DTKI: a new formalized PKI with no trusted parties

Jiangshan Yu[1], Vincent Cheval[2] and Mark Ryan[1]

[1]*School of Computer Science, University of Birmingham, UK*
[2]*School of Computing, University of Kent, UK*
Email: jxy223@cs.bham.ac.uk
vincent.cheval@icloud.com
m.d.ryan@cs.bham.ac.uk

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. This means an oligopoly is created. Another problem with current log-based system is their heavy reliance on trusted parties that monitor the logs.
We propose a distributed transparent key infrastructure (DTKI), which greatly reduces the oligopoly 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.

Keywords: PKI; SSL; TLS; key distribution; certificate; transparency; trust; formal verification

1. INTRODUCTION

The security of web-based applications such as e-commerce and web-mail 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 log-in 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 root certificates 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]. See [10] for a survey on CA compromises.

Proposed solutions

Several interesting solutions have been proposed to address these problems. For a comprehensive survey, see [11].

Key pinning mitigates the problem of untrustworthy CAs, by defining in the client browser the parameters concerning the set of CAs that are considered entitled to certify the key for a given domain [12, 13]. However, scalability is a challenge for key pinning.

Crowd-sourcing techniques have 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 [14, 15, 16, 17, 18, 19, 20, 21]. Crowd-sourcing techniques have solved some CA-based problems. However, the technique cannot distinguish between attacks and authentic certificate updates, and may also suffer from an initial unavailability period.

Solutions for revocation management of certificates have also been proposed; they mostly involve periodically pushing revocation lists to browsers, in order to remove the need for on-the-fly revocation checking [22, 23]. However, these solutions create a window during which the browser's revocation lists are out of date until the next push.

More recently, solutions involving public append-only logs have been proposed. We consider the leading proposals here.

Public log-based systems *Sovereign Keys* (SK) [24] 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 [25, 26] keys, and it is stored in an append-only log on a "time-line server", which is abundantly mirrored. However, in SK, internet users and domain owners have to trust mirrors of time-line servers, as SK does not enable mirrors to provide efficient verifiable proofs that the received certificate is indeed included in the append-only log.

Certificate transparency (CT) [27] 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. They improved the idea of SK by using append-only Merkle tree to organise the append-only log. This enables the log maintainer to provide 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. Domain owners can obtain the proof that their certificates are recorded in the log, and provide the proof together with the certificate to their clients, so the clients can get a guarantee that the received certificate is recorded in the log.

Accountable key infrastructure (AKI) [28] 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 uses the "checks-and-balances" idea that allows parties to monitor each other's misbehaviour. So AKI limits the requirement to trust any party. Moreover, AKI prevents attacks that use fake certificates rather than merely detecting such attacks (as in CT). However, as a result, AKI needs a strong assumption — namely, 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.

Certificate issuance and revocation transparency (CIRT) [29] is a proposal for managing certificates 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, CIRT can only detect attacks that use fake certificates; it cannot prevent them. In addition, since CIRT was proposed for email applications, it does not support the multiplicity of log maintainers that would be required for web certificates.

Attack Resilient Public-Key Infrastructure (ARPKI) [30] is an improvement on AKI. In ARPKI, a client can designate n service providers (e.g. CAs and log maintainers), and only needs to contact one CA to register her certificate. Each of the designated service providers will monitor the behaviour of other designated service providers. As a result, ARPKI prevents attacks even when $n-1$ service providers are colluding together, whereas in AKI, an adversary who successfully compromises two out of three designated service providers can successfully launch attacks [30]. In addition, the security property of ARPKI is proved by using a protocol verification tool called Tamarin prover [31]. The weakness of ARPKI is that all n designated service providers have to be involved in all the processes (i.e. certificate registration, confirmation, and update), which would cause considerable extra latencies and the delay of client connections.

In public log-based systems, efforts have been made to integrate *revocation management* with the certificate auditing. CT introduced revocation transparency (RT) [32] to deal with certificate revocation management; and in AKI and ARPKI, the public log only stores currently valid certificates (revoked certificates are purged from the log). However, the revocation checking processes in both RT and A(RP)KI are linear in the number of issued certificates making it inefficient. CIRT 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 *oligopoly.* The present-day certificate authority model requires that the set of global 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 systems address this issue. In each of those solutions, the set of log maintainers (and where applicable, time-line 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 validator, 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 issue is the requirement of trusted parties. Currently, all existing proposals have to rely on some sort of trusted parties or at least assume that not all parties are colluding together. However, a strong adversary (e.g. a government agency) might be able to control all service providers (used by a given client) in a system.

A third foundational issue of a different nature is that of analysis and correctness. SK, CT, AKI and CIRT 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, attacks on AKI and CIRT have been identified in [30] and in the appendix of our technical report [33], respectively. The flaws may be easily fixed, but only once they have been identified. It is therefore imperative to verify this kind of complex protocol. ARPKI is the first formally verified log-based PKI system. However, they used several abstractions during modelling in Tamarin prover. For example, they represent the underlying log structure (a Merkle tree) as a list. However, in systems like CIRT and this papr with more complex data structures, it is important to have a formalised data structure and its properties to prove the security claim. The formalisation of complex data structures and their properties in the log-based PKI systems is a remaining problem.

The last 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 web certificates, called *Distributed Transparent Key Infrastructure* (DTKI), with the following contributions.

- We identify *anti-oligopoly* 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 oligopoly, prevents attacks that use fake certificates, provides a way to manage certificate revocation, does not rely on any trusted party, and is secure even if all service providers (e.g. CAs and log maintainers) collude together (see Section 5 for our security statement). A comparison of the properties of different log-based systems can be found in Section 6.
- We provide formal machine-checked verification of its core security property using the Tamarin prover. In addition, 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 *oligopoly*, 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 oligopoly, 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. Clients of MLM are not required to trust the MLM, because they can efficiently verify the obtained associations.

Mirrors: Mirrors are servers that maintain a full copy of the mapping log and certificate logs respectively downloaded from the MLM and corresponding CLMs, and the corresponding digest of the log signed by the log maintainer. In other words, mirrors are distributed copies of logs. Anyone (e.g. ISPs, CLMs, CAs, domain owners) can be a mirror. Unlike in SK, mirrors are not required to be trusted in DTKI, because they give a proof for every association that they send to their clients. The proof is associated to the digest of the MLM.

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 (proof of presence in the relevant CLM is also needed).

