Secure Logging Schemes and Certificate Transparency

Benjamin Dowling[1], Felix Günther[2], Udyani Herath[1], and Douglas Stebila[3]

[1] *School of Electrical Engineering and Computer Science*
Queensland University of Technology, Brisbane, Australia
{b1.dowling,udyani.herathmudiyanselage}@qut.edu.au

[2] *Cryptoplexity, Technische Universität Darmstadt, Germany*
guenther@cs.tu-darmstadt.de

[3] *Department of Computing and Software*
McMaster University, Hamilton, Ontario, Canada
stebilad@mcmaster.ca

Abstract

Since hundreds of certificate authorities (CAs) can issue browser-trusted certificates, it can be difficult for domain owners to detect certificates that have been fraudulently issued for their domain. *Certificate Transparency (CT)* is a recent standard by the Internet Engineering Task Force (IETF) that aims to construct public logs of all certificates issued by CAs, making it easier for domain owners to monitor for fraudulently issued certificates. To avoid relying on trusted log servers, CT includes mechanisms by which monitors and auditors can check whether logs are behaving honestly or not; these mechanisms are primarily based on Merkle tree hashing and authentication proofs. Given that CT is now being deployed, it is important to verify that it achieves its security goals.

In this work, we define four security properties of logging schemes such as CT that can be assured via *cryptographic means*, and show that CT does achieve these security properties. We consider two classes of security goals: those involving security against a malicious logger attempting to present different views of the log to different parties or at different points in time, and those involving security against malicious monitors who attempt to frame an honest log for failing to include a certificate in the log. We show that Certificate Transparency satisfies these security properties under various assumptions on Merkle trees all of which reduce to collision resistance of the underlying hash function (and in one case with the additional assumption of unforgeable signatures).

Contents

1 Introduction

The security of web communication via the Transport Layer Security (TLS) protocol relies on safe distribution of public keys in the form of X.509 certificates. *Certificate authorities (CAs)* are trusted third parties that endorse the public keys of *subjects* by performing checks and issuing certificates. Web browsers can accept certificates from hundreds of CAs, and relying parties are unable to determine whether certificates were issued at the request of the subject or fraudulently issued by the CAs, whether by mistake or due to compromise.

In recent years there have been high-profile cases of misissued certificates being used to spoof legitimate websites. For example, in 2011 an intruder managed to issue itself a valid certificate for the domain `google.com` and its subdomains from the prominent Dutch Certificate Authority DigiNotar [Fox12]. This certificate was issued in July 2011 and may have been used maliciously for weeks before the detection on August 28, 2011, of large-scale man-in-the-middle (MITM) attacks on multiple users in Iran. In another instance, the Comodo Group suffered from an attack which resulted in the issuance of nine fraudulent certificates for domains owned by Google, Yahoo!, Skype, and others [Com11].

Certificate Transparency (CT) [Lau14, LLK13] is an experimental protocol originally proposed by Google and standardized by the Internet Engineering Task Force (IETF) Public Notary Transparency working group to mitigate the threat of fraudulently issued certificates by publicly logging certificates. CT provides an open auditing and monitoring system which allows domain owners to verify that no fraudulent certificates have been issued for their domains. The end goal of Certificate Transparency is that web clients should only accept certificates that are publicly logged and that it should be impossible for a CA to issue a certificate for a domain without it being publicly visible. Recent incidents demonstrated the effectiveness of CT logs: Google employees detected unrequested certificates for two of their subdomains issued by a Symantec sub-CA Thawte [SE15]. The certificates were issued on September 14, 2015 and detected by September 17, 2015; the certificates were revoked immediately, limiting the exposure of the certificates to just three days. In another case, the Facebook security team discovered an issuance of two certificates on multiple subdomains violating Facebook's internal security policies [Hua16]. The incident was investigated and both certificates revoked within hours, even before they were deployed to production systems.

1.1 The Web PKI and Certificate Transparency

The basic web public key infrastructure (PKI) includes several types of entities which perform different tasks: web servers, certificate authorities, browser vendors and web browsers. The Certificate Transparency framework adds several new entities which help maintain and monitor public logs:

- *Loggers* or *log servers* maintain publicly accessible append-only logs of certificates. These certificates are received from submitters. As a new entry might not be published immediately for operational reasoning, the logger provides each submitter with a promise to log the certificate within a certain amount of time; the promise is called a *signed certificate timestamp (SCT)*.
- *Submitters*, submit certificates (or partially completed *pre-certificates*) to a log server and receive a signed certificate timestamp from the log.
- *Monitors* are public or private services that watch for misbehaving logs or suspicious certificates by periodically contacting and downloading information from log servers. They inspect every new entry in a log, keep copies of the entire log, and verify the consistency between published revisions of the log.
- *Auditors* verify the correct behaviour of a log, checking that certificates that a logger has promised to include are present in the log. Auditors may be standalone entities or integrated into monitors or web clients.

