DTKI: a new formalized PKI with no trusted parties

Vincent Cheval, Mark Ryan and Jiangshan Yu* †
School of Computer Science
University of Birmingham, UK

***Abstract*—The security of public key validation protocols for web-based applications has recently attracted attention because of weaknesses in the certificate authority model, and consequent attacks.**

Recent proposals using public logs have succeeded in making certificate management more transparent and verifiable. However, those proposals involve a fixed set of authorities which create a monopoly, and they have heavy reliance on trusted parties that monitor the logs.

We propose a distributed transparent key infrastructure (DTKI), which greatly reduces the monopoly of service providers and removes the reliance on trusted parties. In addition, this paper formalises the public log data structure and provides a formal analysis of the security that DTKI guarantees.

1 Introduction

The security of web-based applications such as e-commerce and webmail depends on the ability of a user's browser to obtain authentic copies of the public keys for the application website. For example, suppose a user wishes to log in to her bank account through her web browser. The web session will be secured by the public key of the bank. If the user's web browser accepts an inauthentic public key for the bank, then the traffic (including login credentials) can be intercepted and manipulated by an attacker.

The authenticity of keys is assured at present by *certificate authorities* (CAs). In the given example, the browser is presented with a public key certificate for the bank, which is intended to be unforgeable evidence that the given public key is the correct one for the bank. The certificate is digitally signed by a CA. The user's browser is pre-configured to accept certificates from certain known CAs. A typical installation of Firefox has about 100 CAs in its database.

Unfortunately, numerous problems with the current CA model have been identified. Firstly, CAs must be assumed to be trustworthy. If a CA is dishonest or compromised, it may issue certificates asserting the authenticity of fake keys; those keys could be created by an attacker or by the CA itself. Secondly, the assumption of honesty does not scale up very well. As already mentioned, a browser typically has hundreds of CAs registered in it, and the user cannot be expected to have evaluated the trustworthiness and security of all of them. This fact has been exploited by attackers [1], [2], [3], [4], [5], [6]. In 2011, two CAs were compromised: Comodo [7] and DigiNotar [8]. In both cases, certificates for high-profile sites were illegitimately obtained, and in the second case, reportedly used in a *man in the middle* (MITM) attack [9].

* Corresponding author.
† The names of authors are in alphabetical order.

Proposed solutions

Several interesting solutions have been proposed to address these problems. We briefly review some of the proposals. For a comprehensive survey, see [10].

Certificate pinning: Certificate pinning [11] is a Google project which addresses the problem of untrustworthy CAs, by restricting in the client browser parameters the set of CAs that are considered entitled to certify the key for a given domain. However, the scalability is a challenge for certificate pinning.

DNS Adoption: *Domain name system (DNS)-based authentication of named entities* (DANE [12], [13]) secures connections between clients and domain servers by binding public keys to domain names. This binding is ensured by only allowing CAs to sign domains in a certain scope, and the scope can be verified by using DNS Security Extensions (DNSSEC) [14]. DANE improves certificate security since the compromised signing key of an authority only harms its sub-domains. However, in DANE, parent domain servers are able to issue fake certificates for their sub-domains without being readily detected.

In 2013, Kasten, Wustrow and Halderman proposed *CAge* [15] to restrict CA's signing scope based on domain name. Their research (based on the data observed and presented in [16]) shows that CAs commonly sign for sites only in a small subset of top-level domains (TLDs). In view of this observation, *CAge* suggests to limit a CA's signing scope by only allowing a CA to issue certificates on a restricted set of TLDs, in order to reduce the damage that a dishonest CA can cause.

Difference Observation: This technique (a.k.a. crowd-sourcing) has been proposed in order to detect untrustworthy CAs, by enabling a browser to obtain warnings if the received certificates are different from those that other people are being offered [17], [18], [19], [20], [21], [22], [23], [24], [25]. In 2008, Wendlandt, Andersen and Perrig proposed *Perspectives* [17] to improve *secure shell* (SSH)-style authentication security by asking different observers to detect inconsistent public keys. In Perspectives, to verify a certificate (received by a client) of a domain, the client asks a set of network notaries for all observed certificates for the domain, and checks the consistency between the received certificate and the certificate observed by each notary, then the client needs to make a decision on whether to trust the received certificate based on the checking result.

In 2009, however, Alicherry and Keromytis [26] pointed out that *Perspectives* has privacy issues since network notaries can get user browsing history. In addition, since network notaries update the certificate information of all servers periodically, clients may reject newly issued certificates while they are not observed by all network notaries, but accept revoked ones. To solve these problems, they proposed *DoubleCheck*, in which to verify a received certificate, the client queries the certificate of the domain server again through *Tor* [27]. However, the use of *Tor* adds additional time cost (up to 15 seconds [23]) for each certificate verification. In addition, if a domain server has multiple certificates, then clients will be likely to get a false positive result.

In 2011, Marlinspike proposed *Convergence* to address privacy concerns in *Perspectives* by enabling client side to cache verified certificates, and by placing a randomly selected notary as a proxy between the client and another selected notary. Convergence effectively prevents numerous CA-based attacks. However, it suffers the same problem as *DoubleCheck* – if a domain server has multiple certificates, and a client and a notary have received different authentic certificates, then the client will receive a false positive result and thus reject authentic certificates.

Trust assertions for certificate keys (TACK) [25] was proposed by Marlinspike and Perrin in 2012. In TACK, a domain server has two pair of keys, namely a TACK key pair and a TLS key pair. The private part of TACK key is used to sign the public part of TLS key. The domain name and the associated public part of TACK key will be "pinned" by clients after they observing the consistent TACK multiple times. The "pin" will be valid for a period equal to the length of time the pair has been observed. A TLS public key will be accepted if it is signed by the private part of THE TACK key, and the public part of the TACK key is included in a valid "pin". TACK releases clients from having to trust CAs. However, since a new TACK key pair will only be accepted if it has already been observed multiple times, the new key pair suffers from an initial unavailability period.

The difference observation technique has effectively solved many CA-based problems. However, the technique cannot distinguish attacks from authentic certificate updates, and may also suffer from an initial unavailability period.

Solutions for *revocation management* of certificates have also been proposed (e.g. Certificate Revocation Lists (CRL) [28], [29], [30] and On-line Certificate Status Protocol (OCSP)). They mostly involve periodically pushing revocation lists to browsers, in order to remove the need for on-the-fly revocation checking. However, these solutions create a window during which the browser's revocation lists are out of date until the next push.

Public log adoption: More recently, solutions involving public append-only logs have been proposed to solve the above mentioned problems. We consider the four leading proposals here.

Sovereign Keys (SK) [31] aims to get rid of browser certificate warnings, by allowing domain owners to establish a long term ("sovereign") key and by providing a mechanism by which a browser can hard-fail if it doesn't succeed in establishing security via that key. The sovereign key is used to cross-sign operational TLS [32], [33] keys, and it is stored in an append-only log on a "timeline server", which is abundantly mirrored. However, in SK, internet users or domain owners have to trust mirrors of timeline servers.

Certificate transparency (CT) [34] is a technique proposed by Google that aims to efficiently detect fake public key certificates issued by corrupted certificate authorities, by making certificate issuance transparent. The core idea is that an append-only public log is maintained, showing all the certificates that have been issued. Web browsers using the log can obtain two types of verifiable cryptographic proofs: (a) a proof that the log contains a given certificate, and (b) a proof that a snapshot of the log is an extension of another snapshot (*i.e.*, only appends have taken place between the two snapshot). The time and size for proof generation and verification are logarithmic in the number of certificates recorded in the log. So internet users can verify them easily (in contrast with SK where internet users have to trust what a mirror says). However, in CT internet users still have to trust "monitors" for verifying the behaviour of logs, and CT does not provide an efficient scheme for key revocation.

Accountable key infrastructure (AKI) [35] also uses public logs to make certificate management more transparent. By using a data structure that is based on lexicographic ordering rather than chronological ordering, they solve the problem of key revocations in the log. In addition, AKI prevents attacks that use fake certificates rather than merely detecting such attacks (as in CT). However, as a result, AKI has a strong assumption – CAs, public log maintainers, and validators do not collude together; and heavily relies on third parties called validators to ensure that the log is maintained without improper modifications.

Extended certificate transparency (ECT) [36] is a proposal for managing certificate for end-to-end encrypted email. It proposes an idea to address the revocation problem left open by CT, and the trusted party problem of AKI. It collects ideas from both CT and AKI to provide transparent key revocation, and reduces reliance on trusted parties by designing the monitoring role so that it can be distributed among user browsers. However, ECT can only detect attacks that use fake certificates; it cannot prevent them. In addition, since ECT was proposed for email applications, it does not support the multiplicity of log maintainers that would be required for web certificates.

In public log based systems, efforts have been made to integrate *revocation management* with the certificate auditing. CT introduced revocation transparency (RT) [37] to deal with certificate revocation management; and in AKI, the public log only stores currently valid certificates (revoked certificates are purged from the log). However, the revocation checking process in both RT and AKI are linear in the number of issued certificates making it inefficient. ECT allows efficient proofs of non-revocation, but it does not scale to multiple logs which are required for web certificates.

Remaining problems

A foundational issue is the problem of *monopoly*, or perhaps more accurately, *oligopoly*. The present-day certificate authority model requires that the set of certificate authorities is fixed and known to every browser, which implies an oligopoly. Currently, the majority of CAs in browsers are organisations based in the USA, and it is hard to become a browser-accepted CA because of the strong trust assumption that it implies. This means that a Russian bank operating in Russia and serving Russian citizens living in Russia has to use an American CA for their public key. This cannot be considered satisfactory in the presence of mutual distrust between nations regarding cybersecurity and citizen surveillance, and also trade sanctions which may prevent the USA offering services (such as CA services) to certain other countries.

None of the previously discussed public log based system (SK, CT, AKI or ECT) address this issue. In each of those solutions, the set of log maintainers (and where applicable, timeline servers, validators, etc.) is assumed to be known by the browsers, and this puts a high threshold on the requirements to become a log maintainer (or validators, etc.). Moreover, none of them solve the problem that a multiplicity of log maintainers reduces the usefulness of transparency, since a domain owner has to check each log maintainer to see if it has mis-issued certificates. This can't work if there is a large number of log maintainers operating in different geographical regions, each one of which has to be checked by every domain owner.

A second foundational issue of a different nature is that of analysis and correctness. SK, CT, AKI and ECT are large and complex protocols involving sophisticated data structures, but none of them have been subjected to rigorous analysis. It is well-known that security protocols are notoriously difficult to get right, and the only way to avoid this is with systematic verification. For example, we have identified an attack on ECT which allows the log maintainer to insert fake certificates. This attack is presented in our technical report (appendix A, page 32) [38]. The flaw is easily fixed, but only once it has been identified. It is therefore imperative that the first steps in verification of this kind of protocol are carried out.

The third problem is the management of certificate revocation. As explained previously, existing solutions for managing certificate revocation (e.g. CRL, OCSP, RT) are still unsatisfactory.

This paper

We propose a new public log based architecture for managing certificates, called *Distributed Transparent Key Infrastructure* (DTKI), with the following contributions.

- We identify *anti-monopoly* as an important property for web certificate management which has hitherto not received attention.
- Compared to its predecessors, DTKI is the first system to have all desired features – it minimises the presence of monopolies, prevents attacks that use fake certificates, provides a way to manage certificate revocation, and does not rely on any trusted party.
- We provide a formal analysis of DTKI. We formalise the data structures needed for transparent public logs, and provide rigorous proofs of their properties.

2 Overview of DTKI

Distributed Transparent Key Infrastructure (DTKI) is an infrastructure for managing keys and certificates on the web in a way which is *transparent*, minimises *monopolies*, and eliminates the need for trusted parties. In DTKI, we mainly have the following agents:

Certificate log maintainers (CLM): A CLM maintains a database of all valid and invalid (e.g. expired or revoked) certificates for a particular set of domains for which it is responsible. It commits to digests of its log, and provides efficient proofs of presence and absence of certificates in the log with respect to the digest. CLMs behave transparently: their actions can be verified and therefore they do not require to be trusted.

