absence of a particular certificate. Some of the proofs, in our scheme, enjoys constant proof-size and constant verification cost. We have also provided a thorough analysis of the security and performance of our scheme. To improve the efficiency of the insertion (or revocation) of certificates by updating the data structures in a batch is a future direction in which this work can be extended. It needs a further investigation to check if our scheme can be applied to detect unauthorized usage of keys and to manage encryption keys to obtain an end-to-end mail encryption system.

Blockchain technology can be used to removes the trust from log maintainer and build decentralized Public Key Infrastructure (PKI). Smart contracts can be developed to store certificates on the blockchain which can be later revoked and retrieve from the blockchain. It can also be done to store certificates "off the blockchain" and write smart contract to access this storage and performs above functionalities.

Bibliography

[1] Diginotar, https://en.wikipedia.org/wiki/DigiNotar

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

[3] Alicherry, M., Keromytis, A.D.: Doublecheck: Multi-path verification against man-in-the-middle attacks. In: IEEE Symposium on Computers and Communications - ISCC 2009. pp. 557–563 (2009)

[4] Amann, B., Vallentin, M., Hall, S., Sommer, R.: Revisiting SSL: A large-scale study of the internet's most trusted protocol. Tech. rep., International Computer Science Institute, Berkeley (2012), http://www.icsi.berkeley.edu/pubs/techreports/ICSI_TR-12-015.pdf

[5] Bari, N., Pfitzmann, B.: Collision-free accumulators and fail-stop signature schemes without trees. In: Advances in Cryptology - EUROCRYPT 1997. pp. 480–494 (1997)

[6] Bari, N., Pfitzmann, B.: Collision-free accumulators and fail-stop signature schemes without trees. In: Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceeding. pp. 480–494 (1997), http://dx.doi.org/10.1007/3-540-69053-0_33

[7] Barreto, P., Naehrig, M.: Pairing-friendly elliptic curves of prime order. In: Preneel, B., Tavares, S. (eds.) Selected Areas in Cryptography - SAC 2006, Lecture Notes in Computer Science, vol. 3897, pp. 319–331. Springer Berlin Heidelberg (2006)

[8] Basin, D.A., Cremers, C.J.F., Kim, T.H., Perrig, A., Sasse, R., Szalachowski, P.: ARPKI: attack resilient public-key infrastructure. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014. pp. 382–393 (2014), http://doi.acm.org/10.1145/2660267.2660298

[9] Benaloh, J.C., de Mare, M.: One-way accumulators: A decentralized alternative to digital sinatures (extended abstract). In: Advances in Cryptology - EUROCRYPT 1993. pp. 274–285 (1993)

[10] Bernstein, D.J., Lange, T.: eBASH: ECRYPT benchmarking of all submitted hashes, `http://bench.cr.yp.to/ebash.html`

[11] Brewster, T.: Diginotar goes bankrupt after hack (2011), `http://www.itpro.co.uk/636244/diginotar-goes-bankrupt-after-hack`

[12] Camenisch, J., Lysyanskaya, A.: Dynamic accumulators and application to efficient revocation of anonymous credentials. In: Advances in Cryptology - CRYPTO 2002. pp. 61–76 (2002)

[13] Caronni, G.: Walking the web of trust. In: IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises - WETICE 2000. pp. 153–158 (2000)

[14] Chuengsatiansup, C., Naehrig, M., Ribarski, P., Schwabe, P.: PandA: Pairings and arithmetic. In: Cao, Z., Zhang, F. (eds.) Pairing-Based Cryptography - Pairing 2013, Lecture Notes in Computer Science, vol. 8365, pp. 229–250. Springer International Publishing (2014)

[15] Damgård, I., Triandopoulos, N.: Supporting non-membership proofs with bilinear-map accumulators. IACR Cryptology ePrint Archive 2008, 538 (2008), `http://eprint.iacr.org/2008/538`

[16] Derler, D., Hanser, C., Slamanig, D.: Revisiting cryptographic accumulators, additional properties and relations to other primitives. In: Topics in Cryptology - CT-RSA 2015, The Cryptographer's Track at the RSA Conference 2015. pp. 127–144 (2015)