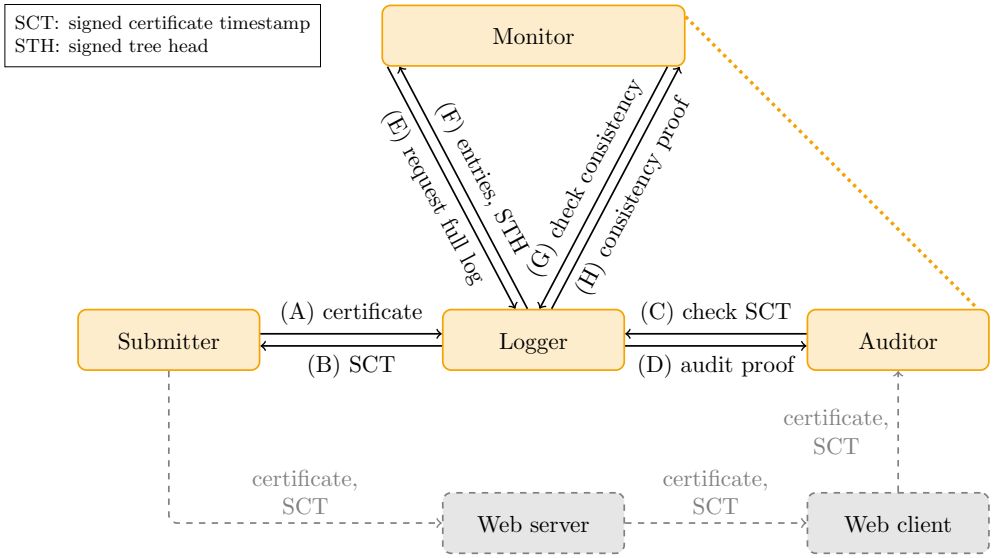


Figure 1: Overview over the interaction between entities in Certificate Transparency; see Section 1.1 for details. Solid-line interactions and solid-line, orange entities are captured by the model in our work while dashed-line interactions and dashed-line, gray entities are not captured. Dotted line–connected entities (monitors and auditors or auditors and web clients) might be the same physical entity.

In CT, the original entities from the web PKI also have some additional tasks:

- CAs should act as submitters above.
- Web servers should include their SCT along with the certificate when communicating with clients. Web servers may choose to submit their certificate to a log server if their CA does not do so for them.
- Web clients, upon receiving an SCT from a web server, may choose to verify that the log named in the SCT actually has publicly logged the certificate (thereby taking on the role of an auditor as above).
- Browser vendors may push updates that remove CAs or revoke certificates based on claims from monitors and web servers about misbehaving CAs.

Figure 1 provides an overview of the involved parties and their interactions in CT.[1] At the submission of a new certificate entry (step A), the logger returns a signed certificate timestamp (SCT) (step B), which is a promise to include the entry in the log. Every log has a published parameter called a *maximum merge delay (MMD)* which indicates the maximum period between issuing a timestamp and the inclusion of the certificate into the log.

In CT, the logger stores the entries of the log in an append-only Merkle hash tree [Mer79, Mer90], a form of a tamper-evident history tree [CW09, Cro09]. Recall for Merkle trees, data is placed at the leaves of a binary tree and each intermediate node is the hash of its two child nodes; the root of the trees acts as a fingerprint of all included data. In CT, the root of the tree is signed and published by the logger, and is called the *signed tree head (STH)*. The observed fingerprints are exchanged by all parties in the system through a so-called "gossiping" protocol [NGR15].

Gossiping allows monitors, auditors, and web clients to share information they receive from log servers, with the goal of collectively detecting misbehavior of log servers while limiting the damage to user privacy. The parties who hold the same fingerprints of a log are (cryptographically) assured that they have the same view of the log at the point in time represented by the fingerprint. Gossiping can take implemented through SCT feedback (where web clients send SCTs through

[1] Note that the labeling of interactions is simply for reference and does not indicate a particular order of the displayed requests.

HTTPS servers), STH pollination (where web clients and CT auditors/monitors use HTTPS servers as STH pools) and trusted auditor streams (where web clients directly communicate with trusted CT auditors/monitors).

To convince other parties that promised certificates are included in a log, and that subsequent published fingerprints are consistent, the logger employs two types of cryptographic proofs: audit proofs and consistency proofs.