A mapping log maintainer (MLM): To minimise monopoly, DTKI does not fix the set of certificate logs. The MLM maintains association between certificate logs and the domains they are responsible for. It also commits to digests of the log, and provides efficient proof of current association, and behaves transparently without requiring to be trusted. MLM has a strategic role of determining the authorised CLMs, and therefore is governed by an international panel (e.g. ICANN).

Mirrors: Mirrors are servers that maintain a full copy of the mapping log downloaded from the MLM. In other words, mirrors are distributed copies of the mapping log. Anyone (e.g. ISPs, CLMs, CAs, domain owners) can be a mirror.

Certificate authorities (CA): They check the identity of domain owners, and create certificates for the domain owners' keys. However, in contrast with today's CAs, the ability of CAs in DTKI is limited since the issuance of a certificate from a CA is not enough to convince web browsers to accept the certificate.

In DTKI, each domain owner has two types of certificate, namely TLS certificate and master certificate. Domain owners can have different TLS certificates but can only have one master certificate. A TLS certificate contains the public key of a domain server for a TLS connection, whereas the master certificate contains a public key, called "master verification key". The corresponding secret key of the master certificate is called "master signing key", which is only used to validate a TLS certificate (of the same subject) by signing it. This limits the ability of certificate authorities since without having a valid signature (issued by using the master signing key), the TLS certificate will not be accepted.

After a domain owner obtains a master certificate or a TLS certificate from a CA, he needs to make a registration request to the corresponding CLM to publish the certificate into the log. To do so, the domain owner signs the certificate using the master signing key, and submits the signed certificate to a CLM determined (typically based on the top-level domain) by the MLM. The CLM checks the signature, and accepts the certificate by adding it to the certificate log if the signature is valid. The process of revoking a certificate is handled similarly to the process of registering a certificate in the log.

When establishing a secure connection with a domain server, the browser receives a corresponding certificate and proofs from a mirror of the MLM and a CLM, and verifies the certificate, the proof that the certificate is valid and recorded in the certificate log, and proof that this certificate log is authorised to manage certificates for the domain. Users and their browsers only accept a certificate if the certificate is issued by a CA, and validated by the domain owner, and current in the certificate log.

Fake master certificates or TLS certificates can be easily detected by the domain owner, because the CA will have had to insert it into the log (in order to be accepted by browsers), and is thus visible to the domain owner.

Rather than relying on trusted parties (e.g. monitors in CT and validators in AKI) to verify the healthiness of logs, DTKI uses a crowdsourcing-like way to monitor the log. In particular, the verification work in DTKI can be broken into independent little pieces, and thus can be done by distributing the pieces to users' browsers. In this way, users' browsers can perform randomly-chosen pieces of the monitoring role in the background. Thus, web users can collectively monitor the integrity of the logs.

To avoid the case that attackers create a "bubble" (i.e. an isolated environment) around a victim, we share the same assumption as other existing protocols (e.g. CT and ECT) – we assume that gossip protocols [39] are used to disseminate digests of the log. So, users of logs can detect if a log maintainer shows different versions of the log to different sets of users. Since log maintainers sign and timestamp their digests, a log maintainer that issues inconsistent digests can be held accountable.

DTKI minimises monopolies, by having just one lightweight "governing party" (our mapping log), which is not required to be trusted, only needed for locating the authorised certificate log for given top-level domains, and distributed to mirrors.

3 The public log

DTKI uses append-only logs to record all requests processed by the log maintainer. Our log structure enables log maintainers to efficiently generate some proofs that can be efficiently verified. These proofs mainly include the proof that some data (e.g. a certificate or a revocation request) has or has not been added to the log, that some data is current (e.g. given a certificate in the log, that no revocation request on this certificate has been added to the log), and that a log is extended from a previous version. So, the log maintainer's behaviour is transparent to the public, and the public is not required to blindly trust log maintainers.

This section defines two abstract data structures encapsulating the desired properties, then introduces how to use the data structures to construct our public logs in a concrete manner. The implementation of data structures is presented in the Appendix. More details can be found in our technical report [38].

3.1 Data structures

A chronological data structure is an append-only data structure, i.e. only the operation of adding some data is allowed. With the append-only property, the chronological data structure enables one to prove that a version of the data structure is an extension of a previous version. We use the notion of *digest* to represent a set of data, such that the size of a digest is a constant. For example, a digest could be the hash value of a set of data. We define the chronological data structure as follows.

Definition 1: Let X be a set and d some data. A *chronological data structure* over X is a data structure S with the following operations.

- $contents(S)$ is a sequence of values of X;
- $digest(S)$ is a value of constant size, called the "digest" of S;
- $add(S, d)$ returns a chronological data structure;

such that the following hold.

- for all chronological data structure S', if $contents(S) \neq contents(S')$, then $digest(S) \neq digest(S')$;
- $contents(add(S, d))$ = $contents(S)$ appended with d;

Moreover, there exists two boolean procedures $\mathsf{VerifPoP}_c$ and $\mathsf{VerifPoE}_c$, whose computation time is linear in the size of their inputs, such that:

- for all d in X, we have d in $contents(S)$, if and only if there exists a value p of size $O(\log(|contents(S)|))$, called *proof of presence of* d *in* $digest(S)$, such that $\mathsf{VerifPoP}_c(digest(S), d, p) = \mathsf{true}$; and
- for all value dg' with integer N', we have that there exists a chronological data structure S' such that $dg' = digest(S')$, $N' = |contents(S')|$, and $contents(S')$ is an initial subsequence of $contents(S)$, if and only if there exists a value p of size $O(\log(N))$, where $N = |contents(S)|$, called *proof of extension of* (dg', N') *into* (dg, N), such that $\mathsf{VerifPoE}_c((dg', N'), (dg, N), p) = \mathsf{true}$.

Intuitively, with the chronological data structure, one can run $\mathsf{VerifPoP}_c$ ("verify proof-of-presence") to efficiently verify the proof of presence that some data d is included in a set $contents(S)$ represented by the corresponding digest; and can run $\mathsf{VerifPoE}_c$ ("verify proof-of-extension) to verify the proof of extension that a sequence of data represented by its digest dg and size N is extended from another sequence of data represented by digest dg' and size N'. In this way, to verify that some data is included in a sequence of data stored in a chronological data structure (of size N), the verifier only needs to download the corresponding digest, and the corresponding proof of presence (with size $O(log(N))$). The verification of proof of extension is similarly efficient. A possible implementation was proposed in CT and is based on binary Merkle hash trees [40].

To verify that a digest really corresponds to a well-formed chronological data structure, one possible solution is letting some trusted parties (e.g. the monitors in the CT) to download the complete sequence of data and compute the corresponding digest, then the web browsers could compare the digest they have received with the digest that the monitor computed. However, since one aim of DTKI is to remove the trusted parties, we use another way, called random verification, to verify the digest of a chronological data structure, without requiring trusted parties.

Definition 2: We say that a chronological data structure is *randomly verifiable* if there exists a boolean procedure $\mathsf{Rand}\exists_C$, whose computation time is linear in the size of its inputs, such that:

- given a value dg, and $N \in \mathbb{N}$, there exists S such that $dg = digest(S)$ and $N = |contents(S)|$, if and only if for all $n \in \{1, \dots, N\}$, there exists a value p of size $O(\log(N))$ such that $\mathsf{Rand}\exists_C(n, dg, N, p) = \mathsf{true}$;
- given the values dg, dg', given the integers $n \leq N' < N$, if there exists p_e such that $\mathsf{VerifPoE}_c((dg', N'), (dg, N), p_e) = \mathsf{true}$, then we have that there exists p' such that $\mathsf{Rand}\exists_C(n, dg', N', p') = \mathsf{true}$, if and only if there exists p such that $\mathsf{Rand}\exists_C(n, dg, N, p) = \mathsf{true}$.

Intuitively, the second condition of the above definition states that if $\mathsf{Rand}\exists_C$ returns true for a given element of a data structure, then the result still holds for the same element in any of its extension, thus does not need to be processed again.

Thanks to $\mathsf{Rand}\exists_C$, verifying that some value is a digest of a chronological data structure can be divided into some smaller verifications; and the time and size needed for each single piece of verification is logarithmic in the size of the data. Thus, these verifications can be distributed to the users' browsers, and the requirement for trusted monitors are eliminated.

The chronological data structure enables one to efficiently verify whether some data was added in the log (by using $\mathsf{VerifPoP}_c$), and to ensure that the log maintainer never remove or modified anything from the log (by using $\mathsf{VerifPoE}_c$). This is useful for our public log since it enables users to verify the history of a log maintainer's behaviours. However, the chronological data structure does not provide all desired features. For example, it is very inefficient to verify that some data (e.g. a revocation request) is not in the chronological data structure (the cost is $O(N)$, where N is the size of the data structure). To provide missing features, we introduce the *ordered data structure*. (To enhance readability, we omitted some technical constrains concerning the ordering relation; these can be found in the definition in our technical report [38].)

Definition 3: Let X be a partially ordered set, d some data. An ordered data structure over X is a data structure S with the following operations.

- $contents_O(S)$ is a set of values of X;
- $digest_O(S)$ is the digest of S;
- $add_O(S,d)$ is the operation to add d into $contents_O(S)$;
- $del_O(S,d)$ is the operation to delete d from $contents_O(S)$;
- $repl_O(S,d,d')$ is to replace data $d \in contents_O(S)$ by some value d';

such that

- for all ordered data structure S', $contents_O(S) \neq contents_O(S')$ if, and only if, $digest_O(S) \neq digest_O(S')$;
- $add_O(S,d)$ succeeds if $d \notin contents_O(S)$, and $contents_O(S') = contents_O(S') \cup \{d\}$; and similar conditions for $del_O(S,d)$ and $repl_O(S,d,d')$.

Moreover, there exists the following five boolean procedures whose computation time is linear in the size.

- $\mathsf{VerifPoP}_O$ (resp. $\mathsf{VerifPoAbs}_O$) is a boolean procedure to verify the proof of presence (resp. absence) of some data in a digest.
- $\mathsf{VerifPoAdd}_O$ (resp. $\mathsf{VerifPoD}_O$ and $\mathsf{VerifPoM}_O$) is a boolean procedure to verify a proof that the addition (resp. deletion and modification) of an element in a digest.

All the procedures presented in Definition 3 are cryptographically sound and complete. Note that the ordered data structure can prove that the current version of the data is extended from a previous version, by using a series of combinations of proof of addition, deletion, and modification. However, it is very inefficient since the number of proofs needed is linear in the number of changes between the two versions. In contrast, proof of extension for chronological data structure is logarithmic.

To have all desired features, namely efficient proofs of presence, absence, and extension, we combine both chronological data structures and the ordered data structure in our log. Intuitively, the log is organised by using the chronological data structure, while each entry of the log is a request with some digests, such that each of the digests represents an ordered data structure storing data for different purposes. For example, a digest could represent an ordered data structure storing all revoked certificates. In this way, the history of the ordered data structure is committed in the chronological data structure. So, we can use the random checking to verify the extension of the ordered data structure.

3.2 Mapping log

To minimise monopoly, DTKI uses multiple certificate logs, and does not fix the set of certificate logs and the mapping between domains and certificate logs. A mapping log is used to record associations between domain names and certificate log maintainers, and can provide efficient proofs regarding the current association. It would be rather inefficient to explicitly associate each domain name to a certificate log, due to the large number of domains. To efficiently manage the association, we use a class of simple regular expressions to present a group of domain names, and record the associations between regular expressions and certificate logs in the mapping log. For example, the mapping might include (*\.org, Clog_1) and ([a-h].*\.com, Clog_1) to mean that certificate log maintainer Clog_1 deals with domains ending *.org* and domains starting with letters from a to h ending *.com*.