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". Similar to the "sovereign key" in SK [24], the master signing key is only used to validate a TLS certificate (of the same subject) by issuing a signature on 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. Hence, the TLS secret key is the one for daily use; and the master signing key is rarely used. It will only be used for validating a new certificate, or revoke an existing certificate. We assume that domain owners can take care of their master signing key.

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 such fake certificates 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 and the relations between logs, DTKI uses a crowdsourcing-like way to ensure the integrity of the log and the relations between mapping log and a certificate log, and between certificate logs. In particular, the monitoring 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 (e.g. once a day). 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 CIRT) – we assume that gossip protocols [34] 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 time-stamp their digests, a log maintainer that issues inconsistent digests can be held accountable.

3. THE PUBLIC LOG

DTKI uses append-only logs to record all requests processed by the log maintainer, and allows log maintainers to efficiently generate some proofs that can be efficiently verified. These proofs mainly include that some data (e.g. a certificate or a revocation request) has or has not 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. Public log data structures have been widely studied [35, 36, 37, 38, 24, 27, 29]. To the best of our knowledge, no single data structure can provide all proofs required by DTKI. We adopt and extend the idea of CIRT log structure [29] which makes use of two data structures to provide all the kinds of proofs needed for DTKI.

This section presents the intuition of 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 by extending the CIRT data structure. The formalisation of our abstract data structures, log structures, and their properties, and our detailed implementation, are presented in our technical report [33]. We also present some examples of data structures there.

3.1. Data structures

Our log makes use of two data structures, namely chronological data structure and ordered data structure, to provide all proofs required by DTKI. We use the notion of *digest* to represent a unique 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.

[ht]

Function	Output
Chronological Data Structure	
digest	given input a sequence S of data, it outputs the digest of sequence S of data organised by using chronological data structure
$\mathsf{VerifPoP}_c$	given input $(\mathsf{digest}(S), d, p)$, it outputs a boolean value indicating the verification result of the proof p that some data d is included in a set S
$\mathsf{VerifPoE}_c$	given input $((dg', N'), (dg, N), p)$, it outputs a boolean value indicating the verification result of the proof p 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'
Ordered Data Structure	
$\mathsf{digest}_\mathsf{O}$	given input a sequence S of data, it outputs the digest of sequence S of data organised by using ordered data structure
$\mathsf{VerifPoP}_O$ (resp. $\mathsf{VerifPoAbs}_O$)	given input $(\mathsf{digest}_o(S), d, p)$, it outputs a boolean value indicating the verification result of the proof p that some data d is (resp. is not) included in a set S
$\mathsf{VerifPoAdd}_O$ (resp. $\mathsf{VerifPoD}_O$)	given input (d, dg, dg', p), it outputs a boolean value indicating the verification result of the proof p that dg' is the digest obtained after adding data d into (resp. deleting data d from) the sequence of data represented by digest dg
$\mathsf{VerifPoM}_O$	given input (d, d', dg, dg', p), it outputs a boolean value indicating the verification result of the proof p that dg' is the digest obtained after replacing d with d' in the sequence of data represented by dg

TABLE 1: Some functions supported by the data structures, of size N. The full list of operations and functions supported by the data structures, and the detailed properties of the data structures, are formalised in our technical report.

A chronological data structure is an append-only data structure, i.e. only the operation of adding some data is allowed. With a chronological data structure, for a given sequence S of data of size N and with digest dg, we have $d \in S$ for some data d, if and only if there exists a proof p of size $O(\log(N))$, called the proof of presence of d in S, such that p can be efficiently verified by using $\mathsf{VerifPoP}_c$ (see Table 1); and for all sequence S' with digest dg' and size $N' < N$, we have that S' is a prefix of S, if and only if there exists a proof p' of size $O(\log(N))$, called the proof of extension of S from S', such that p' can be efficiently verified by using $\mathsf{VerifPoE}_c$ (see Table 1).

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. Possible implementations are append-only Merkle tree [35] and append-only skip list, as proposed in [27] and [37], respectively.

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. This is useful for our public log since it enables users to verify the history of a log maintainer's behaviours.

Unfortunately, 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 need to use the *ordered data structure*.

An ordered data structure is a data structure allowing one to insert, delete, and modify stored data. In addition, with an ordered data structure, for a given sequence S of data of size N and with digest dg, we have $d \in S$ (resp. $d \notin S$) for some data d, if and only if there exists a proof p of size $O(\log(N))$, called the proof of presence (resp. absence) of d in (resp. not in) S, such that p can be efficiently verified by using $\mathsf{VerifPoP}_O$ (resp. $\mathsf{VerifPoAbs}_O$) (see Table 1).

Possible implementations of ordered data structure are Merkle tree which is organised as a binary search tree (as proposed in [29]), and authenticated dictionaries [36].

With ordered data structure, however, the size of proof that the current version of the data is extended from a previous version is $O(N)$. As the chronological data structure and the ordered data structure have complementary properties, we use the combination of them to organise our log.

3.2. Mapping log

To minimise oligopoly, 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*. In our technical report [33], we have formally defined some constraints on the regular expressions we use, the relations between them, and how to use random verification to verify that no overlap between regular expressions exists.

Intuitively, as presented in Figure 1, the mapping log is organised by using a chronological data structure, and stores received requests[3] together with the request time, and four digests of different ordered data structures representing the status of the log. Each entry is of the form

$$\mathsf{h}(\mathit{req}, t, dg^s, dg^{bl}, dg^r, dg^i)$$

In the formula, *req* is the request received by the mapping log at time t; dg^s[4] stores information about CLMs (e.g. the certificate of the CLM, and the current digest of the certificate log *clog*); dg^{bl} stores the identity of blacklisted certificate log maintainers; dg^r stores the mapping from a regular expression to the identity of CLMs, and dg^i stores the mapping from the identity of CLMs to a set of regular expressions.

In more detail, each entry of the mapping log contains digests after processing the request *req* (received by the mapping log maintainer at time t) on the digest stored in the previous record. Each of the notations is explained as follows:

- *req* can be $\mathsf{add}(rgx, id)$, $\mathsf{del}(rgx, id)$, $\mathsf{new}(cert)$, $\mathsf{mod}(cert, \mathsf{sign}_{sk}(cert'), \mathsf{sign}_{sk'}(n, dg, t))$, $\mathsf{bl}(id)$, and end, respectively corresponding to a request to add a mapping (rgx, id) of regular expression rgx and identity id of a *clog*, to delete a mapping (rgx, id), to add a certificate *cert* of a new *clog*, to change the certificate of a *clog* from *cert* to $cert'$, to blacklist id of an existing *clog*, and to close the update request; where sk and sk' are signing keys associated to the certificate *cert* and $cert'$, respectively; *cert* and $cert'$ share the same subject, and n and dg are the size and the digest of the corresponding *clog* at time t, respectively;
- 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 *clog*, data are ordered by rgx;
- 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 *clog* to a digest dg^{irgx} of ordered data structure storing a set of regular expressions, data are ordered by id.