An *audit proof* allows an auditor to verify that a particular certificate/SCT that a logger has promised to include is actually included in the log represented by a fingerprint, shown in steps C and D. In CT, an audit proof is essentially an authentication path in the Merkle tree from the leaf containing the certificate in question to the root hash/fingerprint contained in the signed tree hash.

A *consistency proof* allows an auditor or monitor to verify that the log is append-only, in particular that the log represented by a fingerprint at one point in time t_0 is a prefix of the log represented by a fingerprint at a later point in time $t_1 > t_0$, shown in steps G and H. In CT, a consistency proof is a subset of intermediate nodes in the Merkle tree needed to connect the two root hashes.

Monitors can also request that a logger provides them with the full set of entries represented by a fingerprint (steps E and F). In CT, this can be verified by recomputing the Merkle tree hash of the entries.

1.2 Threat Model

To achieve the goal of allowing domain owners to learn when fraudulent certificates are issued for their domain, one might initially expect that the log servers must be trusted and that CAs must submit all their certificates to the log servers. A large part of the CT framework aims to achieve this goal without relying on trusted log servers or honest CAs. Instead, trust is decentralized among the loggers, monitors, and auditors, who collectively watch each other. While this approach eliminates the requirement of a single trusted third party, one must now deal with the threat that one or more of these entities is malicious.

The informational IETF draft "Attack Model for Certificate Transparency" [Ken15] described potential attack scenarios when Certificate Transparency is used in the context of web public-key infrastructure. Here we associate the threats named in the draft with the various interacting parties in CT:

1. Misbehaving log server
   - **(L1)** Creating an entry for a fake certificate
   - **(L2)** Presenting different log entry views for entities
   - **(L3)** Not performing syntactic checks on certificate entries
   - **(L4)** Issuing SCTs for fake certificates
   - **(L5)** Reporting syntactic errors for a syntactically valid certificate
2. Misbehaving monitor
   - **(M1)** Not notifying the lack of SCT in the certificate received from CA
   - **(M2)** Not informing the targeted domain owner about fake certificates
   - **(M3)** Issuing false warnings to a targeted domain owner
   - **(M4)** Not reporting syntactic errors of certificates when noted
   - **(M5)** Reporting syntactic errors for a syntactically valid certificate
3. Malicious submitter/certificate authority
   - **(S1)** Failing to log a certificate
   - **(S2)** Issuing an erroneous certificate and causing the log to not perform checks
   - **(S3)** Refusing to revoke/delay revoking once mis-issuance is detected and reported
4. Web client

(C1) Not rejecting certificates that do not have SCTs[2]

In this paper, we will view CT as an instantiation of logging schemes in general and model the security properties that can be cryptographically assured in such logging schemes. We will not require log entries to be certificates nor assume a PKI. Thus, some of the threats above will not be considered. For example, some threats (L3, L5, M4, M5, and S2) relate to the validity or syntax of certificates, which we will exclude from our model of general-purpose logging schemes.

Threats L1, L2, and L4 concern a logger that misbehaves by logging a fake entry and try to cover up their misbehaviour by presenting different views to different entities. Of these threats, we capture L2 directly as a collection of cryptographic security notions which prevent loggers from splitting the view presented to different parties, removing certificates, or falsely claiming the inclusion of entries. Indirectly, this also protects against threats L1 and L4 as any fake certificate included or promised for inclusion in Certificate Transparency will become known to the monitoring parties.

Threat M3 involves a malicious monitor framing an honest log for bad behaviour. We capture this in our model by showing how CT cryptographically prevents a monitor from falsely framing a logger for not including a promised certificate.

Threats M1 and M2 concern the failure of a monitor in notifying affected parties of misbehaviour. While an important threat, this is best captured via contractual obligations between monitors and other entities, rather than cryptographic means. S3 is similar.

Threats S1 and C1 go hand-in-hand with each other and with the gossiping protocol. No theoretical measure can force a CA to log a certificate or force a client to reject something. However, if clients do reject certificates that lack an SCT, then submitters who violate S1 will find their certificates rejected by clients.

From this categorization of the threats against CT, we see that the primary threats that can be analyzed cryptographically are L2 and M3, and thus our model of general-purpose logging schemes will focus on these threats and the parties involved in them, specifically loggers and monitors/auditors. (We will generally refrain from separating monitors and auditors in our model, as they collectively work together via gossiping to verify log behaviour.)