Definition 4: A *mapping log* is a randomly verifiable chronological data structure over a set of elements of the form $\mathsf{h}(req, t, dg^s, dg^{bl}, dg^r, dg^i)$, where req is a request received by the mapping log maintainer at time t, and $dg^s, dg^{bl}, dg^r, dg^i$ are digests, as follows:

- req mainly includes $\mathsf{add}(rgx, id)$, $\mathsf{del}(rgx, id)$, $\mathsf{new}(cert)$, $\mathsf{mod}(cert, \mathsf{sign}_{sk}(cert'), \mathsf{sign}_{sk'}(n, \mathsf{h}, t))$, $\mathsf{bl}(id)$, and end, corresponding to a request to add a mapping (rgx, id), to delete a mapping (rgx, id), to add a certificate $cert$ of a new certificate log, to change the certificate $cert$ of a certificate log to the $cert'$, to blacklist the identity id of an existing certificate log, and to close the update request; where rgx is a regular expression; sk and sk' are signing keys associated to the certificate $cert$ and $cert'$, respectively; $cert$ and $cert'$ share the same subject, and n, h, t are some values;
- dg^s is the digest of an ordered data structure storing the identity information of the form $(cert, \mathsf{sign}_{sk}(n, dg, t))$ for the currently active certificate logs, where $cert$ is the certificate for the signing key sk of the certificate log, and n and dg are respectively the size and digest of the certificate log at time t. Data are ordered by the domain name in $cert$.
- dg^{bl} is the digest of an ordered data structure storing the domain names of blacklisted certificate logs. Data are ordered by the stored domain names.
- dg^r is the digest of an ordered data structure storing elements of the form (rgx, id), which represents the mapping from regular expression rgx to the identity id of a certificate log, data are ordered by rex;

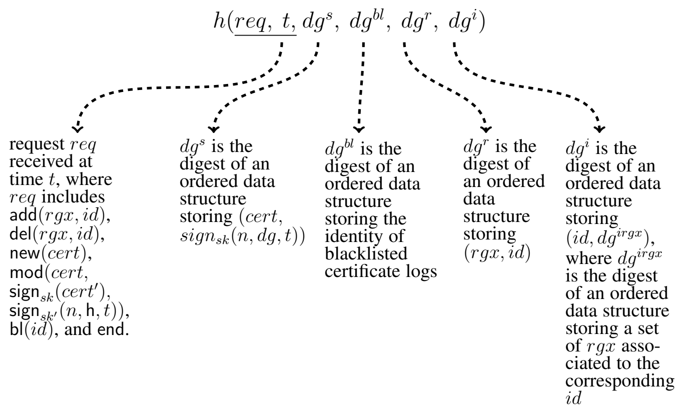

$h(\underline{req,\ t,}\ dg^{s},\ dg^{bl},\ dg^{r},\ dg^{i})$
request req received at time t, where req includes $\mathsf{add}(rgx, id)$, $\mathsf{del}(rgx, id)$, $\mathsf{new}(cert)$, $\mathsf{mod}(cert, \mathsf{sign}_{sk}(cert'), \mathsf{sign}_{sk'}(n, \mathsf{h}, t))$, $\mathsf{bl}(id)$, and end.
dg^{s} is the digest of an ordered data structure storing $(cert, sign_{sk}(n, dg, t))$
dg^{bl} is the digest of an ordered data structure storing the identity of blacklisted certificate logs
dg^{r} is the digest of an ordered data structure storing (rgx, id)
dg^{i} is the digest of an ordered data structure storing (id, dg^{irgx}), where dg^{irgx} is the digest of an ordered data structure storing a set of rgx associated to the corresponding id

Figure 1: A figure representation of the format of each record in the mapping log defined in Definition 4.

- dg^{i} is the digest of an ordered data structure storing elements of the form (id, dg^{irgx}), which represents the mapping from identity id of a certificate log to a digest dg^{irgx} of ordered data structure storing a set of regular expressions, data are ordered by id.

Intuitively, a mapping log is a randomly verifiable chronological data structure, which stores received requests with digests of different ordered data structures representing the status of the log. In particular, each record of the mapping log contains a request req with different digests that is the result of processing the request req on the digest stored in the previous record.

The requests are used for modifying mappings or the existing set of certificate log maintainers. When a request $\mathsf{del}(rgx, id)$ has been processed, the maintainer of certificate log with identity id needs to remove all certificates whose subject is an instance of regular expression rgx; when a request $\mathsf{add}(rgx, id)$ has been processed, the maintainer of certificate log with identity id needs to download all certificates whose subject is an instance of rgx from the previous authorised log maintainer, and adds them into his log. These requests require certificate logs to synchronise with the mapping log; see Section 3.4.

3.3 Certificate logs

A certificate log records certificates of domains, such that the domain name is an instance of the regular expression linked to the certificate log. As we previously mentioned, a domain owner has two types of certificate, namely TLS certificate and master certificate. Each domain owner can only publish one master certificate in the certificate log, but can publish many TLS certificates. The corresponding signing key of the master certificate is used as an extra way to authenticate TLS public keys. In other words, in DTKI, a certificate issued by a CA and recorded in the public log will not be accepted by the users' browsers, unless it has also been signed by the master signing key corresponding to the unique master certificate in the log. In this way, as long as the master certificate in the log is authenticated, the fake certificates won't be accepted. To distinguish the master certificate and TLS certificate, certificate authorities need to specify the certificate type in the certificate extension field.

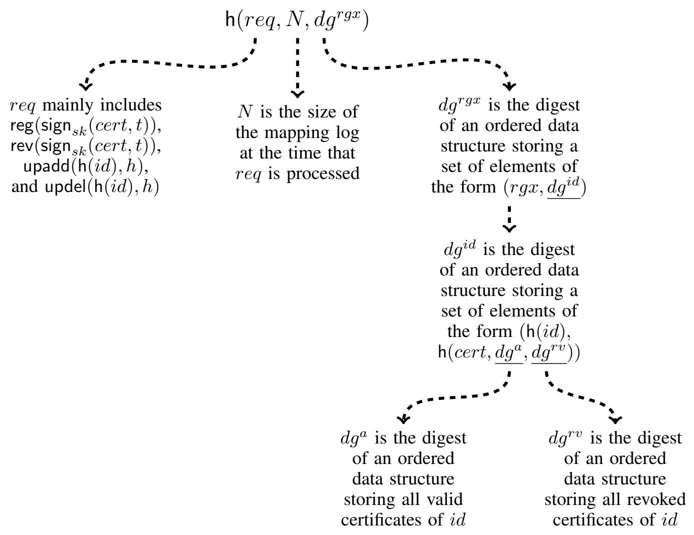

$\mathsf{h}(req, N, dg^{rgx})$
req mainly includes $\mathsf{reg}(\mathsf{sign}_{sk}(cert, t))$, $\mathsf{rev}(\mathsf{sign}_{sk}(cert, t))$, $\mathsf{upadd}(\mathsf{h}(id), h)$, and $\mathsf{updel}(\mathsf{h}(id), h)$
N is the size of the mapping log at the time that req is processed
dg^{rgx} is the digest of an ordered data structure storing a set of elements of the form $(rgx, \underline{dg^{id}})$
dg^{id} is the digest of an ordered data structure storing a set of elements of the form $(\mathsf{h}(id), \mathsf{h}(cert, \underline{dg^{a}}, \underline{dg^{rv}}))$
dg^{a} is the digest of an ordered data structure storing all valid certificates of id
dg^{rv} is the digest of an ordered data structure storing all revoked certificates of id

Figure 2: A figure representation of the format of each record in the certficate log defined in Definition 5.

Definition 5: Let $mlog$ be a mapping log. A *certificate log* is a randomly verifiable chronological data structure over a set of elements of the form $\mathsf{h}(req, N, dg^{rgx})$, where

- req includes $\mathsf{reg}(\mathsf{sign}_{sk}(cert, t))$, $\mathsf{rev}(\mathsf{sign}_{sk}(cert, t))$, $\mathsf{upadd}(\mathsf{h}(id), h)$, and $\mathsf{updel}(\mathsf{h}(id), h)$, corresponding to a request to register and revoke a certificate $cert$ at an agreed time t such that $(cert, t)$ is additionally signed by the master key sk, and update the certificate log by adding and by deleting certificates of identity id according to the changes of $mlog$, respectively.
- N is the size of $mlog$ at the time req is processed;
- dg^{rgx} is the digest of an ordered data structure storing a set of elements of the form (rgx, dg^{id}), where rgx is a regular expression, and dg^{id} the digest of an ordered data structure storing a set of elements of the form $(\mathsf{h}(id), \mathsf{h}(cert, dg^{a}, dg^{rv}))$, where id is an instance of rgx and is the subject of master certificate $cert$, and dg^{a} and dg^{rv} are digests of two ordered data structures each of which stores a set of TLS certificates. In addition, data in the structure represented by dg^{rgx} and dg^{id} are ordered by rgx and $h(id)$, respectively; data in the structure represented by dg^{a} and dg^{rv} are ordered by the subject of TLS certificates.

Intuitively, in a certificate log, each entry contains a request, the size of the mapping log, and a digest dg^{rgx} of an ordered data structure. The size of the mapping log indicates the status of the mapping log when the request is processed; it helps a verifier to run the random checking to verify whether the certificate log is authorised to manage the request with regard to the mapping log. As it was the case in the mapping log, the digest dg^{rgx} in Definition 5 represents the status of the certificate log after processing the request. dg^{rgx} represents all the regular expressions, which the certificate log is associated to, with their corresponding dg^{id}. dg^{id} is the digest of ordered data structure storing elements of the form $(\mathsf{h}(id), \mathsf{h}(cert, dg^a, dg^{rv}))$, such that id is an instance of the regular expression reg, and dg^a and dg^{rv} respectively represent the set of active and revoked certificates. In other words, dg^{id} represents all domains associated to rgx. Since all active and revoked certificates are respectively stored in ordered data structures represented by dg^a and dg^{rv}, one can easily verify whether a certificate is still active by checking the proof of presence of the certificate in dg^a; and a domain owner can easily detect fake certificates published in the certificate log by downloading certificates stored in the set of data represented by dg^a and dg^{rv}.

Note that requests $\mathsf{upadd}(\mathsf{h}(id), h)$ and $\mathsf{updel}(\mathsf{h}(id), h)$ are made according to the mapping log. Even though these modifications are not requested by domain owners, it is important to record them in the certificate log to ensure the transparency of the log maintainer's behaviour. Request $\mathsf{upadd}(\mathsf{h}(id), h)$ states that the certificate log maintainer is authorised to manage certificates for the domain name id from now on, and the current status of certificates for id is represented by h, where $h = \mathsf{h}(cert, dg^a, dg^{rv})$ for some certificate $cert$ and some digest dg^a and dg^{rv} representing the active and revoked certificates of id. h is the value obtained from the certificate log that is previously authorised to manage certificates for domain id. Similarly, request $\mathsf{updel}(\mathsf{h}(id), h)$ indicates that the certificate log cannot manage certificates for domain id any more according to the request in the mapping log.

3.4 Synchronising the mapping log and certificate logs

The mapping log periodically (e.g. every day) publishes a signature $\mathsf{sign}_{sk}(t, dg, N)$, called *signed Mlog timestamp*, on a time t indicating the publishing time, and the digest dg and size N of the mapping log. Mirrors of the mapping log need to download this signed data, and update their copy of the mapping log when it is updated. A *signed Mlog timestamp* is only valid during the issue period (e.g. the day of issue). Note that mirrors can provide the same set of proofs as the mapping log maintainer, because the mirror has the copy of the entire mapping log; but mirrors are not required to be trusted, they do not need to sign anything, and a mirror which changed the log by itself will not be able to convince other users to accept it since the mirror cannot forge the *signed Mlog timestamp*.