[3]The request includes adding, removing, and modifying a certificate log and/or a mapping.

[4]We simplified the description here: we should say the ordered data structure represented by dg^s stores the information, rather than the digest dg^s stores it. We will use this simplification through the paper.

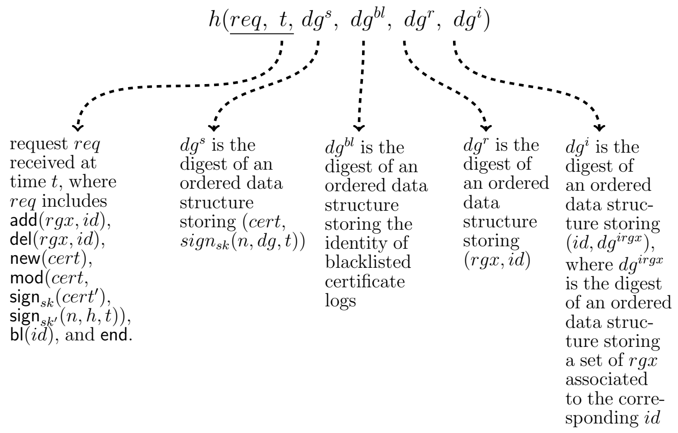

h(req, t, dg^s, dg^bl, dg^r, dg^i)
request req received at time t, where req includes add(rgx, id), del(rgx, id), new(cert), mod(cert, sign_sk(cert′), sign_sk′(n, h, t)), bl(id), and end.
dg^s is the digest of an ordered data structure storing (cert, sign_sk(n, dg, t))
dg^bl is the digest of an ordered 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.

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

The mapping log determines which certificate log is used for a domain. The certificates for the domain are stored in that certificate log.

A certificate log mainly stores certificates for domains according to the mappings presented in the mapping log. In particular, a certificate log is also organised by using a chronological data structure, and each entry of the log is of the form

$$\mathsf{h}(\mathit{req}, N, dg^{rgx})$$

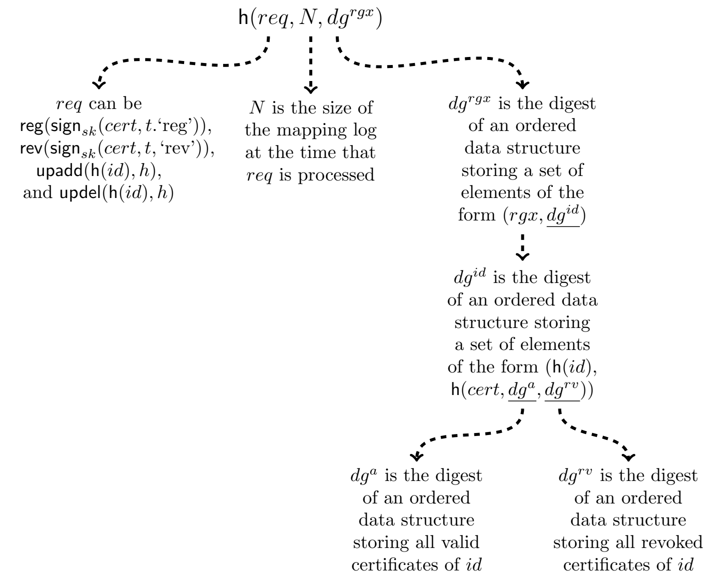
where *req* is the received request and is processed at the time such that the mapping log is of size N; dg^{rgx} represents an ordered data structure storing a set of mappings from regular expressions to the information associated to the corresponding domains, such that the domain name is an instance of the regular expression.

The stored information of a domain includes the identity and the master certificate of the domain, and two digests dg^a and dg^{rv} each presents an ordered data structure storing a set of active TLS certificates and a set of expired or revoked TLS certificates, respectively.

Elements in a record (as shown in 2) of a certificate log are detailed as follows.

- req can be $\mathsf{reg}(\mathsf{sign}_{sk}(cert, t, \text{'reg'}))$, $\mathsf{rev}(\mathsf{sign}_{sk}(cert, t, \text{'rev'}))$, $\mathsf{upadd}(\mathsf{h}(id), h)$, 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 $(cert, t, \text{'reg'})$ or 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. 'reg' and 'rev' are constant, and h is some value and we will explain it later.
- 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}), represents the status of the certificate log after processing the request req, and stores all the regular expressions rgx that the certificate log is associated to. dg^{id} is the digest of an ordered data structure storing a set of elements of the form $(\mathsf{h}(id), \mathsf{h}(cert, dg^a, dg^{rv}))$. It represents all domains associated to rgx. id is an instance of rgx and is the subject of master certificate $cert$. dg^a and dg^{rv} are digests of two ordered data structures each of which respectively stores a set of active and revoked 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.

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.



$\mathsf{h}(req, N, dg^{rgx})$
req can be $\mathsf{reg}(\mathsf{sign}_{sk}(cert, t.\text{'reg'}))$, $\mathsf{rev}(\mathsf{sign}_{sk}(cert, t, \text{'rev'}))$, $\mathsf{upadd}(\mathsf{h}(id), h)$, 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 certificate 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 time-stamp*, 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 time-stamp* 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 time-stamp*.

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 mis-behaving 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} (as presented in 3.2).

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 insertion and revocation

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

Upon receiving the request, the mirror locates the certificate of the authorised CLM, and generates the proofs that

a) the CLM is authorised for the domain; and

b) the certificate is the current valid for the CLM.

Loosely speaking, proof a) is the proof that the mapping from regular expression rgx to identity id is present in the digest dg^r (as presented in the mapping log structure), such that *example.com* is an instance of rgx, and id is the identity of the CLM; proof b) is the proof that the certificate with subject id is present in dg^s; additionally, a proof that both dg^s and dg^r are present in the latest record of the mapping log is needed. All proofs should be lined to the latest digest signed by the MLM. If Bob has previously observed a version of the mlog, then a proof that the current mlog is an extension of the version that Bob observed will also be provided.

Bob accepts the response if all proofs are valid. He then stores the verified data in his cache for future connection until the signed digest is expired.

In more detail, after receives a request from Bob, 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 time-stamp* containing the time t_{mlog}, and digest dg_{mlog} and size N_{mlog} of the mapping log.