1.3 Our Contribution

Given the practical significance of Certificate Transparency, it is important to have a formal understanding of the security goals of CT and analyse whether CT achieves those goals. The objective of our work is to define security goals of logging schemes using the formalism of provable security, and attempt to prove that CT satisfies these security goals under suitable cryptographic assumptions. Our model of logging schemes does not assume a PKI context, so we do not assume that log entries must have a particular syntax, and thus we leave the threats involving validity or syntax of log entries to existing analyses on certificate validity. Similarly, we omit consideration of threats where an entity *fails* to act.

As noted above, we will focus on two particular threats in the CT threat model: whether a misbehaving log server can present different views of the log and whether a misbehaving monitor can frame an honest log server for bad behaviour. Thus, our model will focus on two entities: the logger and the monitor/auditor.

Definition of logging schemes. In Section 3 we formally define logging schemes, naming operations that each entity can perform. This model does not attach any semantic meaning to the entries being logged; in particular, we do not assume that log entries are certificates. Subsequently, we describe the operations of Certificate Transparency as a specific instantiation of the logging scheme framework.

[2]The draft does not expect clients to be strict on rejection in this case, as CT might not (yet) be deployed for the certificate received.

Security definitions. Next, we introduce cryptographic security properties for logging schemes in Section 4 that are inspired by the CT threat model but reflect the according ideas in general terms. More specifically, we treat two types of properties. First, we define security notions which concern a malicious logger:

- entry-coll: can a malicious logger present two different sets of entries corresponding to the same fingerprint?
- proof-coll: can a malicious logger present an audit proof that claims a single fingerprint represents both a particular entry as well as a set of entries such that the particular entry is not actually in the list of entries?
- entry-cons: can a malicious logger present two fingerprints connected by a valid consistency proof and two sets of entries such that the entries corresponding to the first fingerprint are not a prefix of the entries corresponding to the second fingerprint?

Second, we define a security notion concerning a malicious monitor:

- promise-incl: can a malicious monitor frame an honest logger for not include a promised entry when it actually has?

Security of Certificate Transparency. Finally, we analyze the security of Certificate Transparency in Section 5 and show that CT both prevents logger misbehaviour (i.e., CT satisfies the entry-coll, proof-coll, and entry-cons security properties) as well as protection from framing of honest loggers by misbehaving monitors (i.e., CT satisfies the promise-incl property.) All of these proofs are based on properties of Merkle tree hashing and audit/consistency proofs, all of which ultimately derive from the collision resistance of the hash function. The last property, promise-incl, also depends on the unforgeability of the signature scheme used by loggers.

Generality of Definitions. Our definition of a logging scheme and its security properties are not specific to CT, and have the potential to be applied to other constructions. In Section 3.4, we discuss the applicability of our definitions to CONIKS [MBB+15], a logging scheme aimed at transparency of user keys: our logging scheme definitions capture some aspects of CONIKS, but also highlights important differences between the functionality and goals of CT versus CONIKS.

1.4 Related Work

New PKI technologies. Recent certificate mis-issuances and security breaches in CAs have motivated research in alternatives to having a trusted third party vouching for the binding between domain name and its private key. *Public key pinning* [EPS15] and *DANE* [HS12] are such proposals that allow domain owners to proactively and directly state their trusted public keys for the domain. Certificate Transparency takes a responsive rather than a proactive approach: instead of preventing mis-issuance in the first place, it aims to *detect* mis-issuance by making certificates visible through a public authenticated log.

History trees. The data structures in CT are similar to the *history trees* of Crosby and Wallach [CW09, Cro09]. Two of their results [Cro09] connect with our security notions: their Corollary 1 shows that "reconstructed hashes" that are equal imply the entry sets from which they were constructed are equal, where "reconstructed hashes" can mean reconstructed from the leaves directly (like in a full hash tree computation) or from membership proofs. Their Theorem 1 shows that, given a consistency proof between two roots and a membership proof for the same index to each root (two membership proofs total), the leaves at that index must be the same in both trees; this is similar to our entry-cons property, though we focus on entry sets rather than membership proofs. A limitation of Crosby's results is that they assume that each root was computed from an underlying entry set, but one cannot be sure when the adversary generates roots (as in CT); our definitions make no such assumption. We furthermore capture

several extensions that CT makes, including delays for entry inclusion and protection of honest loggers from framing (our promise-incl property). Finally, our presentation is notably different: Crosby's descriptions of the history tree operations and the proofs [Cro09, §3] are generally descriptive rather than algorithmic, whereas we state the operations fully algorithmically and provide complete algorithmic reductions for all proofs.