When a mapping log maintainer needs to update the mapping log, he requests all certificate log maintainers to perform the required update, and expects to receive the digest and size of all certificate logs once they are updated. After the mapping log maintainer receives these confirmations from all certificate log maintainers, he publishes the series of update requests in the mapping log, and appends an extra constant request end after them in the log to indicate that the update is done.

Log maintainers only answer requests according to their new updated log if the mapping log maintainer has published the update requests in the mapping log. If in the log update period, some user sends requests to the mapping log maintainer or certificate log maintainers, then they give answers to the user according to their log before the update started.

We say that the mapping log and certificate logs are *synchronised*, if certificate logs have completed the log update according to the request in the mapping log. Note that a misbehaving certificate log maintainer (e.g. one recorded fake certificates in his log, or did not correctly update his log according to the request of the mapping log) can be terminated by the mapping log maintainer by putting the certificate log maintainer's identity into the blacklist, which is organised as an ordered data structure represented by dg^{bl} in Definition 4.

4 Distributed transparent key infrastructure

Distributed transparent key infrastructure (DTKI) contains three main phases, namely certificate publication, certificate verification, and log verification. In the certificate publication phase, domain owners can upload new certificates and revoke existing certificates in the certificate log they are assigned to; in the certificate verification phase, one can verify the validity of a certificate; and in the log verification phase, one can verify whether a log behaves correctly.

We present DTKI using the scenario that a TLS user Alice wants to securely communicate with a domain owner Bob who maintains the domain $example.com$.

4.1 Certificate publication

To publish or revoke certificates in the certificate log, the domain owner Bob needs to know which certificate log is currently authorised to record certificates for his domain. This can be done by communicating with a mirror of the mapping log. We detail the protocol for requesting the mapping for Bob's domain.

4.1.1 Request mappings: Bob starts by sending a request with his domain name to a mirror of the mapping log. Upon receiving the request, the mirror locates the certificate $cert$ of the authorised certificate log maintainer and generates the proofs that will be verified by Bob. To do so, the mirror obtains the data of the latest element of its copy of the mapping log, denoted $h = \mathsf{h}(req, t, dg^s, dg^{bl}, dg^r, dg^i)$, and generates the proof of its presence in the digest (denoted dg_{mlog}) of its log of size N. Then, it generates the proof of presence of the element $(cert, \mathsf{sign}_{sk}(n, dg, t))$ in the digest dg^s for some $\mathsf{sign}_{sk}(n, dg, t)$, proving that the certificate log maintainer whose $cert$ belongs to is still active. Moreover, it generates the proof of presence of some element (rgx, id) in the digest dg^r where id is the subject of $cert$ and $example.com$ is an instance of the regular expression rgx, proving that id is authorised to stores the certificates of $example.com$. The mirror then sends to Bob the hash h, the signature $\mathsf{sign}_{sk}(n, dg, t)$, the regular expression rgx, the three generated proofs of presence, and the latest *signed Mlog timestamp* containing the time t_{Mlog}, and digest dg_{mlog} and size N of the mapping log.

Bob first verifies the received *signed Mlog timestamp* with the public key of the mapping log maintainer embedded in the browser, and verifies whether t_{Mlog} is valid. Then Bob checks that $example.com$ is an instance of rgx, and verifies the three different proofs of presence. If all checks hold, then Bob sends the *signed Mlog timestamp* containing $(t'_{Mlog}, dg'_{mlog}, N')$ that he stored during a previous connection, and expects to receive a proof of extension of (dg'_{mlog}, N') into (dg_{mlog}, N). If the received proof of extension is valid, then Bob stores the current *signed Mlog timestamp*, and believes that the certificate log with identity id, certificate $cert$, and size that should be no smaller than n, is currently authorised for managing certificates for his domain.

4.1.2 Certificate publication: The first time Bob wants to publish a certificate for his domain, he needs to generate a pair of master signing key, denoted sk_m, and verification key. The latter is sent to a certificate authority, which verifies Bob's identity and issues a master certificate $cert_m$ for Bob. After Bob receives his master certificate, he checks the correctness of the information in the certificate. The TLS certificate can be obtained in the same way.

Figure 3 presents the process to publish the master certificate $cert_m$. Bob signs the certificate together with the current time t by using the master signing key sk_m, and sends it together with the request to the authorised certificate log maintainer whose signing key is denoted sk_{clog}. The certificate log maintainer checks whether there exists a valid master certificate for $example.com$; if there is one, then the log maintainer aborts the conversation. Otherwise, the log maintainer verifies the validity of time t and the signature.

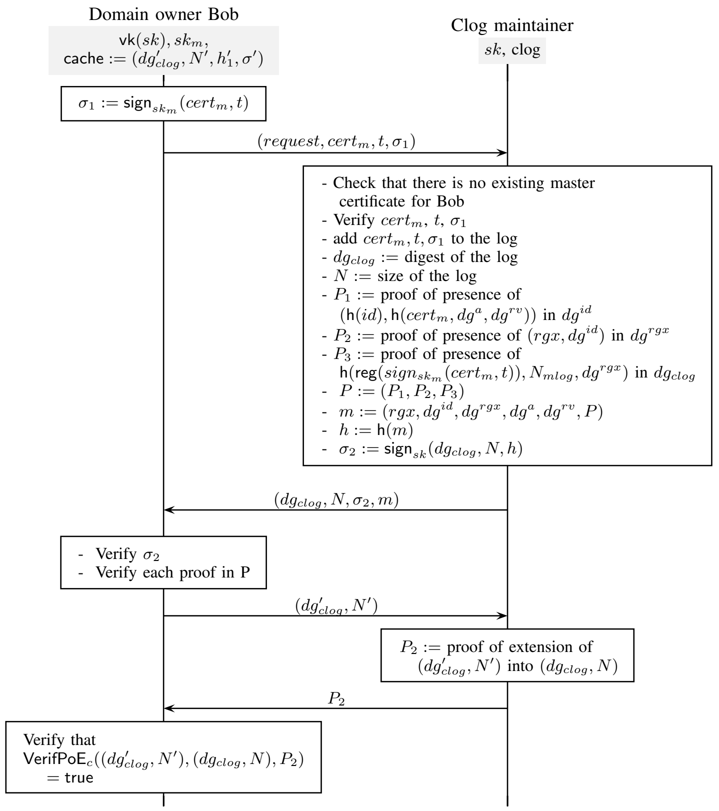
If they are all valid, the log maintainer updates the log, generates the proof of presence of $(\mathsf{h}(id), \mathsf{h}(cert_m, dg^a, dg^{rv}))$ in dg^{id}, (rgx, dg^{id}) in dg^{rgx}, and $\mathsf{h}(\mathsf{reg}(sign_{sk_m}(cert_m, t)), N_{mlog}, dg^{rgx})$ is the last element in the data structure represented by dg_{clog}, where id is the subject of $cert_m$ and an instance of rgx; $\mathsf{reg}(sign_{sk_m}(cert_m, t))$ is the register request to adding $cert_m$ into the certificate log with digest dg_{clog} at time t. The log maintainer then issues a signature on (dg_{clog}, N, h), where N is the size of the certificate log, and $h = \mathsf{h}((rgx, dg^{id}), dg^{rgx}, P)$, where P is the sequence of the generated proofs, and sends the signature σ_2 together with $(dg_{clog}, N, rgx, dg^{id}, dg^{rgx}, dg^a, dg^{rv}, P)$ to Bob. If the signature and the proof are valid, and N is no smaller than the size n contained in the *signed Mlog timestamp* that Bob received from the mirror, then Bob stores the signed (dg_{clog}, N, h), sends the previous stored (dg'_{clog}, N') to the certificate log maintainer, and expects to receive a proof of extension of (dg'_{clog}, N') into (dg_{clog}, N). If the received proof of extension is valid, then Bob believes that he has successfully published the new certificate.

Note that it is important to send (dg'_{clog}, N') after receiving (dg_{clog}, N), because otherwise the log maintainer could learn the digest that Bob has, then give a pair (dg''_{clog}, N'') of digest and size of the log such that $N' < N'' < N$. This may open a window to attackers who wants to convince Bob to use a certificate which was valid in dg''_{clog} but revoked in dg_{clog}.

The process of adding a TLS certificate is similar to the process of adding a master certificate, but the log maintainer needs to verify that the TLS certificate is signed by the valid master signing key corresponding to the master certificate in the log. The process of adding a certificate revocation request is also similar to the process of adding a new certificate. However, for a revocation request with $\mathsf{sign}_{sk_m}(cert, t)$, the log maintainer needs to additionally check that $\mathsf{sign}_{sk_m}(cert, t')$ is already in the log and $t > t'$.

4.2 Certificate verification

When Alice wants to securely communicate with $example.com$, she sends the connection request to Bob, and



Domain owner Bob
$\mathsf{vk}(sk), sk_m$,
$\mathsf{cache} := (dg'_{clog}, N', h'_1, \sigma')$
Clog maintainer
sk, clog
$\sigma_1 := \mathsf{sign}_{sk_m}(cert_m, t)$
$(request, cert_m, t, \sigma_1)$
- Check that there is no existing master certificate for Bob
- Verify $cert_m$, t, σ_1
- add $cert_m, t, \sigma_1$ to the log
- dg_{clog} := digest of the log
- N := size of the log
- P_1 := proof of presence of $(\mathsf{h}(id), \mathsf{h}(cert_m, dg^a, dg^{rv}))$ in dg^{id}
- P_2 := proof of presence of (rgx, dg^{id}) in dg^{rgx}
- P_3 := proof of presence of $\mathsf{h}(\mathsf{reg}(sign_{sk_m}(cert_m, t)), N_{mlog}, dg^{rgx})$ in dg_{clog}
- $P := (P_1, P_2, P_3)$
- $m := (rgx, dg^{id}, dg^{rgx}, dg^a, dg^{rv}, P)$
- $h := \mathsf{h}(m)$
- $\sigma_2 := \mathsf{sign}_{sk}(dg_{clog}, N, h)$
$(dg_{clog}, N, \sigma_2, m)$
- Verify σ_2
- Verify each proof in P
(dg'_{clog}, N')
P_2 := proof of extension of (dg'_{clog}, N') into (dg_{clog}, N)
P_2
Verify that $\mathsf{VerifPoE}_c((dg'_{clog}, N'), (dg_{clog}, N), P_2)$ = true

Figure 3: The protocol presenting how domain owner Bob communicates with certificate log (clog) maintainer to publish a master certificate $cert_m$.

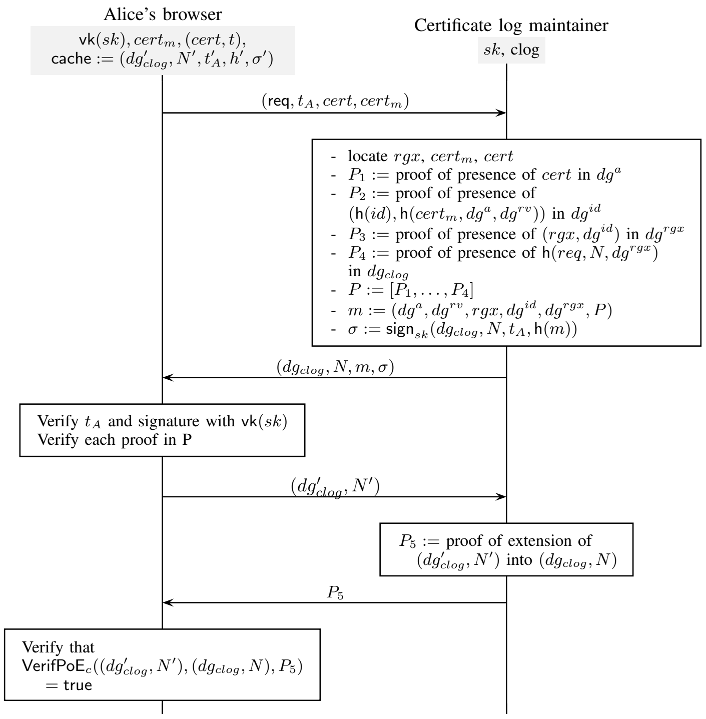
expects to receive a master certificate $cert_m$ and a signed TLS certificate $\mathsf{sign}_{sk_m}(cert, t)$ from him. To verify the received certificates, Alice checks whether the certificates are expired. If both of them are still in the validity time period, Alice requests (as described in 4.1.1) the corresponding mapping from a mirror to find out the authorised certificate log for $example.com$, and communicates with the authorised certificate log maintainer to verify the received certificate.