[17] Eckersley, P., Burns, J.: Is the SSLiverse a safe place? Chaos Communication Congress (2010), `https://www.eff.org/files/ccc2010.pdf`

[18] Freeman, D., Scott, M., Teske, E.: A taxonomy of pairing-friendly elliptic curves. Journal of Cryptology 23(2), 224–280 (2010)

[19] Fromknecht, C., Velicanu, D., Yakoubov, S.: CertCoin: A NameCoin based decentralized authentication system. Tech. rep., Massachusetts Institute of Technology, MA, USA. 6.857 Class Project (2014), `https://courses.csail.mit.edu/6.857/2014/files/19-fromknecht-velicann-yakoubov-certcoin.pdf`

[20] Galbraith, S.D., Paterson, K.G., Smart, N.P.: Pairings for cryptographers. Discrete Applied Mathematics 156(16), 3113–3121 (Sep 2008)

[21] Google: Certificate transparency, https://www.certificate-transparency.org/

[22] Kim, T.H., Huang, L., Perrig, A., Jackson, C., Gligor, V.D.: Accountable key infrastructure (AKI): a proposal for a public-key validation infrastructure. In: 22nd International World Wide Web Conference, WWW '13, Rio de Janeiro, Brazil, May 13-17, 2013. pp. 679–690 (2013), http://dl.acm.org/citation.cfm?id=2488448

[23] Koblitz, N., Menezes, A.: Pairing-based cryptography at high security levels. In: Smart, N. (ed.) Cryptography and Coding, Lecture Notes in Computer Science, vol. 3796, pp. 13–36. Springer Berlin Heidelberg (2005)

[24] Langley, A.: Public key pinning. Imperial Violet: Adam Langley's Weblog (2011), https://www.imperialviolet.org/2011/05/04/pinning.html

[25] Laurie, B., Langley, A., Kasper, E.: Certificate transparency (June 2013), https://tools.ietf.org/html/rfc6962

[26] Li, J., Li, N., Xue, R.: Universal accumulators with efficient nonmembership proofs. In: International Conference on Applied Cryptography and Network Security - ACNS 2007. pp. 253–269 (2007)

[27] Marlinspike, M.: Trust assertions for certificate keys. Internet Draft (2013), http://tack.io/draft.html

[28] Melara, M.S., Blankstein, A., Bonneau, J., Felten, E.W., Freedman, M.J.: CONIKS: bringing key transparency to end users. In: 24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015. pp. 383–398 (2015), https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/melara

[29] Merkle, R.: A certified digital signature. In: Brassard, G. (ed.) Advances in Cryptology - CRYPTO 1989, Lecture Notes in Computer Science, vol. 435, pp. 218–238. Springer New York (1990)

[30] Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008), http://bitcoin.org/bitcoin.pdf

[31] Nguyen, L.: Accumulators from bilinear pairings and applications. In: Topics in Cryptology - CT-RSA 2005, The Cryptographer's Track at the RSA Conference 2005. pp. 275–292 (2005)

[32] Prince, B.: Comodo attack sparks ssl certificate security discussions (2011), http://www.crn.com/news/security/229400284/comodo-attack-sparks-ssl-certificate-security-discussions.htm

[33] Ryan, M.D.: Enhanced certificate transparency and end-to-end encrypted mail. In: 21st Annual Network and Distributed System Security Symposium - NDSS 2014 (2014), http://www.internetsociety.org/doc/enhanced-certificate-transparency-and-end-end-encrypted-mail

[34] Smart, N.P., Vercauteren, F.: On computable isomorphisms in efficient asymmetric pairing-based systems. Discrete Applied Mathematics 155(4), 538–547 (Feb 2007)

[35] Wendlandt, D., Andersen, D.G., Perrig, A.: Perspectives: Improving SSH-style host authentication with multi-path probing. In: 2008 USENIX Annual Technical Conference. pp. 321–334 (2008)

[36] Yu, J., Cheval, V., Ryan, M.: DTKI: a new formalized PKI with no trusted parties. IACR Cryptology ePrint Archive 2014, 600 (2014), http://eprint.iacr.org/2014/600

[37] Yu, J., Ryan, M., Cremers, C.: How to detect unauthorised usage of a key. IACR Cryptology ePrint Archive 2015, 486 (2015), http://eprint.iacr.org/2015/486