Transparency logs. In recent years a few more approaches have emerged around the concept of *transparency logs*. Because CT itself does not cover revocation of certificates, Laurie and Kasper [LK12] propose Revocation Transparency as a mechanism to handle these aspects. Ryan [Rya14] introduces *Enhanced Certificate Transparency* which is an alternative mechanism for handling revocation and has applied it specifically to end-to-end encrypted email. To date, however, neither of these proposals nor any other is under consideration by the IETF Public Notary Transparency working group, so we omit the study of revocation from this work. Apart from revocation, more advanced features such as limitations on certificate issuance, validation, and update have been incorporated in some proposals [BCK+14, KHP+13].

The Electronic Frontier Foundation's *Sovereign Keys Project* [Ele] constitutes a system where certificates are cross-signed by sovereign keys to be considered valid. Sovereigns keys are then published in a semi-centralized, append-only data structure called "timeline servers" which differ from Certificate Transparency in particular by not using Merkle trees. Melara et al. [MBB+15] present *CONIKS*, a system that builds on transparency logs using Merkle trees similar to Certificate Transparency, but in contrast focuses on transparency of *user* keys in end-to-end encryption/secure messaging scenarios. CONIKS eliminates the need for global third party monitors and aims at additional privacy properties for identity–key bindings, however without providing a formal security model or cryptographic proofs.

Other fields beyond PKI have also embraced notions of transparency logs. Secure Untrusted Data Repository *SUNDR* was introduced by Li et al. [LKMS04], which considers securing data against unauthorized user modification. Like CT, SUNDR is concerned with detection (in this case, ensuring users have the same view of the modification history when the data is stored on an untrusted server), opposed to prevention of attacks. In addition, SUNDR achieves so-called *fork consistency* against untrusted servers, a similar property to our notions of entry-coll, which allow users to detect differences in user views of modification history if they can communicate between themselves. These examples provide evidence of the potential of our security model to be used in analysis of protocols other than CT.

Merkle trees. Introduced by Merkle [Mer79], Merkle trees have been used in many areas of cryptography and computer science, including in the construction of public key signatures from hash functions [Mer90]. Most uses of Merkle trees concern a static dataset, but in CT we are concerned with a dynamic dataset, and in particular the append-only nature of the dataset. (We note the distinction between dynamic updating of datasets and work on amortizing the cost of generating various authentication paths from trees over a static dataset (e.g., [Szy04]).)

There has been some work on authentication trees and more generally signatures on dynamic data sets. Bellare et al. [BGG94, BM97] introduced the notion of *incremental cryptography*: for example, after signing a message, the signer can produce a signature on a closely related ("incremental") message more efficiently. Bellare et al. immediately dismiss Merkle trees for their application due to the size requirements of storing the internal state, but this is not a problem in CT.

Naor and Nissim [NN98] use dynamic Merkle trees (although specifically 2-3 trees rather than binary Merkle trees) in the context of certificate revocation and updates: this allows a CA and directory to synchronize on their view of revoked, new, and updated certificates by sending only the list of updated nodes and the new root of the tree. This differs from the consistency proofs in CT which focus only on appending nodes and only send the (logarithmically-many)

intermediate nodes required to connect two tree roots. Li et al. [LHKR06] use dynamic Merkle B-trees to authenticate index structures in outsourced databases, but allow arbitrary updates (insertions/deletions).

Villemson [Vil02] investigated the characteristics of incremental *authentication graphs*, which are a generalization of Merkle trees to arbitrary graph structures. Villemson focuses on incremental graphs where previous graphs are subgraphs of later graphs. Similarly, Ogawa et al. [OHO05] investigate incremental Merkle trees, but where each incremental tree is a subgraph of a single overall tree. In CT consistency proofs, by contrast, while the *leaves* of an earlier tree are a subset of the leaves of a later tree, the earlier tree is not itself a subgraph of the later tree since the intermediate nodes will "rebalance" as more leaves are added.

Cryptographic PKI analyses. Maurer [Mau96] introduced a formal model for public key infrastructures (PKIs) which subsequently was further extended [MS05, BKH13]. This line of work approaches the dynamic nature of PKI issuance through an event-based system that captures the view of potential users at a certain point in time, using a combination of events that have happened and logical rules that infer certain conclusions from events. Our work differs from this approach by following a game-based approach focusing on the interaction between the parties involved. Our approach also conceptually distinguishes between values generated by honest parties, claims by dishonest parties, and conclusions drawn from events.

2 Cryptographic Building Blocks

In this section we review cryptographic building blocks involved in Certificate Transparency.