The Fig. 4 presents the process of verifying a certificate. After Alice learns the identity of the authorised certificate log, she sends the verification request with her local time t_A and the received certificate to the certificate log maintainer. The time t_A is used to prevent replay attacks, and will later be used for accountability. The certificate log maintainer checks whether t_A is in an acceptable time range (e.g. t_A is in the same day as his local time). If it is, then he locates the corresponding (rgx, dg^{id}) in dg^{rgx} in the latest record of his log such that $example.com$ is an instance of regular expression rgx, locates $(\mathsf{h}(id), \mathsf{h}(cert_m, dg^a, dg^{rv}))$ in dg^{id} and $cert$ in dg^a, then generates the proof of presence of $cert$ in dg^a, $(\mathsf{h}(id), \mathsf{h}(cert_m, dg^a, dg^{rv}))$ in dg^{id}, (rgx, dg^{id}) in dg^{rgx}, and $\mathsf{h}(req, N, dg^{rgx})$ is the latest record in the digest dg_{clog} of the log with size N. Then, the certificate log maintainer signs (dg_{clog}, N, t_A, h), where $h = \mathsf{h}(dg^a, dg^{rv}, rgx, dg^{id}, dg^{rgx}, P)$, and P is the set of proofs, and sends $(dg_{clog}, N, dg^a, dg^{rv}, rgx, dg^{id}, dg^{rgx}, \sigma, P)$ to Alice.

After verifying the signature and proofs, Alice sends the previously stored dg'_{clog} with the size N' to the log maintainer, and expects to receive the proof of extension of (dg'_{clog}, N') into (dg_{clog}, N). If they all valid, then Alice replaces the corresponding cache by the signed (dg_{clog}, N, t_A, h) and believes that the certificate is an authentic one.

In order to preserve privacy of Alice's browsing history, instead of asking Alice to query all proofs from the log maintainer, Alice can send the request to Bob who will redirect the request to the log maintainer, and redirect the received proofs from the log maintainer to Alice.

With DTKI, Alice is able to verify whether Bob's domain has a certificate by querying the proof of absence of certificates for $example.com$ in the corresponding certificate log. This is useful to prevent TLS stripping attacks, where an attacker can maliciously convert a HTTPS connection into a HTTP connection.



Alice's browser
$\mathsf{vk}(sk), cert_m, (cert, t)$,
$\mathsf{cache} := (dg'_{clog}, N', t'_A, h', \sigma')$
Certificate log maintainer
sk, clog
$(\mathsf{req}, t_A, cert, cert_m)$
- locate rgx, $cert_m$, $cert$
- $P_1 :=$ proof of presence of $cert$ in dg^a
- $P_2 :=$ proof of presence of $(\mathsf{h}(id), \mathsf{h}(cert_m, dg^a, dg^{rv}))$ in dg^{id}
- $P_3 :=$ proof of presence of (rgx, dg^{id}) in dg^{rgx}
- $P_4 :=$ proof of presence of $\mathsf{h}(req, N, dg^{rgx})$ in dg_{clog}
- $P := [P_1, \ldots, P_4]$
- $m := (dg^a, dg^{rv}, rgx, dg^{id}, dg^{rgx}, P)$
- $\sigma := \mathsf{sign}_{sk}(dg_{clog}, N, t_A, \mathsf{h}(m))$
$(dg_{clog}, N, m, \sigma)$
Verify t_A and signature with $\mathsf{vk}(sk)$
Verify each proof in P
(dg'_{clog}, N')
$P_5 :=$ proof of extension of (dg'_{clog}, N') into (dg_{clog}, N)
P_5
Verify that
$\mathsf{VerifPoE}_c((dg'_{clog}, N'), (dg_{clog}, N), P_5)$
$= \mathsf{true}$

Figure 4: The protocol for verifying a certificate with the corresponding certificate log maintainer.

4.3 Log verification

To verify whether a certificate log authorised for Bob's domain contains fake certificates, Bob needs to periodically check that all certificates for his domain recorded in the certificate log are authentic. To do so, he can check all certificates for his domain stored in the certificate log, and verify the proof that the corresponding digest (i.e. dg^a and dg^{rv}) are recorded in the certificate log. Note that every time when a certificate log maintainer is blacklisted by the mapping log maintainer, Bob needs to run this verification to check his certificates.

In addition, we need to ensure that the mapping log maintainer and certificate log maintainers behaved honestly. In particular, we need to ensure that the mapping log maintainer and certificate log maintainers did update their log correctly according to the request, and certificate log maintainers did follow the latest mappings specified in the mapping log.

These checks can be easily done if there are trusted third parties (TTPs) who can monitor the log. However, since we aim to provide a TTP-free system, DTKI uses a crowdsourcing-like method, based on random checking, to monitor the correctness of the public log. The basic idea of random checking is that each user randomly selects a record in the log, and verifies whether the request and data in this record have been correctly managed. If all records are verified, the entire log is verified. Users only need to run the random checking periodically (e.g. once a day). The full version (with formalisation) of random checking can be found in our technical report. We give a flavour here by providing some examples. Example 1 presents the random checking process to verify the correct behaviour of the mapping log.

Example 1: If the verifier has randomly selected the k^{th} record labelled by $\mathsf{h}(\mathsf{add}(rgx, id), t_k, dg^s_k, dg^{bl}_k, dg^r_k, dg^i_k)$ in the mapping log, then it means that all digests in this record are updated from the $(k-1)^{\text{th}}$ record by adding a new mapping (rgx, id) in the mapping log at time t_k.

Let the label of the $(k-1)^{\text{th}}$ record be $\mathsf{h}(req_{k-1}, t_{k-1}, dg^s_{k-1}, dg^{bl}_{k-1}, dg^r_{k-1}, dg^i_{k-1})$, then to verify the correctness of this record, the verifier should run the following process:

- verify that $dg^s_k = dg^s_{k-1}$ and $dg^{bl}_k = dg^{bl}_{k-1}$; and
- verify that dg^r_k is the result of adding (rgx, id) into dg^r_{k-1} by using $\mathsf{VerifPoAdd}_O$, and id is an instance of rgx; and
- verify that (id, dg^{irgx}_k) is the result of replacing (id, dg^{irgx}_{k-1}) in dg^i_{k-1} by (id, dg^{irgx}_k) by using $\mathsf{VerifPoM}_O$; and
- verify that dg^{irgx}_k is the result of adding rgx into dg^{irgx}_{k-1} by using $\mathsf{VerifPoAdd}_O$.

Note the all proofs required in the above are given by the log maintainer. If the above tests succeed, then the mapping log maintainer has behaved correctly for this record.

The verification on the certificate log is similar to the mapping log. However, there is one more thing needed to be verified – the synchronisation between the mapping log and certificate logs. This verification includes that the certificate log only manage the certificates for domains they are authorised to (according to the mapping log); and if there are modifications on the mapping, then the corresponding certificate log maintainer should add or remove all certificates according to the modified mapping. We present an example to show what a verifier should do to verify that the certificate log was authorised to add or remove a certificate.

Example 2: If the verifier has randomly selected the k^{th} record labelled by $\mathsf{h}(\mathsf{reg}(\mathsf{sign}_{sk}(cert_{TLS}, t)), N_k, dg^{rgx}_k)$ in the certificate log, where dg^{rgx}_k is the digest of ordered sequence of format (rgx, dg^{id}_k), dg^{id}_k is the digest of ordered sequence of format $(\mathsf{h}(id), \mathsf{h}(cert_m, dg^a_k, dg^{rv}_k))$, $cert_m$ is a master certificate, and $cert_{TLS}$ is a TLS certificate. Let dg^{rgx}_k be the digest dg^{rgx} in the $k-1^{\text{th}}$ record, and similarly for dg^{id}_{k-1}, dg^a_{k-1} ,dg^{rv}_{k-1}. Let the subject of $cert_{TLS}$ be id'. The verifier should verify the following tests:

- Verify that $\mathsf{sign}_{sk}(cert_{TLS}, t)$ is correctly signed according to $cert_m$; and
- Verify that $cert_m$ is not expired, and shares the same subject id' with $cert_{TLS}$, and $id' = id$; and
- Verify that dg^a_k is the result of adding $cert_{TLS}$ into dg^a_{k-1}; and
- Verify that dg^{id}_k is the result of replacing $(\mathsf{h}(id), \mathsf{h}(cert_m, dg^a_{k-1}, dg^{rv}_{k-1}))$ by $(\mathsf{h}(id), \mathsf{h}(cert_m, dg^a_k, dg^{rv}_k))$ in dg^{id}_{k-1}; and
- Verify that $dg^{rv}_k = dg^{rv}_{k-1}$; and
- Verify that dg^{rgx}_k is the result of replacing (rgx, dg^{id}_{k-1}) by (rgx, dg^{id}_k) in dg^{rgx}_{k-1}; and
- Verify that (rgx', id'') is in the $dg^r_{N_k}$ in the N^{th}_k element of the mapping log, such that $rgx' = rgx$, and id' is the identity of the certificate log.

If the above tests succeed, then the certificate log maintainer behaves correctly on this record.

4.4 Performance Evaluation

In this section, we measure the cost of different protocols in DTKI.

Assumptions: We assume that the size of a certificate log is 10^8 (the total number of registered domain names currently is 2.71×10^8 [41], though only a fraction of them have certificates). In addition, we assume that the number of stored regular expressions, the number of certificate logs, and the size of the mapping log are 1000 each. (In fact, if we assume a different number or size (e.g. 100 or 10000) for them, it makes almost no difference to the conclusion). Moreover, in the certificate log, we assume that the size of the set of data represented by dg^{rgx} is 10, by dg^{id} is 10^5, by dg^a is 10, and by dg^{rv} is 100. These assumptions are based on the fact that dg^{rgx} represents the set of regular expressions maintained by a certificate log; the dg^{id} represents the set of domains which is an instance of a regular expression; and dg^a and dg^{rv} represent the set of currently valid certificates and the revoked certificates, respectively. Furthermore, we assume that the size of a certificate is 1.5 KB, the size of a signature is 256 bytes, the length of a regular expression and an identity is 20 bytes each, and the size of a digest is 32 bytes.

Space: Based on these assumptions, the approximate size of the transmitted data in the protocol for publishing a certificate is 4 KB, for requesting a mapping is 3 KB, and for verifying a certificate is 5 KB. Since the protocols for publishing a certificate and requesting a mapping are run occasionally, we mainly focus on the cost of the protocol for verifying a certificate, which is required to be run between a log server and a web browser in each secure connection.

By using Wireshark, we[1] measure that the size of data for establishing an HTTPS protocol to login to the internet bank of HSBC, Bank of America, and Citibank are 647.1 KB, 419.9 KB, and 697.5 KB, respectively. If we consider the average size ($\approx$588 KB) of data for these three HTTPS connections, and the average size ($\approx$6 KB) of date for their corresponding TLS establishment connections, we have that in each connection, DTKI incurs 83% overhead on the cost of the TLS protocol. However, since the total overhead of a HTTPS connection is around 588 KB, so the cost of DTKI only adds 0.9% overhead to each HTTPS connection, which we consider acceptable.

Time: Our implementation uses a SHA-256 hash value as the digest of a log and a 2048 bit RSA signature scheme. The time to compute a hash[2] is ≈ 0.01 millisecond (ms) per 1KB of input, and the time to verify a 2048 bit RSA signature is 0.48 ms. The approximate verification time on the user side needed in the protocol for verifying certificates is 0.5 ms.