Bob first verifies the received *signed Mlog time-stamp* 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 time-stamp* containing $(t'_{Mlog}, dg'_{mlog}, N'_{mlog})$ that he stored during a previous connection, and expects to receive a proof of extension of (dg'_{mlog}, N'_{mlog}) into (dg_{mlog}, N_{mlog}). If the received proof of extension is valid, then Bob stores the current *signed Mlog time-stamp*, 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. Insert and revoke certificates

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.

To publish the master certificate, 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 *AddReq* 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 that the master certificate for Bob is included in the log, and sends the signed proof and the updated digest of the log back to Bob. If the signature and the proof are valid, and the size of the log is no smaller than what the mirror says, then Bob accepts and stores the response as an evidence of successfully certificate publication. If Bob has previously observed a version of the clog, then a proof that the current clog is an extension of the version that Bob observed is also required.

Figure 3 presents the detailed process to publish the master certificate $cert_m$. After receives and verifies the request from Bob, 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 $\mathsf{h}(\mathsf{reg}(sign_{sk_m}(cert_m, t, \text{‘reg’})), N_{mlog}, dg^{rgx})$ 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; $\mathsf{reg}(sign_{sk_m}(cert_m, t, \text{‘reg’}))$ 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 time-stamp* 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}.

In addition, if Bob has run the request mapping protocol more than once, and has obtained a digest that is different from his local copy of the corresponding certificate log, then he should ask the CLM to prove that one of the digests is an extension of the other.

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.

To revoke a (master or TLS) certificate, the domain owner can perform a process similar to the process of adding a new certificate. For a revocation request with $\mathsf{sign}_{sk_m}(cert, t)$, the log maintainer needs to check that $\mathsf{sign}_{sk_m}(cert, t')$ is already in the log and $t > t'$. This ensures that the same master key is used for the revocation.

4.2. Certificate verification

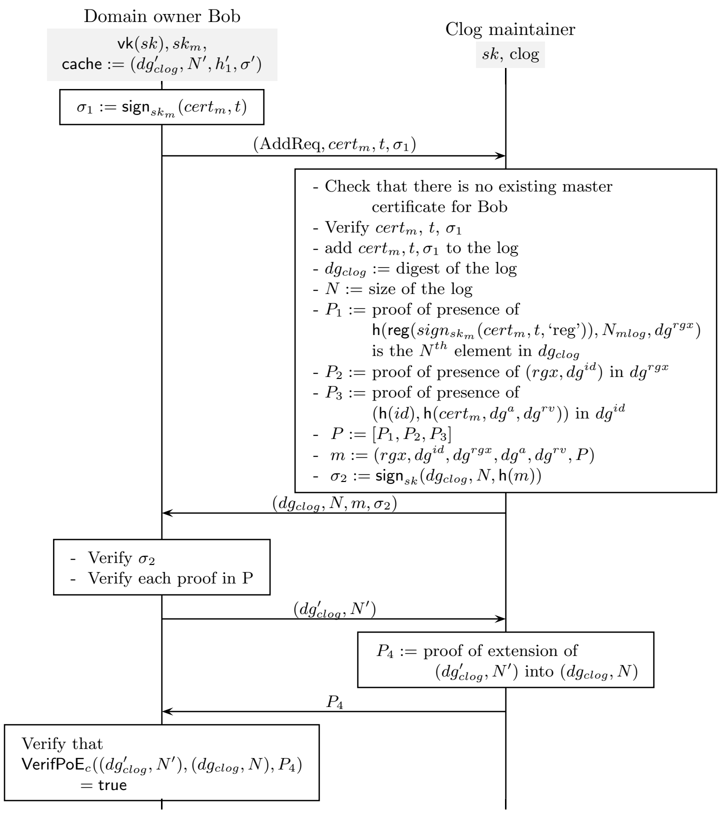

Domain owner Bob
$\mathsf{vk}(sk), sk_m$,
$\mathsf{cache} := (dg'_{clog}, N', h'_1, \sigma')$
Clog maintainer
sk, clog
$\sigma_1 := \mathsf{sign}_{sk_m}(cert_m, t)$
$(\text{AddReq}, cert_m, t, \sigma_1)$
- Check that there is no existing master certificate for Bob
- Verify $cert_m$, t, σ_1
- add $cert_m, t, \sigma_1$ to the log
- dg_{clog} := digest of the log
- N := size of the log
- P_1 := proof of presence of $\mathsf{h}(\mathsf{reg}(sign_{sk_m}(cert_m, t, \text{‘reg’})), N_{mlog}, dg^{rgx})$ is the N^{th} element in dg_{clog}
- P_2 := proof of presence of (rgx, dg^{id}) in dg^{rgx}
- P_3 := proof of presence of $(\mathsf{h}(id), \mathsf{h}(cert_m, dg^a, dg^{rv}))$ in dg^{id}
- $P := [P_1, P_2, P_3]$
- $m := (rgx, dg^{id}, dg^{rgx}, dg^a, dg^{rv}, P)$
- $\sigma_2 := \mathsf{sign}_{sk}(dg_{clog}, N, \mathsf{h}(m))$
$(dg_{clog}, N, m, \sigma_2)$
- Verify σ_2
- Verify each proof in P
(dg'_{clog}, N')
P_4 := proof of extension of (dg'_{clog}, N') into (dg_{clog}, N)
P_4
Verify that $\mathsf{VerifPoE}_c((dg'_{clog}, N'), (dg_{clog}, N), P_4)$ = true

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

When Alice wants to securely communicate with *example.com*, she sends the connection request to Bob, and 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 (mirror of) authorised certificate log maintainer to verify the received certificate.

Note that this verification requests extra communication round trips, but it gives a higher security guarantee. An alternative way is that Bob provides both certificates and proofs, and Alice verifies the received proofs directly.

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.

The Fig. 4 presents the detailed process of verifying a certificate. After Alice learns the identity of the authorised certificate log, she sends the verification request $VerifReq$ 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 $\mathsf{h}(req, N_{mlog}, dg^{rgx})$ 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 $h = \mathsf{h}(m)$ such that $m = (dg^a, dg^{rv}, rgx, dg^{id}, req, N_{mlog}, dg^{rgx}, P)$, and P is the set of proofs, and sends (dg_{clog}, N, σ) to Alice.