Notation. We denote by $\vec{E}$ an ordered list of entries, where () denotes the empty list. Indexing is 0-based: $\vec{E} = (e_0, \ldots, e_{n-1})$, and we write $\vec{E}[i]$ to denote e_i and $\vec{E}[i:j]$ to denote the sublist $(e_i, \ldots, e_{j-1})$. We adopt the convention that $\vec{E}[-1] = ()$. We write $e \in \vec{E}$ to indicate that an entry e is contained in the list $\vec{E}$. We let $\vec{E} \| \vec{E}'$ denote the concatenation of two entry lists and write $\vec{E} \prec \vec{E}'$ if $\vec{E}$ is a prefix of $\vec{E}'$. If we define $P \leftarrow (t, e, \sigma)$, then we can later access fields of P using "object-oriented" notation: $P.t$, $P.e$, $P.\sigma$. Moreover, if $\vec{P}$ is a list $(P_0, \ldots, P_{n-1})$, then the notation $\vec{P}.e$ means the list $(P_0.e, \ldots, P_{n-1}.e)$. The expression $k \leftarrow 2^{\lceil \log_2(n/2) \rceil}$ corresponds to setting k to be the largest power of two less than n, i.e., $\frac{n}{2} \leq k = 2^i < n$.

Definition 1 (Signature scheme unforgeability)**.** Let $\mathcal{M}$ be a set. A digital signature scheme is defined as a tuple of algorithms $\mathtt{SIG} = (\text{KeyGen}, \text{Sign}, \text{Vfy})$:

- $\text{KeyGen}() \xrightarrow{\$} (pk, sk)$: A probabilistic key generation algorithm that outputs a public key pk and secret key sk.
- $\text{Sign}_{sk}(m) \xrightarrow{\$} \sigma$: A probabilistic signing algorithm that takes as input a secret key sk a message $m \in \mathcal{M}$ and outputs a signature σ.
- $\text{Vfy}_{pk}(m, \sigma) \to \{0, 1\}$: A deterministic signature verification algorithm that takes as input a public key pk, message m, and signature σ, and outputs either 0 or 1.

The security experiment $\text{Exp}^{\mathsf{euf\text{-}cma}}_{\mathtt{SIG}}$ for *existential unforgeability under chosen-message attacks* is given in Figure 2. If $\mathcal{A}$ is an algorithm, we define $\text{Adv}^{\mathsf{euf\text{-}cma}}_{\mathtt{SIG}}(\mathcal{A}) = \Pr\left[\text{Exp}^{\mathsf{euf\text{-}cma}}_{\mathtt{SIG}}(\mathcal{A}) = 1\right]$.

Definition 2 (Hash collision finding)**.** Let $\mathcal{M}$ be a set, let $\text{H} : \mathcal{M} \to \{0,1\}^\lambda$ be an unkeyed hash function, and let $\mathcal{A}$ be an algorithm. We say that $\mathcal{A}$ *finds a collision in* H if $\mathcal{A}$ outputs a pair (m, m') such that $m \neq m'$ and $\text{H}(m) = \text{H}(m')$.

2.1 Merkle Trees

The use of hash trees for authenticating large amounts of data was first proposed by Merkle [Mer79, Mer90]. Let $\text{H} : \{0,1\}^* \to \{0,1\}^\lambda$ be a hash function. In a Merkle hash tree for $\vec{E}$, the values of

$\underline{\mathrm{Exp}^{\text{euf-cma}}_{\mathtt{SIG}}(\mathcal{A})}$:
1: $(pk, sk) \xleftarrow{\$} \mathrm{KeyGen}()$
2: $M \leftarrow \{\}$
3: $(m, \sigma) \xleftarrow{\$} \mathcal{A}^{\mathsf{OSign}}(pk)$
4: **return** $\mathrm{Vfy}_{pk}(m, \sigma) \wedge (m \notin M)$

$\underline{\mathsf{OSign}(m)}$:
1: $\sigma \leftarrow \mathrm{Sign}_{sk}(m)$
2: $M \leftarrow M \cup \{m\}$
3: **return** (m, σ)

Figure 2: Security experiment for existential unforgeability under chosen message attack of a signature scheme $\mathtt{SIG} = (\mathrm{KeyGen}, \mathrm{Sign}, \mathrm{Vfy})$.

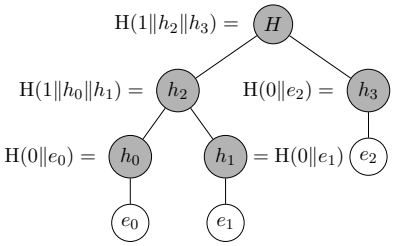


Figure 3: Merkle tree hash calculation of $H = \mathtt{MTH}_{\mathrm{H}}(\vec{E})$ where $\vec{E} = (e_0, e_1, e_2)$. ○ denotes leaf nodes, ● denotes inner nodes.