Hence, on the user side, the computational cost on the protocol for verifying certificates incurs 83% on the size of data for establishing a TLS protocol, and 9% on the size of data for establishing an HTTPS protocol; the verification time on the protocol for verifying certificates is 1.25 % of the time for establishing a TLS session (which is approximately 40 ms measured with Wireshark on the TLS connection to HSBC bank).

5 Security statement and analysis

We consider an adversary who can compromise the private key of all infrastructure servers in DTKI. In other words, the adversary can collude with all log servers and certificate authorities to launch attacks.

Main result: Our security analysis shows that

- if the distributed random checking has verified all required tests, and domain owners have successfully verified their initial master certificates, then DTKI can prevent attacks from the adversary; and
- if the distributed random checking has not completed all required tests, or domain owners have not successfully verified their initial master certificates, then an adversary can launch attacks, but the attacks will be detected afterwards.

The fully detailed analysis is presented in our technical report [38], and we only give a reduced analysis here due to the space limitation.

Consider a scenario where an internet user Alice wants to share some secret data with a domain owner Bob by running the TLS protocol. The main purpose of DTKI is to enable Alice to verify that the certificate she received in the TLS session is indeed a valid certificate of Bob. In DTKI, a valid certificate means that the certificate is active. A certificate is active if the certificate is authentic and not revoked; and a certificate is authentic if the certificate's subject has run the registration protocol to register it.

To formally define an authentic certificate and an active certificate, we define a function keys_B to model the status of all public keys of B. We present time by integers e.g. seconds, consider that all protocols are run within one unit of time, and denote the infinite set of all public keys by $\mathcal{PK}$.

Definition 6: Let B be a domain. A key function keys_B for B is a function from $\mathbb{N}$ to a set of finite sets of elements in $\mathcal{PK} \times \{0, 1\}$ such that for all $pk \in \mathcal{PK}$, for all $t \in \mathbb{N}$, pk

[1]We use the MacBook Air 1.8 GHz Intel Core i5, 8 GB 1600 MHz DDR3.

[2]SHA-256 on 64 byte size block.

occurs at most once in $\mathsf{keys}_B(t)$. Moreover, for all $pk \in \mathcal{PK}$, if there exists t such that pk occurs in $\mathsf{keys}_B(t)$ then:

- either there exists $t_{reg}, t_{rev} \in \mathbb{N}$ such that for all $t < t_{reg}$, pk does not occur in $\mathsf{keys}_B(t)$; and for all $t_{reg} \le t < t_{rev}$, $(pk, 1) \in \mathsf{keys}_B(t)$; and for all $t \ge t_{rev}$, $(pk, 0) \in \mathsf{keys}_B(t)$;
- or there exists $t_{reg} \in \mathbb{N}$ such that for all $t < t_{reg}$, pk does not occur in $\mathsf{keys}_B(t)$; and for all $t \ge t_{reg}$, $(pk, 1) \in \mathsf{keys}_B(t)$.

We say that a public key $pk \in \mathcal{PK}$ is *authentic* (w.r.t domain B) at time t if $(pk, b) \in \mathsf{keys}_B(t)$ for some $b \in \{0, 1\}$; and pk is *active* at the time t if $(pk, 1) \in \mathsf{keys}_B(t)$.

In addition, given user A and log maintainer L, we consider a function $\mathsf{dig}_{\mathsf{sz}(A,L)}$ such that given a time t as input, $\mathsf{dig}_{\mathsf{sz}(A,L)}(t)$ outputs the pair of values (expected to be the digest and size, respectively, of L's log) given by L, and stored in the cache of A's browser at time t. Note that we have $\mathsf{dig}_{\mathsf{sz}(A,L)}(0) = (\mathsf{null}, 0)$ for participants A and L, where null is the null bitstring.

We assume that both Alice and Bob are honest, meaning that they run the protocols of DTKI correctly. We say that (dg, N) represents a chronological log S if $digest(S) = dg$ and $|contents(S)| = N$. We have the following lemma to show that if a participant stores a pair of values after successfully running a protocol with a log maintainer L at time t, and the pair of values are indeed the digest and size of a log, then all previously stored values associated to L are also pairs of digest and size of a historic version of the log.

Lemma 1: Let A be an honest participant, and L a log maintainer. If there exists a time $t \in \mathbb{N}$ and a log S such that $\mathsf{dig}_{\mathsf{sz}(A,L)}(t)$ represents S at time t, then for all $t' \le t$, there exists a log S' such that $\mathsf{dig}_{\mathsf{sz}(A,L)}(t')$ represents S' and $content(S') \subseteq contents(S)$.

Informally, the above lemma holds because A, being honest, will run the verification of proof of extension, and will accept and store the digest at time t only if it was successful; and a valid proof of extension ensures that a chronological data structure represented by the newly received digest is an extension of a chronological data structure represented by a previously stored digest. In addition, the condition that $\mathsf{dig}_{\mathsf{sz}(A,L)}(t)$ represents S at the time t will be verified by the random checking procedure $\mathsf{Rand}\exists_C$.

When a participant wants to register (or revoke, or verify) a certificate, she requests the corresponding certificate log information (e.g, the certificate of the log maintainer, the digest and size of the log) from the mapping log maintainer, then runs the corresponding protocol for registering (or revoking, or verifying) a certificate with the certificate log maintainer. In the protocol, she obtains a digest and size of the certificate log. She should verify that the pair of digest and size is a latter (or the same) version of the pair received from the mapping log maintainer. This is formally described as follows.

Lemma 2: Let A be an honest participant running the protocol for verifying (resp. registering, revoking) a certificate $cert$ at time t. Let M be the mapping log maintainer. If the protocol was successful, then there exists a certificate log maintainer C, called *designated certificate log maintainer*, such that if there exists a mapping log S_M and a certificate log S_C such that $\mathsf{dig}_{\mathsf{sz}(A,M)}(t)$ represents S_M and $\mathsf{dig}_{\mathsf{sz}(A,C)}(t)$ represents S_C, then the following properties hold:

- there exists req, N, dg such that $\mathsf{h}(req, N, dg)$ is the last element of $contents(S_C)$ and $N = |contents(S_M)|$; and
- there exists t', dg^s, dg^{bl}, dg^r and dg^i such that $\mathsf{h}(\mathsf{end}, t', dg^s, dg^{bl}, dg^r, dg^i)$ is the last element of $contents(S_M)$; and
- $t' \le t$; and
- dg^s is the digest of an ordered data structure S_s such that there exists $cert_c$, sk, n' and dg' such that $(cert_c, \mathsf{sign}_{sk}(n', dg', t')) \in contents_O(S_s)$, C is the subject of $cert_c$, $n' \le |contents(S_C)|$ and $(digest(S_C), |contents(S_C)|)$ is an extension of (dg', n'); and
- dg^r is the digest of an ordered data structure S_r such that there exists $(rgx, id) \in contents_O(S_r)$, where id is the identity of C and is an instence of regular expression rgx.

We have the following theorem.

Theorem 1: Let M be the mapping log maintainer; Bob an honest participant having a master certificate and some TLS certificates, and having successfully verified his certificates at time t_B with the designated certificate log maintainer C_B; and $Alice$ an honest participant having successfully verified a certificate $cert$, whose subject is B, by running certificate verification protocol at time $t_A \ge t_B$ with the designated certificate log maintainer C_A. Assume that:

- there exists a mapping log S_M and certificate logs $S_{C_1} \ldots, S_{C_n}$ such that S_M and $S_{C_1} \ldots, S_{C_n}$ are synchronised; and
- there exists $i, j \in \{1, \ldots n\}$ such that $C_A = C_i$ and $C_B = C_j$; and
- $(digest(S_M), |contents(S_M)|) = \mathsf{dig}_{\mathsf{sz}(A,M)}(t_A)$; and
- $(digest(S_{C_A}), |contents(S_{C_A})|) = \mathsf{dig}_{\mathsf{sz}(A,C_A)}(t_A)$; and
- $(digest(S_M), |contents(S_M)|)$ is an extension of $\mathsf{dig}_{\mathsf{sz}(B,M)}(t_B)$; and
- $(digest(S_{C_B}), |contents(S_{C_B})|)$ is an extension of $\mathsf{dig}_{\mathsf{sz}(B,C_B)}(t_B)$; and
- between the time that $\mathsf{dig}_{\mathsf{sz}(B,M)}(t_B)$ and $\mathsf{dig}_{\mathsf{sz}(A,M)}(t_A)$ were generated, no new certificate log maintainer was blacklisted by the mapping log maintainer.

We have that the public key contained in $cert$ is active at the time that $\mathsf{dig}_{\mathsf{sz}(A,C_A)}(t_A)$ was generated.

Loosely speaking, to convince Alice to accept a TLS certificate, an attacker needs to make some fake proofs (detailed in the section 4.2) and to forge a signature corresponding to the master certificate. However, if the master certificate that Alice received is the same as the master certificate Bob published, then the attacker cannot forge such a signature on TLS certificates, though an attacker who colluded with the corresponding certificate log maintainer could forge the proofs (but it would be detected later).

Consider the scenario that an internet user Alice wants to securely communicate with a domain owner Bob who has successfully registered a master certificate and some TLS certificates. Let t_B be the time when Bob has successfully verified his certificates by communicating the mapping log maintainer M and a certificate maintainer C_B. We show how to achieve the conditions listed in the theorem 1 to guarantee the certificate Alice received in the TLS session is active w.r.t. Bob's domain.

After Alice receives a certificate $cert$ from Bob, Alice contacts the mapping log maintainer M and obtains the identity information of the authorised certificate log maintainer C_A for Bob's domain, then runs the certificate verification protocol with C_A. Let $t_A > t_B$ be the time when Alice has successfully verified $cert$ with C_A.

Condition 1 is a property that expresses the existence and synchronisation of logs, and condition 1 states that the logs of the designated log maintainers are part of the synchronised set of logs. Both condition 1 and 1 are ensured in practice by using the distributed random checking. As discussed in the section of log verification (Section 4.3), the full coverage of the random verification can be expected to be achieved because of the large number of internet users.

Conditions 1 and 1 ensure that the mapping log and certificate log maintained by the designated log maintainers are represented by the pairs of digit and size that they sent to Alice; and conditions 1 and 1 indicates that Bob (or Alice) was not in a "bubble" created by the attacker. These conditions can be guaranteed by using the gossip protocol.

The last condition requires that no new certificate log maintainer is blacklisted between time t_b that Bob verified his certificates with C_B and t_A that Alice verified $cert$ with C_A. Since we assume that Bob is an honest participant, then as required by the protocol, he will verify his certificates at least when a certificate log maintainer is blacklisted by the mapping log maintainer.

Thus, thanks to Theorem 1, by the end of the protocol, Alice can be sure that the certificate she received from the TLS session is active.

6 Discussion

Coverage of random checking: As mentioned, several aspects of the logs are verified by user's browsers performing randomly-chosen checks. The number of things to be checked depends on the size of the mapping log and certificate logs. The size of the mapping log mainly depends on the number of certificate logs and the mapping from regular expressions to certificate logs; and the size of certificate logs mainly depends on the number of domain servers that have a TLS certificate. Currently, there are 2.71×10^8 domains [41] (though not every domain has a certificate), and 3×10^9 internet users [42]. Thus, if every user makes one random check per day, then everything will on average, be checked 10 times per day.

Gossip protocol: As mentioned in the overview, to avoid victims being trapped in a "bubble" created by very powerful attackers who controls the network and all service infrastructures such as ISPs and log maintainers, DTKI assumes the existence of a gossip protocol [39] that can be used for users to detect if a log maintainer shows different versions (i.e. different pairs of digest and size) of the log to different sets of users. The gossip protocol allows client browsers to exchange with other users the digest and size of the log that they have received in the DTKI protocols. The gossip protocol provides a means for a browser to identify peers with whom to exchange digests. The mobility of phones and laptops help ensure maximum gossip performance. At any time, a user can request a proof that the pair of digest and size currently offered by the log is an extension of a previous pair of digest and size of the log received from other users via the gossip protocol.