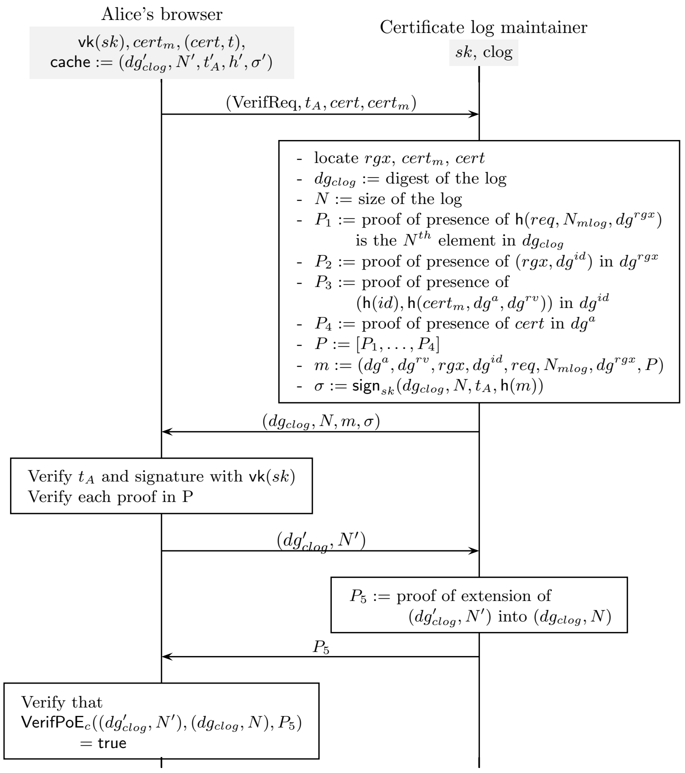
Alice should verify that N_{mlog} is the same as her local copy of the size of mapping log. If the received N_{mlog} is greater than the copy, then it means that the mapping log is changed (it rarely happens) and Alice should run the request mapping protocol again. If N_{mlog} is smaller, then it means the CLM is misbehaved. Alice then verifies the signature and proofs, and 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.

4.3. Log verification

Users of the system need to verify that the mapping log maintainer and certificate log maintainers did update their log correctly according to the requests they have received, and certificate log maintainers did follow the



Alice's browser
$\mathsf{vk}(sk), cert_m, (cert, t)$,
$\mathsf{cache} := (dg'_{clog}, N', t'_A, h', \sigma')$
Certificate log maintainer
sk, clog
$(\text{VerifReq}, t_A, cert, cert_m)$
- locate rgx, $cert_m$, $cert$
- dg_{clog} := digest of the log
- N := size of the log
- P_1 := proof of presence of $\mathsf{h}(req, N_{mlog}, dg^{rgx})$ is the N^{th} element in dg_{clog}
- P_2 := proof of presence of (rgx, dg^{id}) in dg^{rgx}
- P_3 := proof of presence of $(\mathsf{h}(id), \mathsf{h}(cert_m, dg^a, dg^{rv}))$ in dg^{id}
- P_4 := proof of presence of $cert$ in dg^a
- $P := [P_1, \ldots, P_4]$
- $m := (dg^a, dg^{rv}, rgx, dg^{id}, req, N_{mlog}, dg^{rgx}, P)$
- $\sigma := \mathsf{sign}_{sk}(dg_{clog}, N, t_A, \mathsf{h}(m))$
$(dg_{clog}, N, m, \sigma)$
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.

latest mappings specified in the mapping log.

These checks can be easily done by a trusted monitor. However, since we aim to provide a TTP-free system, DTKI uses a crowdsourcing-like method, based on random checking, to any mlog 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 an example. Example 1 presents the random checking process to verify the correct behaviour of the mapping log.

Example 1. Suppose verifier has randomly selected the k^{th} record of the mapping log, and the record has the form $\mathsf{h}(\mathsf{add}(rgx, id), t_k, dg^s_k, dg^{bl}_k, dg^r_k, dg^i_k)$. The verifier must check 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 $dg_k^s = dg_{k-1}^s$ and $dg_k^{bl} = dg_{k-1}^{bl}$; and
- verify that dg_k^r is the result of adding (rgx, id) into dg_{k-1}^r by using $\mathsf{VerifPoAdd}_O$, and id is an instance of rgx; and
- verify that (id, dg_k^{irgx}) is the result of replacing (id, dg_{k-1}^{irgx}) in dg_{k-1}^i by (id, dg_k^{irgx}) by using $\mathsf{VerifPoM}_O$; and
- verify that dg_k^{irgx} is the result of adding rgx into dg_{k-1}^{irgx} 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.

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 [39], 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[5] measure that the size of data for establishing an HTTPS protocol to log-in 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 data 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.

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

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[6] is $\approx$ 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 0.9% 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 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.

We provide all source codes and files required to understand and reproduce our security analysis at [40]. In particular, these include the complete DTKI models and the verified proofs.

5.1. Formal analysis

We analyse the main security properties of the DTKI protocol using the TAMARIN prover [31]. The TAMARIN prover is a symbolic analysis tool that can prove properties of security protocols for an unbounded number of instances and supports reasoning about

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

protocols with mutable global state, which makes it suitable for our log-based protocol. Protocols are specified using multiset rewriting rules, and properties are expressed in a guarded fragment of first order logic that allows quantification over timepoints.

Tamarin is capable of automatic verification in many cases, and it also supports interactive verification by manual traversal of the proof tree. If the tool terminates without finding a proof, it returns a counter-example. Counter-examples are given as so-called dependency graphs, which are partially ordered sets of rule instances that represent a set of executions that violate the property. Counter-examples can be used to refine the model, and give feedback to the implementer and designer.

Modeling aspects We used several abstractions during modeling. We model our log as lists, similar to the abstraction used in [30, 41]. We also assume that the random checking is verified.

We model the protocol roles D (domain server), M (mapping log maintainer), C (certificate log maintainer), and CA (certificate authority) by a set of rewrite rules. Each rewrite rule typically models receiving a message, taking an appropriate action, and sending a response message. Our modeling approach is similar to the one used in most Tamarin models. Our modeling of the roles directly corresponds to the protocol descriptions in the previous sections. Tamarin provides built-in support for a Dolev-Yao style network attacker, i.e., one who is in full control of the network. We additionally specify rules that enable the attacker to compromise service providers, namely teh mapping log maintainer, certificate log maintainers and CAs, learn their secrets, and modify public logs.

The final DTKI model consists of 959 lines for the base model and five main property specifications, examples of which we will give below.

Proof goals We state several proof goals for our model, exactly as specified in Tamarin's syntax. Since Tamarin's property specification language is a fragment of first-order logic, it contains logical connectives (`|`, `&`, `==>`, `not`, ...) and quantifiers (`All`, `Ex`). In Tamarin, proof goals are marked as `lemma`. The `#`-prefix is used to denote timepoints, and "`E @ #i`" expresses that the event E occurs at timepoint i.