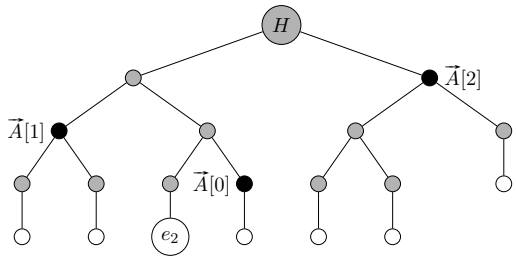


Figure 4: Merkle tree authentication path $\vec{A}$ from leaf e_2 to root H. ● denotes nodes corresponding to authentication path values.

$\vec{E}$ are placed at the leaves of a binary tree and each intermediate node is the hash of its two child nodes; the root of the trees acts as a fingerprint of all the data contained in the tree; this is the output of the algorithm $\mathtt{MTH}_{\mathrm{H}}(\vec{E})$ in Figure 6. A sample Merkle tree hash calculation is shown in Figure 3. Note the use of prefixes 0 and 1 in hash function calculations provides "domain separation" between hash calculations for leaves $(\mathrm{H}(0\| \ldots))$ and intermediate nodes $(\mathrm{H}(1\| \ldots))$; preventing an attacker from gluing part of a tree into a leaf or vice versa.

A common technique is the use of an *authentication path* to demonstrate that a piece of data is in a leaf of a tree corresponding to a particular root. For example, in Figure 4, the authentication path $\vec{A} = (\vec{A}[0], \vec{A}[1], \vec{A}[2])$ shows that e_2 is the third leaf in the tree corresponding to root H, and this can be verified by computing $h_0 = \mathrm{H}(0\|e_2)$, then $h_1 = \mathrm{H}(1\|h_0\|\vec{A}[0])$, then $h_2 = \mathrm{H}(1\|\vec{A}[1]\|h_1)$, then $H' = \mathrm{H}(1\|h_2\|\vec{A}[2])$ and comparing H' with H. The index of the leaf is an essential part of verifying an authentication path. The authentication path generation algorithm $\mathtt{Path}_{\mathrm{H}}(m, \vec{E})$ and verification algorithm $\mathtt{CheckPath}_{\mathrm{H}}(e, H, n, \vec{A}, m)$ are shown in Figure 6.

A lesser-known technique is the use of a *consistency proof* to demonstrate that the data corresponding to one root is a subset (prefix) of the data corresponding to another root, used, for example, in the context of tamper-evident history trees [CW09, Cro09]. In Figure 5, the consistency proof $\vec{C}$ shows that the data corresponding to root H_0 is a prefix of the data corresponding to root H_1. Consistency proofs reconstruct each of the two roots from relevant parts of the proof and compare them against the actual roots; the size of the two trees is essential in verifying a consistency proof. Consistency proofs may be viewed as an authentication path

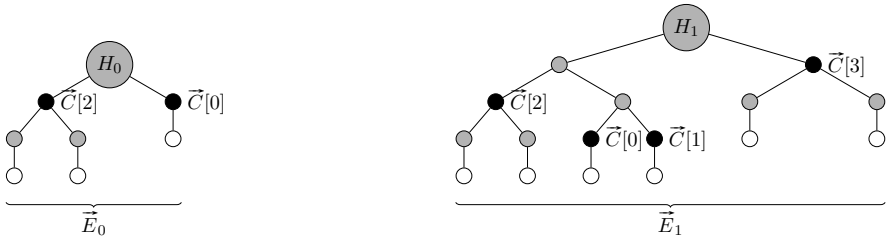


Figure 5: Merkle tree consistency proof $\vec{C} = (\vec{C}[0], \ldots, \vec{C}[3])$ between roots H_0 (for a tree of size 3) and H_1 (for a tree of size 6). ● denotes nodes corresponding to consistency proof values.

from the inner node immediately above the last leaf node in the first tree (i.e., an authentication path from $\mathrm{H}(e_2) = \vec{C}[0]$ to root H_1 in the right side of Figure 5). The consistency proof generation algorithm $\texttt{ConsProof}_{\mathrm{H}}(m, n, \vec{E})$ and verification algorithm $\texttt{CheckConsProof}_{\mathrm{H}}(n_0, H_0, n_1, H_1, \vec{C})$ are shown in Figure 6. We have reformulated these from how they appear in the RFC [LLK13]: ours use a top-down recursive approach, whereas the RFC versions are bottom-up looping algorithms; the two are equivalent, but our versions are more helpful in proving our theorems.

The precise definitions of Merkle tree algorithms follow.