Accountability of mis-behaving parties: The main goal of new certificate management schemes such as CT, AKI and DTKI is to address the problem of mis-issued certificates, and to make the mis-behaving (trusted) parties accountable.

In DTKI, a domain owner can readily check for rogue certificates for his domain. First, he queries a mirror of the mapping log maintainer to find which certificate log maintainers (CLM) are allowed to log certificates for the domain (section 4). Then he examines the certificates for his domain that have been recorded by those CLMs. The responses he obtains from the mirror and the CLMs are accompanied by proofs. If he detects a mis-issued certificate, he requests revocation in the CLM. If that is refused, he can complain to the top-level domain, who in turn can request MLM to change the CLM for his domain (after that, the offending CLM will no longer be consulted by browsers). This request can't be refused because MLM is governed by an international panel. The intervening step, of complaining to the top-level domain, reflects the way domain names are actually managed in practice. Different Top-level domains have different terms and conditions, and domain owners take them into account when purchasing domain names. In DTKI, log maintainers are held accountable because they sign and timestamp their outputs. If a certificate log maintainer issues inconsistent digest, this fact will be detected and the log maintainer can be blamed and blacklisted. If the mapping log misbehaved, then its governing panel must meet and resolve the situation.

In certificate transparency, this process is not as smooth. Firstly, the domain owner doesn't get proof that the list of issued certificates is complete; he needs to rely on monitors and auditors. Next, the process for raising complaints with log maintainers who refuse revocation requests is less clear (indeed, the RFC [34] says that what domain owners should do if they see an incorrect log entry is beyond scope of their document). In CT, a domain owner has no ability to dissociate himself from a log maintainer and use a different one.

AKI addresses this problem by saying that log maintainer that refuses to unregister an entry will eventually lose credibility through a process managed by validators, and will be subsequently ignored. The details of this credibility management are not very clear, but it does not seem to offer an easy way for domain owners to control which log maintainers are relied on for their domain.

Avoidance of monopoly: As we mentioned in the introduction, the predecessors (SK, AKI, E(CT)) of DTKI do not solve a foundational issue, namely *monopoly* (or *oligopoly*). These proposals require that all browser vendors agree on a fixed list of log maintainers and/or validators, and build it into their browsers. This means there will be a large barrier to create a new log maintainer.

CT has some support for multiple logs, but it doesn't have any method to allocate different domains to different logs. In CT, when a domain owner wants to check whether misissued certificates are recorded in logs, he needs to contact all existing logs, and download all certificates in each of the logs, because there is no way to prove to the domain owner that no certificates for his domain is in the log, or to prove that the log maintainer has showed all certificates in the log for his domain to him. Thus, to be able to detect fake certificates, CT has to keep a very small number of log maintainers. This prevents new log providers being flexibly created, creating an oligopoly.

In contrast to its predecessors, DTKI does not have a fixed set of certificate log maintainers (CLMs) to manage certificates for domain owners, and it is easy to add or remove a certificate log maintainer by updating the mapping log. DTKI only has one lightweight governing party, i.e. the mapping log maintainer (MLM), which needs to be built into browsers. However, we minimise the monopoly on the MLM (it is hard to be avoid), because

- the MLM has no bias on certain countries since it is maintained by an international panel; and
- the MLM modifies the mapping log only for strategic and long term reasons; it only periodically (e.g. every day) publishes a *signed Mlog timestamp*; and it is not involved in day-to-day management (which is the work of CLMs and mirrors of the mapping log); and
- the MLM is not required to be trusted by users' browsers.

Additional latency: DTKI introduces additional round-trips in the TLS connecion to verify certificates and prevent potential attacks. This will add some extra latency to the TLS connection. This may be considered justified by the fact that DTKI offers a strong security guarantee.

In fact, the additional latency can be eliminated by delaying the added verification process from the user side. In this case, users obtain a slightly weaker security guarantee: they are still able to verify the authenticity of received certificates afterwards and therefore can detect misissued certificates.

Synchronization concerns: The synchronization among a large number (e.g. thousands) of participants is normally a difficult task. However, in DTKI, the synchronization among the MLM and CLMs is not expected to be a problem. First, the mapping log is rarely changed – it will be changed only if a new CLM has been added or terminated. In the steady state, this is likely to be no more than a few times per year. Second, the MLM can send the corresponding update request to CLMs in advance, and the synchronization process is allowed to take an acceptable time period. During this time period, users will use the current logs until all logs are synchronised. Third, the MLM can terminate a CLM that has failed to update on time (e.g. have not finished the update process in a certain time period). So, in a long run, all parties will be able to do their work properly.

7 Conclusions and future work

Sovereign keys (SK), certificate transparency (CT), accountable key infrastructure (AKI), and enhanced certificate transparency (ECT) are recent proposals to make public key certificate authorities more transparent and verifiable, by using public logs. CT is currently being implemented in servers and browsers. Google is building a certificate transparency log containing all the current known certificates, and is integrating verification of proofs from the log into the Chrome web browser.

Unfortunately, as it currently stands, CT risks creating a monopoly or small oligopoly of log maintainers (as discussed in section 6), of which Google itself will be a principal one. Therefore, adoption of CT risks investing more power about the way the internet is run in a company that arguable already has too much power.

In this paper we proposed DTKI – a TTP-free public key validation system using an improved construction of public logs. DTKI can prevent attacks based on mis-issued certificates, and minimises undesirable oligopoly situations by using the mapping log. In addition, we formalised the public log structure and its implementation; such formalisation work was missing in the previous systems (i.e. SK, CT, AKI, and ECT). Since devising new security protocols is notoriously error-prone, we provide a formalisation of DTKI, and correctness proofs.

References

[1] P. Eckersley, "Iranian hackers obtain fraudulent HTTPS certificates: How close to a web security meltdown did we get?" Electronic Frontier Foundation, 2011. [Online]. Available: https://www.eff.org/deeplinks/2011/03/iranian-hackers-obtain-fraudulent-https

[2] J. Leyden, "Trustwave admits crafting SSL snooping certificate: Allowing bosses to spy on staff was wrong, says security biz," The Register, 2012. [Online]. Available: www.theregister.co.uk/2012/02/09/tustwave_disavows_mitm_digital_cert

[3] "MS01-017: Erroneous Verisign-issued digital certificates pose spoofing hazard," Microsoft Support. [Online]. Available: http://support.microsoft.com/kb/293818

[4] P. Roberts, "Phony SSL certificates issued for Google, Yahoo, Skype, others," Threat Post, March 2011. [Online]. Available: http://threatpost.com/phony-ssl-certificates-issued-google-yahoo-skype.-others-032311

[5] T. Sterling, "Second firm warns of concern after Dutch hack," Yahoo! News, September 2011. [Online]. Available: http://news.yahoo.com/second-firm-warns-concern-dutch-hack-215940770.html

[6] N. Falliere, L. O. Murchu, and E. Chien, "W32.stuxnet dossier. Technical report, Symantec Corporation," 2011.

[7] J. Appelbaum, "Detecting certificate authority compromises and web browser collusion," Tor Blog, 2011.

[8] "Black tulip report of the investigation into the DigiNotar certificate authority breach," Fox-IT (Tech. Report), 2012.

[9] C. Arthur, "Rogue web certificate could have been used to attack Iran dissidents," The Guardian, 2011.

[10] J. Clark and P. C. van Oorschot, "SSL and HTTPS:revisiting past challenges and evaluating certificate trust model enhancements," in *IEEE Symposium on Security and Privacy*, 2013.

[11] A. Langley, "Public-key pinning," *ImperialViolet* (blog), 2011.

[12] R. Barnes, "Use Cases and Requirements for DNS-Based Authentication of Named Entities (DANE)," RFC 6394 (Informational), Internet Engineering Task Force, Oct. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6394.txt

[13] P. Hoffman and J. Schlyter, "The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA," RFC 6698 (Proposed Standard), Internet Engineering Task Force, Aug. 2012. [Online]. Available: http://www.ietf.org/rfc/rfc6698.txt

[14] S. Weiler and D. Blacka, "Clarifications and Implementation Notes for DNS Security (DNSSEC)," RFC 6840 (Proposed Standard), Internet Engineering Task Force, Feb. 2013. [Online]. Available: http://www.ietf.org/rfc/rfc6840.txt

[15] J. Kasten, E. Wustrow, and J. A. Halderman, "CAge: Taming certificate authorities by inferring restricted scopes," in *Financial Cryptography*, 2013.

[16] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman, "Mining your Ps and Qs: detection of widespread weak keys in network devices," in *Proceedings of the 21st USENIX conference*

on Security symposium, ser. Security'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 35–35. [Online]. Available: http://dl.acm.org/citation.cfm?id=2362793.2362828

[17] D. Wendlandt, D. G. Andersen, and A. Perrig, "*Perspectives:* improving SSH-style host authentication with multi-path probing," in *USENIX Annual Technical Conference*, 2008, pp. 321–334.

[18] P. Eckersley and J. Burns, "Is the SSLiverse a safe place?" Chaos Communication Congress, 2010.

[19] M. Alicherry and A. D. Keromytis, "Doublecheck: Multi-path verification against man-in-the-middle attacks," in *ISCC*, 2009, pp. 557–563.

[20] B. Amann, M. Vallentin, S. Hall, and R. Sommer, "Revisiting SSL: A large-scale study of the internet's most trusted protocol," Technical report, ICSI, 2012.

[21] "The EFF SSL Observatory," https://www.eff.org/observatory.

[22] "Certificate patrol," http://patrol.psyced.org.

[23] C. Soghoian and S. Stamm, "Certified lies: Detecting and defeating government interception attacks against SSL," in *Financial Cryptography*, 2011, pp. 250–259.

[24] M. Marlinspike, "SSL and the future of authenticity," in *Black Hat, USA*, 2011.

[25] M. Marlinspike and T. Perrin, "Trust assertions for certificate keys (TACK)," Internet draft, 2012.

[26] M. Alicherry and A. D. Keromytis, "Doublecheck: Multi-path verification against man-in-the-middle attacks," in *ISCC*, 2009, pp. 557–563.

[27] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The second-generation onion router," in *In Proceedings of the 13th USENIX Security Symposium*, 2004, pp. 303–320.

[28] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile," RFC 5280 (Proposed Standard), Internet Engineering Task Force, May 2008, updated by RFC 6818. [Online]. Available: http://www.ietf.org/rfc/rfc5280.txt

[29] R. L. Rivest, "Can we eliminate certificate revocation lists?" in *Financial Cryptography*. Springer, 1998, pp. 178–183.

[30] A. Langley, "Revocation checking and Chrome's CRL," *ImperialViolet* (blog), 2012.

[31] P. Eckersley, "Sovereign key cryptography for internet domains," Internet Draft, 2012.

[32] T. Dierks and E. Rescorla, "The transport layer security (TLS) protocol version 1.2," RFC 5246 (Proposed Standard), Internet Engineering Task Force, Aug. 2008, updated by RFCs 5746, 5878, 6176. [Online]. Available: http://www.ietf.org/rfc/rfc5246.txt

[33] S. Turner and T. Polk, "Prohibiting secure sockets layer (SSL) version 2.0," RFC 6176 (Proposed Standard), Internet Engineering Task Force, Mar. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6176.txt

[34] B. Laurie, A. Langley, and E. Kasper, "Certificate Transparency," RFC 6962 (Experimental), Internet Engineering Task Force, 2013.

[35] T. H.-J. Kim, L.-S. Huang, A. Perrig, C. Jackson, and V. Gligor, "Accountable key infrastructure (AKI): A proposal for a public-key validation infrastructure," in *the 22nd International World Wide Web Conference (WWW 2013)*, 2013.

[36] M. Ryan, "Enhanced certificate transparency and end-to-end encrypted mail," in *NDSS*, 2014.