The first goal is a check for executability that ensures that our model allows for the successful transmission of a message. It is encoded in the following way.

```
lemma protocol_correctness:
 exists-trace
 " /* It is possible that */
   Ex D Did n rgx ltpkD stpkD #i1.

   /* The user received a confirmation of receiving the message
   i.e. hashed secret the user has sent to the domain */

   Com_Done(D, Did, n, rgx, ltpkD, stpkD) @ #i1

     /* without the adversary compromising any party. */

   &   not (Ex #i2 CA ltkCA.
             Compromise_CA(CA,ltkCA) @ #i2)

   &   not (Ex #i3 C ltkC.
             Compromise_CLM(C,ltkC) @ #i3)

   &   not (Ex #i4 M ltkM.
             Compromise_MLM(M,ltkM) @ #i4)

"
```

The property holds if the Tamarin model exhibits a behaviour in which a domain server received a message without the attacker compromising any service providers. This property mainly serves as a sanity check on the model. If it did not hold, it would mean our model does not model the normal (honest) message flow, which could indicate a flaw in the model. Tamarin automatically proves this property in several minutes and generates the expected trace in the form of a graphical representation of the rule instantiations and the message flow.

We additionally proved several other sanity-checking properties to minimize the risk of modeling errors.

The second example goal is a secrecy property with respect to a classical attacker, and expresses that when no service provider is compromised, the attacker cannot learn the message exchanged between a user and a domain server. Note that `K(m)` is a special event that denotes that the attacker knows m at this time.

```
lemma message_secrecy_no_compromised_party:
 "
 All D Did m rgx ltpkD stpkD #i1.

   /* The user received a confirmation of receiving the message
   i.e. hashed secret the user has sent to the domain */

   (Com_Done(D, Did, m, rgx, ltpkD, stpkD) @ #i1

/* and no party has been compromised */
   &   not (Ex #i2 CA ltkCA.
             Compromise_CA(CA,ltkCA) @ #i2)

   &   not (Ex #i3 C ltkC.
             Compromise_CLM(C,ltkC) @ #i3)

   &   not (Ex #i4 M ltkM.
             Compromise_MLM(M,ltkM) @ #i4)
     )
     ==>
     ( /* then the adversary cannot know m */
      not (Ex #i5. K(m) @ #i5)
     )
 "
```

Tamarin proves this property automatically (in 575 steps).

The above result implies that if a domain server D, whose domain name is Did such that Did is an instence of regular expression rgx, receives a message that was sent by a user, and the attacker did not compromise server providers, then the attacker will not learn the message.

The next two properties encode the unique security guarantees provided by our protocol, in the case that even all service providers are compromised.

The first main property we prove is that when all service providers (i.e. CAs, the MLM, and CLMs) are compromised, and the domain owner has successfully verified his master certificate in the log, then the attacker cannot learn the message exchanged between a user and a domain owner. It is proven automatically by Tamarin in 5369 steps.

```
lemma message_secrecy_compromise_all_domain_verified_master_cert:
 "
 All D Did m rgx ltpkD stpkD #i1.

   /* The user received a confirmation of receiving the message
   i.e. hashed secret the user has sent to the domain */

   (Com_Done(D, Did, m, rgx, ltpkD, stpkD) @ #i1

   /* and at an earlier time, the domain server has verified his
      master certificate */
     & Ex #i2.
     VerifiedMasterCert(D, Did, rgx, ltpkD) @ #i2
     & #i2 < #i1

/* and all parties can be compromised*/

      )
      ==>
      ( /* then the adversary cannot know m */
       not (Ex #i3. K(m) @ #i3)
      )
 "
```

The property states that if a domain server D receives a message that was sent by a user, and at an earlier time, the domain server has verified his master certificate, then even if the attacker can compromise all server providers, the attacker cannot learn the message.

The final property states that when all service providers can be compromised, and a domain owner has not verified his/her master certificate, and the attacker learns the message exchanged between a user and the domain owner, then afterwards the domain owner can detect this attack by checking the log. It is also verified by Tamarin within a few minutes.

```
lemma detect_bad_records_in_the_log_when_master_cert_not_verified:

 "
 All D Did m rgx ltpkD flag stpkD #i1 #i2 #i3.

   /* The user received a confirmation of receiving the message
   i.e. hashed secret the user has sent to the domain */

   (Com_Done(D, Did, m, rgx, ltpkD, stpkD) @ #i1

   /* and all parties can be compromised*/

   /* and the master certificate of the domain was not
      initially verified */

   /* and the adversary knows m */

     &  K(m) @ #i2

   /* and we afterwards check the log */
     &  CheckedLog(D, Did, rgx, ltpkD, flag, stpkD) @ #i3
     &   #i1 < #i3)
      ==>
     ( /* then we can detect a fake record in the log */
        (flag = 'bad')
     )
 "
```

6. COMPARISON

As mentioned previously, DTKI builds upon a wealth of ideas from SK [24], CT [27], CIRT [29], and AKI [28]. Figure 5 shows the dimensions along which DTKI aims to improve on those systems.

Compared with CT, DTKI supports revocation by enabling log providers to offer proofs of *absence* and *currency* of certificates. In CT, there is no mechanism for revocation. CT has proposed additional data structures to hold revoked certificates, and those data structures support proofs of their contents [42]. However, there is no mechanism to ensure that the data structures are maintained correctly in time.

Compared to CIRT, DTKI extends the log structure of CIRT to make it suitable for multiple log maintainers, and provides a stronger security guarantee as it prevents attacks rather than merely detecting them. In addition, the presence of the mapping log maintainer and multiple certificate log maintainers create some extra monitoring work. DTKI solves it by using a detailed crowd-sourcing verification system to distribute the monitoring work to all users' browsers.

Compared to AKI and ARPKI, in DTKI the log providers can give proof that the log is maintained append-only from one step to the next. The data structure in A(RP)KI does not allow this, and therefore they cannot give a verifiable guarantee to the clients that no data is removed from the log.

DTKI improves the support that CT and A(RP)KI have for multiple log providers. In CT and AKI, domain owners wishing to check if there exists a log provider that has registered a certificate for him has to check all the log providers, and therefore the full set of log providers has to be fixed and well-known. This prevents new log providers being flexibly created, creating an oligopoly. In contrast, DTKI requires the browsers only to have the MLM public key built-in, minimising the oligopoly element.