[37] B. Laurie and E. Kasper, "Revocation transparency," September 2012, google Research.

[38] "Full report corresponding to this submission." [Online]. Available: https://www.dropbox.com/s/fdtn9vf9g0nf3y7/Report.pdf

[39] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. Van Steen, "Gossip-based peer sampling," *ACM Transactions on Computer Systems (TOCS)*, vol. 25, no. 3, p. 8, 2007.

[40] R. C. Merkle, "A digital signature based on a conventional encryption function," in *CRYPTO*, 1987, pp. 369–378.

[41] "Domain name industry brief," Verisign, 2014. [Online]. Available: http://www.verisigninc.com/en_US/innovation/dnib/index.xhtml

[42] "Internet users," Internet Usage Statistics, 2014. [Online]. Available: http://www.internetlivestats.com/internet-users/

Appendix

Implementation of data structures

This section shows the implemetation of the chronological data structure and ordered data structure. We give some examples to show how the proofs could be done. Full details can be found in our technical report. We consider a secure hash function (e.g. SHA256), denoted h.

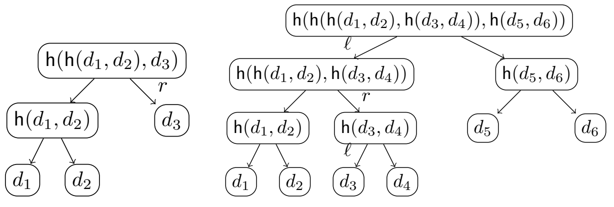
Chronological data structure: The chronological data structure is implemented based on Merkle tree structure that we call *ChronTree*.

A *ChronTree* T is a binary tree whose nodes are labelled by bitstrings such that:

- every non-leaf nodes in T has two children, and is labeled with $\mathsf{h}(t_\ell, t_r)$ where t_ℓ (resp. t_r) is the label of its left child (resp. right child); and
- the subtree rooted by the left child of a node is perfect, and its height is greater than or equal to the height of the subtree rooted by the right child.

Here, a subtree is "perfect" if its every non-leaf node has two children and all its leaves have the same depth.

Note that a ChronTree is a not necessarily a balanced tree. The two trees in Figure 5 are examples of ChronTrees where the data stored are the bitstrings denoted $d_1, \ldots, d_6$.


$\mathsf{h}(\mathsf{h}(d_1, d_2), d_3)$
r
$\mathsf{h}(d_1, d_2)$
d_3
d_1
d_2
$\mathsf{h}(\mathsf{h}(\mathsf{h}(d_1, d_2), \mathsf{h}(d_3, d_4)), \mathsf{h}(d_5, d_6))$
ℓ
$\mathsf{h}(\mathsf{h}(d_1, d_2), \mathsf{h}(d_3, d_4))$
$\mathsf{h}(d_5, d_6)$
r
$\mathsf{h}(d_1, d_2)$
$\mathsf{h}(d_3, d_4)$
ℓ
d_5
d_6
d_1
d_2
d_3
d_4

(a) ChronTree T_a (b) ChronTree T_b

Figure 5: Example of two ChronTrees, T_a and T_b.

Given a ChronTree T with k leaves, we denote by $\mathcal{S}(T) = [d_1, \ldots, d_k]$ the sequence of bitstrings stored in T. Note that a ChronTree is completely defined by the sequence of data stored in the leaves. Moreover, we say that the size of a ChronTree is the number its leaves.

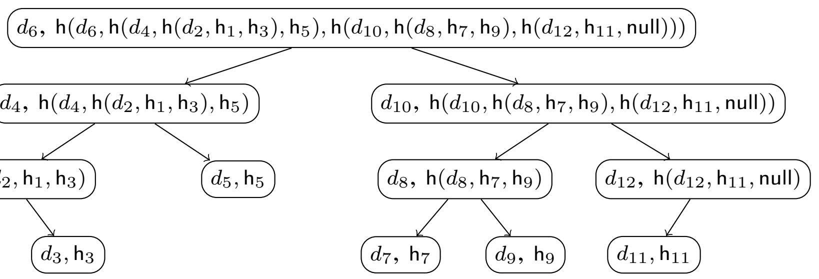
Given a bitstring d and a ChronTree T, the *proof of presence of d in T* exists if there is a leaf n_1 in T labeled by d; and is defined as $(w, [b_1, \ldots, b_k])$ such that:

- w is the position in $\{\ell, r\}^*$ of n_1 (that is, the sequence of left or right choices which lead from the root to n_1), and $|w| = k$; and
- if $n_1, \ldots, n_{k+1}$ is the path from n_1 to the root, then for all $i \in \{1, \ldots, k\}$, b_i is the label of the sibling node of n_i.

Intuitively, a proof of presence of d in T contains the minimum amount of information necessary to recompute the label of the root of T from the leaf containing d.

Example 3: Consider the ChronTree T_b of Figure 5. The proof of presence of d_3 in T_b is the tuple (w, seq) where:

- $w = \ell \cdot r \cdot \ell$


d_6, $\mathsf{h}(d_6, \mathsf{h}(d_4, \mathsf{h}(d_2, \mathsf{h}_1, \mathsf{h}_3), \mathsf{h}_5), \mathsf{h}(d_{10}, \mathsf{h}(d_8, \mathsf{h}_7, \mathsf{h}_9), \mathsf{h}(d_{12}, \mathsf{h}_{11}, \mathsf{null})))$
d_4, $\mathsf{h}(d_4, \mathsf{h}(d_2, \mathsf{h}_1, \mathsf{h}_3), \mathsf{h}_5)$
d_{10}, $\mathsf{h}(d_{10}, \mathsf{h}(d_8, \mathsf{h}_7, \mathsf{h}_9), \mathsf{h}(d_{12}, \mathsf{h}_{11}, \mathsf{null}))$
$d_2, \mathsf{h}(d_2, \mathsf{h}_1, \mathsf{h}_3)$
d_5, h_5
d_8, $\mathsf{h}(d_8, \mathsf{h}_7, \mathsf{h}_9)$
d_{12}, $\mathsf{h}(d_{12}, \mathsf{h}_{11}, \mathsf{null})$
d_1, h_1
d_3, h_3
d_7, h_7
d_9, h_9
d_{11}, h_{11}
Figure 6: An example of a LexTree T_c, where $\mathsf{h}_i = \mathsf{h}(d_i, \mathsf{null}, \mathsf{null})$ for all $i = \{1, 3, 5, 7, 9, 11\}$

- $seq = [d_4, \mathsf{h}(d_1, d_2), \mathsf{h}(d_5, d_6)]$

Note that the size of the proof of presence is logarithmic in the size of the tree; even if the tree grows considerably, the size of the proof does not increase much.

Let T' and T be ChronTrees of size $N' \leq N$ respectively, containing the data $\mathcal{S}(T) = [d_1, \ldots, d_{N'}, \ldots, d_N]$, and $\mathcal{S}(T') = [d_1, \ldots, d_{N'}]$ for some bitstrings $d_1, \ldots, d_{N'}, \ldots, d_N$. Let m be the smallest position of the bits 1 in the binary representation of N'; and let (d, w) be the $(m+1)^{\text{th}}$ node in the path of the node labeled by $d_{N'}$ to the root in T, where d is a bitstring and $w \in \{\ell, r\}^*$ indicates the position. At last, let (w, seq') be the proof of presence of d in T. The proof of extension of T' into T is defined as the sequence seq of bitstrings such that

- if $N' = 2^k$ for some k, then $seq = seq'$; otherwise
- $seq = d :: seq'$, where :: is the concatenation operation.

Example 4: The proof of extensions of T_a into T_b (Figure 5) is the sequence $seq = [d_3, d_4, \mathsf{h}(d_1, d_2), \mathsf{h}(d_5, d_6)]$.

While a proof of presence is the minimal amount of information necessary to recompute the hash value of a ChronTree from the leaf containing some particular data, the proof of extension is the minimal amount of information necessary to recompute the hash value of ChronTree T from the hash value of a ChronTree T' where T is an extension of T'. Intuitively, the proof of extension of a ChronTree T' into a ChronTree T is the proof of presence in T of the last inserted data of T', *i.e.* $d_{N'}$ when $\mathcal{S}(T') = [d_1, \ldots, d_{N'}]$. With this proof and the sizes of both trees, we can reconstruct the label of the root T but also the label of the root of T' as means to verify the proof of extension. Note that when $N' = 2^k$ for some k, it implies that the tree T' is perfect and so the label of the root of T' is also a label of a node in T. Therefore, to reconstruct the label of the root of T, we only need a fragment of the proof of presence of $d_{N'}$ in T.

Example 5: Coming back to Example 4, consider the bitstrings $h_b = \mathsf{h}(\mathsf{h}(\mathsf{h}(d_1, d_2), \mathsf{h}(d_3, d_4)), \mathsf{h}(d_5, d_6))$ and $h_a = \mathsf{h}(\mathsf{h}(d_1, d_2), d_3)$. seq proves the extension of h_a of size 3 into h_b of size 6. Figure 5 is the graphical representation of the verification of seq given h_a and h_b. In particular, $(\ell \cdot r \cdot \ell, [d_4, \mathsf{h}(d_1, d_2), \mathsf{h}(d_5, d_6)])$ proves the presence of d_3 in h_b and $(r, [\mathsf{h}(d_1, d_2)])$ proves the presence of d_3 in h_a.

Ordered data structure: The ordered data structure is implemented as the combination of a binary search tree and a Merkle tree. The idea is that we can regroup all the information about a subject into a single node of the binary search tree, and while being able to efficiently generate and verify the proof of presence. We consider a total order on bitstrings denoted $\leq$. This order could be the lexicographic order in the ASCII representations but it could be any other total order on bitstrings. The implementation is called $LexTree$.

A *LexTree* T is a binary search tree over pairs of bitstrings

- for all two pairs (d, h) and (d', h') of bitstrings in T, (d, h) occurs in a node left of the occurrence of (d', h') if and only if $d \leq d'$ lexicographically;
- for all nodes $n \in T$, n is labeled with the pair $(d, \mathsf{h}(d, h_\ell, h_r))$ where d is some bistring and (d_ℓ, h_ℓ) (resp. (d_r, h_r)) is the label of its left child (resp. right child) if it exists; or the constant null otherwise.

Note that contrary to a ChronTree, the same set of data can be represented by different LexTrees depending on how the tree is balanced. To avoid this situation, we assume that there is a pre-agreed way for balancing trees.

Example 6: The tree in Figure 6 is an example of LexTree where $d_1 \leq d_2 \leq \ldots \leq d_{12}$.

Example 7: Consider the LexTree T of Figure 6. The proof of presence of d_8 in T is the tuple $(h_\ell, h_r, seq_d, seq_h)$ where:

- $h_\ell = \mathsf{h}_7$ and $h_r = \mathsf{h}_9$; and
- $seq_d = [d_{10}, d_6]$
- $seq_h = [\mathsf{h}(d_{12}, \mathsf{h}_{11}, \mathsf{null}),\ \mathsf{h}(d_4, \mathsf{h}(d_2, \mathsf{h}_1, \mathsf{h}_3), \mathsf{h}_5)]$

Like in ChronTrees, verifying the proof of presence of some data d in a LexTree T consists of reconstructing the hash value of the root of T.

Example 8: Consider the T_c of Figure 6. Consider some data d such that $d_7 \leq d \leq d_8$. The proof of absence of d in T_c is the tuple $(\mathsf{null}, \mathsf{null}, seq_d, seq_h)$ where:

- $seq_d = [d_7, d_8, d_{10}, d_6]$
- $seq_h = [\mathsf{h}_9,\ \mathsf{h}(d_{12}, \mathsf{h}_{11}, \mathsf{null}),\ \mathsf{h}(d_4, \mathsf{h}(d_2, \mathsf{h}_1, \mathsf{h}_3), \mathsf{h}_5)]$