In DTKI, no trusted party is required, as it uses crowd-sourcing verification to eliminate the need of trusted parties, i.e. a trusted party's verification work can be done probabilistically in small pieces, meaning that users' browsers can collectively perform the monitoring role.

Unlike the mentioned previous work, DTKI allows the possibility that all service providers (i.e. the MLM, CLMs, and mirrors) to collude together, and can still prevent attacks. In contrast, SK and CT can only detect attacks, and to prevent attacks, A(RP)KI requires that not all service providers collude together. Similar to A(RP)KI, DTKI also assumes that the domain is initially registered by an honest party to prevent attacks, otherwise A(RP)KI and DTKI can only detect attacks.

7. 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

	SK [24]	CT [27]	AKI [28]	ARPKI [30]	DTKI
Terminology					
Log provider	Time-line server	Log	Integrity log server (ILS)	Integrity log server (ILS)	Certificate/Mapping log maintainer (CLM, MLM)
Log extension	-	Log consistency	-	-	Log extension
Trusted party	Mirror	Auditor & monitor	Validator	Validator (optional)	-
Whether answers to queries rely on trusted parties or are accompanied by a proof					
Certificate-in-log query:	Rely	Proof	Proof	Proof	Proof
Certificate-current-in-log query:	Rely	Rely	Proof	Proof	Proof
Subject-absent-from-log query:	Rely	Rely	Proof	Proof	Proof
Log extension query:	Rely	Proof	Rely	Rely	Proof
Non-necessity of trusted party					
Trusted party role can be distributed randomly to browsers	No	No	No$^+$	No$^+$	Yes
Trust assumptions					
Not all service providers collude together	Yes	Yes	Yes	Yes	No
Domain is initially registered by an honest party	No	No	Yes*	Yes*	Yes*
Security guarantee					
Attacks detection or prevention	Detection	Detection	Prevention	Prevention	Prevention
Oligopoly issues					
Log providers required to be built into browser (oligopoly)	Yes	Yes	Yes	Yes	Only MLM
Monitors required to be built into browser (oligopoly and trust non-agility)	Yes	No	Yes	Yes†	No

+ The system limits the trust in each server by letting them to monitor each other's behaviour.

* Without the assumption, the security guarantee is detection rather than prevention.

† The trusted party is optional, if there is a trusted party, then the trusted party is required to be built into browser.

FIGURE 5: Comparison of log-based approaches to certificate management. **Terminology** helps compare the terminology used in the papers. **How queries rely on trusted parties** shows whether responses to browser queries come with proof of correctness or rely on the honesty of trusted parties. **Necessity of trusted parties** shows whether the TP role can be performed by browsers. **Trust assumptions** shows the assumption for the claimed security guarantee. **Oligopoly issues** shows the entities that browsers need to know about.

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 [39] (though not every domain has a certificate), and 3×10^9 internet users [43]. Thus, if every user makes one random check per day, then everything will on average, be checked 10 times per day.

Master certificate concerns One concern is that a CA might publish fake master certificates for domains that the CA doesn't own and are not yet registered. However, this problem is not likely to occur: CAs are businesses, they cannot afford the bad press from negative public opinion and they cannot afford the loss of reputation. Hence, they will only want to launch attacks that would not be caught. (Such an adversary model has been described by Franklin and Yung [44], Canetti and Ostrovsky [45], Hazay and Lindell [46], and Ryan [29]). In DTKI, if a CA attempts to publish a fake master certificate for some domain, it will have to leave evidence of its misbehaviour in the log, and the misbehaviour will eventually be detected by the genuine domain owner.

Avoidance of oligopoly As we mentioned in the introduction, the predecessors (SK, CT, CIRT, AKI, ARPKI) of DTKI do not solve a foundational issue, namely *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 mis-issued 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. In DTKI, the public log of the MLM is the only thing that browsers need to know.

MLM may be thought to represent a monopoly; to the extent that it does, it is a much weaker monopoly than the oligopoly of CAs or log maintainers. CAs and log maintainers offer commercial services and compete with each other, by offering different levels of service at different price points in different markets. MLM does not offer any commercial services; it performs a purely administrative role, and is not required to be trusted because it behaves fully transparently and does not manage any certificate for web domains.

8. CONCLUSIONS AND FUTURE WORK

Sovereign keys (SK), certificate transparency (CT), accountable key infrastructure (AKI), certificate issuance and revocation transparency (CIRT), and attack resilient PKI (ARPKI) 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 an oligopoly of log maintainers (as discussed in section 7), 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, A(RP)KI, and CIRT). Since devising new security protocols is notoriously error-prone, we provide a formalisation of DTKI, and formally proved its security properties by using Tamarin prover.

REFERENCES

[1] Eckersley, P. (2011). Iranian hackers obtain fraudulent HTTPS certificates: How close to a web security meltdown did we get? Electronic Frontier Foundation.

[2] Leyden, J. (2012). Trustwave admits crafting SSL snooping certificate: Allowing bosses to spy on staff was wrong, says security biz. The Register.

[3] MS01-017: Erroneous Verisign-issued digital certificates pose spoofing hazard. Microsoft Support.

[4] Roberts, P. (2011). Phony SSL certificates issued for Google, Yahoo, Skype, others. Threat Post.

[5] Sterling, T. (2011). Second firm warns of concern after Dutch hack. Yahoo! News.

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

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

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

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

[10] Niemann, A. and Brendel, J. (2013) A survey on CA compromises. , **?**

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

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

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

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

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

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

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

[18] The EFF SSL Observatory.

[19] Certificate patrol.

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

[21] Marlinspike, M. (2011) SSL and the future of authenticity. *Black Hat, USA*.

[22] Rivest, R. L. (1998) Can we eliminate certificate revocation lists? *Financial Cryptography*, pp. 178–183. Springer.

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

[24] Eckersley, P. (2012). Sovereign key cryptography for internet domains. Internet Draft.

[25] Dierks, T. and Rescorla, E. (2008). The transport layer security (TLS) protocol version 1.2. RFC 5246 (Proposed Standard). Updated by RFCs 5746, 5878, 6176.

[26] Turner, S. and Polk, T. (2011). Prohibiting secure sockets layer (SSL) version 2.0. RFC 6176 (Proposed Standard).

[27] Laurie, B., Langley, A., and Kasper, E. (2013). Certificate Transparency. RFC 6962 (Experimental).

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

[29] Ryan, M. (2014) Enhanced certificate transparency and end-to-end encrypted mail. *NDSS.*

[30] Basin, D., Cremers, C., Kim, T. H.-J., Perrig, A., Sasse, R., and Szalachowski, P. (2014) ARPKI: Attack resilient public-key infrastructure. *ACM Conference on Computer and Communications Security.*

[31] Meier, S., Schmidt, B., Cremers, C., and Basin, D. (2013) The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In Sharygina, N. and Veith, H. (eds.), *Computer Aided Verification, 25th International Conference, CAV 2013, Princeton, USA, Proc.*, Lecture Notes in Computer Science, **8044**, pp. 696–701. Springer.

[32] Laurie, B. and Kasper, E. (2012) Revocation transparency. Google Research.

[33] Yu, J., Cheval, V., and Ryan, M. DTKI: a new formalised PKI with no trusted parties. Technical Report.

[34] Jelasity, M., Voulgaris, S., Guerraoui, R., Kermarrec, A.-M., and Van Steen, M. (2007) Gossip-based peer sampling. *ACM Transactions on Computer Systems (TOCS)*, **25**, 8.

[35] Merkle, R. C. (1987) A digital signature based on a conventional encryption function. *CRYPTO*, pp. 369–378.

[36] Anagnostopoulos, A., Goodrich, M. T., and Tamassia, R. (2001) Persistent authenticated dictionaries and their applications. *Information Security, 4th International Conference, ISC 2001, Malaga, Spain, October 1-3, 2001, Proceedings*, pp. 379–393.

[37] Maniatis, P. and Baker, M. (2003) Authenticated append-only skip lists. *CoRR*, **cs.CR/0302010**.

[38] Yumerefendi, A. R. and Chase, J. S. (2007) Strong accountability for network storage. *ACM Transactions on Storage*, **3**.

[39] (2014). Domain name industry brief. Verisign.

[40] Yu, J. (2015). Tamarin models for the DTKI protocol. `http://www.jiangshanyu.com/doc/paper/DTKI-proof.zip`.

[41] Yu, J., Ryan, M., and Cremers, C. (2015) How to detect unauthorised usage of a key. *IACR Cryptology ePrint Archive*, **2015**, 486.

[42] Laurie, B. and Kasper, E. (2012). Revocation transparency. Google Research.

[43] (2014). Internet users. Internet Usage Statistics.

[44] Franklin, M. K. and Yung, M. (1992) Communication complexity of secure computation (extended abstract). *STOC*, pp. 699–710.

[45] Canetti, R. and Ostrovsky, R. (1999) Secure computation with honest-looking parties: What if nobody is truly honest? (extended abstract). *STOC*, pp. 255–264.

[46] Hazay, C. and Lindell, Y. (2008) Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. *TCC*, pp. 155–175.

IMPLEMENTATION OF DATA STRUCTURES

This section shows the implementation 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

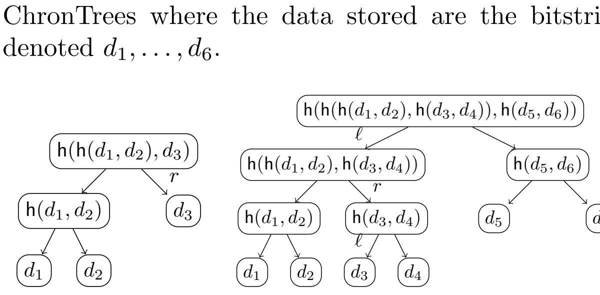
The chronological data structure is implemented based on Merkle tree structure that we call *ChronTree.*

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 labelled 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 .1 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 .1: 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

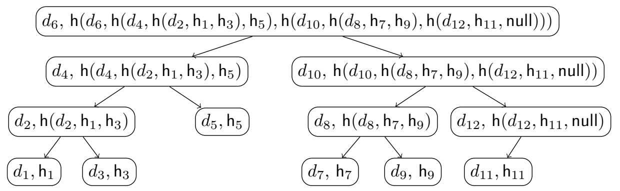

d_6, $\mathsf{h}(d_6, \mathsf{h}(d_4, \mathsf{h}(d_2, \mathsf{h}_1, \mathsf{h}_3), \mathsf{h}_5), \mathsf{h}(d_{10}, \mathsf{h}(d_8, \mathsf{h}_7, \mathsf{h}_9), \mathsf{h}(d_{12}, \mathsf{h}_{11}, \mathsf{null})))$
d_4, $\mathsf{h}(d_4, \mathsf{h}(d_2, \mathsf{h}_1, \mathsf{h}_3), \mathsf{h}_5)$
d_{10}, $\mathsf{h}(d_{10}, \mathsf{h}(d_8, \mathsf{h}_7, \mathsf{h}_9), \mathsf{h}(d_{12}, \mathsf{h}_{11}, \mathsf{null}))$
$d_2, \mathsf{h}(d_2, \mathsf{h}_1, \mathsf{h}_3)$
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 .2: An example of a LexTree T_c, where $\mathsf{h}_i = \mathsf{h}(d_i, \mathsf{null}, \mathsf{null})$ for all $i = \{1, 3, 5, 7, 9, 11\}$

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 labelled by d; and is defined as $(w, [b_1, \dots, b_k])$ 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 $n_1, \dots, n_{k+1}$ is the path from n_1 to the root, then for all $i \in \{1, \dots, k\}$, 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 2. Consider the ChronTree T_b of Figure .1. The proof of presence of d_3 in T_b is the tuple (w, seq) where:

- $w = \ell \cdot r \cdot \ell$
- $\mathit{seq} = [d_4, \mathsf{h}(d_1, d_2), \mathsf{h}(d_5, d_6)]$

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 $\mathcal{S}(T) = [d_1, \dots, d_{N'}, \dots, d_N]$, and $\mathcal{S}(T') = [d_1, \dots, d_{N'}]$ for some bitstrings $d_1, \dots, d_{N'}, \dots, d_N$. 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 labelled 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 $\mathit{seq} = \mathit{seq}'$; otherwise
- $\mathit{seq} = d :: \mathit{seq}'$, where :: is the concatenation operation.

Example 3. The proof of extensions of T_a into T_b (Figure .1) is the sequence $\mathit{seq} = [d_3, d_4, \mathsf{h}(d_1, d_2), \mathsf{h}(d_5, d_6)]$.

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 $\mathcal{S}(T') = [d_1, \dots, d_{N'}]$. 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 4. Coming back to Example 3, 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 .1 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 labelled 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 5. The tree in Figure .2 is an example of LexTree where $d_1 \leq d_2 \leq \ldots \leq d_{12}$.

Example 6. Consider the LexTree T of Figure .2. 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 7. Consider the T_c of Figure .2. 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:

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