Definition 3 (Merkle tree hash)**.** Let $\mathrm{H} : \{0,1\}^* \to \{0,1\}^\lambda$ be a hash function and let $\vec{E}$ be a list of entries. The *Merkle tree hash* system consists of the following algorithms:

- $\texttt{MTH}_{\mathrm{H}}(\vec{E}) \to h$: A deterministic tree hashing algorithm that takes as input a list of entries $\vec{E}$ each of which is in $\mathcal{M}$ and outputs a hash value $h \in \{0,1\}^\lambda$ representing the complete list of entries as defined in Figure 6.
- $\texttt{Path}_{\mathrm{H}}(m, \vec{E}) \to \vec{A}$: A deterministic authentication path generation algorithm that takes as input an index $m \in \{0, \ldots, |\vec{E}| - 1\}$ and a list of entries $\vec{E}$ and outputs an authentication path $\vec{A}$ as defined in Figure 6.
- $\texttt{CheckPath}_{\mathrm{H}}(e, H, n, \vec{A}, m) \to \{0,1\}$: A deterministic authentication path verification algorithm that takes as input an entry $e \in \mathcal{M}$, a hash $H \in \{0,1\}^\lambda$ representing a list of n entries, a claimed authentication path $\vec{A}$, and an index $m \in \{0, \ldots, n-1\}$, and outputs 0 or 1 as defined in Figure 6. The goal is to output 1 if and only if $\vec{A}$ is an authentication path demonstrating that e is the m-th leaf of the n-leaf tree represented by hash value H.
- $\texttt{ConsProof}_{\mathrm{H}}(m, n, \vec{E}) \to \vec{C}$: A deterministic consistency proof generation algorithm that takes as input a list of entries $\vec{E}$, and two indices $0 \le m \le n \le |\vec{E}|$, and outputs a consistency proof $\vec{C}$ as defined in Figure 6. $\vec{C}$ consists of intermediate nodes required to simultaneously construct the roots of the trees $\vec{E}[0:m]$ and $\vec{E}[0:n]$.
- $\texttt{CheckConsProof}_{\mathrm{H}}(n_0, H_0, n_1, H_1, \vec{C}) \to \{0,1\}$: A deterministic consistency proof verification algorithm that takes as input two indices $0 \le n_0 \le n_1$, two hash values $H_0, H_1 \in \{0,1\}^*$, and a claimed consistency proof $\vec{C}$, and outputs 0 or 1 as defined in Figure 6. The goal is to output 1 if and only if the n_0 entries represented by H_0 are a prefix of the n_1 entries represented by H_1.

2.2 Merkle Tree Security Properties

We now note some well-known facts about the collision resistance of Merkle tree hashing and the security of authentication paths in Merkle trees [Mer79, Mer90]. For completeness, the facts are proven in Appendix A.

Lemma 1 (Collision Resistance of Merkle Trees)**.** *If* H *is collision-resistant, then Merkle-tree hashing using* H *is also collision-resistant. More precisely, if* $\mathcal{A}$ *finds a collision in* $\texttt{MTH}_{\mathrm{H}}$*, then there exists algorithm* $\mathcal{B}_1^{\mathcal{A}}$ *given in Figure 11 in Appendix A that finds a collision in* H*. Moreover, the runtime of* $\mathcal{B}_1^{\mathcal{A}}$ *consists of the runtime of* $\mathcal{A}$*, plus at most a quadratic (in the size of the larger list) number of hash evaluations.*

$\underline{\texttt{MTH}_{\mathrm{H}}(\vec{E}) \to H\text{:}}$

$n \leftarrow |\vec{E}|$
if $n = 1$, **return** $\mathrm{H}(0 \| \vec{E}[0])$
else $(n > 1)$
  $k \leftarrow 2^{\lceil \log_2(n/2) \rceil}$
  return $\mathrm{H}(1 \| \texttt{MTH}_{\mathrm{H}}(\vec{E}[0:k])$
    $\| \texttt{MTH}_{\mathrm{H}}(\vec{E}[k:n]))$

$\underline{\texttt{Path}_{\mathrm{H}}(m, \vec{E}) \to \vec{A}\text{:}}$

$n \leftarrow |\vec{E}|$
if $n = 1$, **return** $()$
else $(n > 1)$
  $k \leftarrow 2^{\lceil \log_2(n/2) \rceil}$
  if $m < k$
    return $\texttt{Path}_{\mathrm{H}}(m, \vec{E}[0:k])$
      $\| \texttt{MTH}_{\mathrm{H}}(\vec{E}[k:n]))$
  else $(m \geq k)$
    return $\texttt{Path}_{\mathrm{H}}(m-k, \vec{E}[k:n])$
      $\| \texttt{MTH}_{\mathrm{H}}(\vec{E}[0:k]))$

$\underline{\texttt{CheckPath}_{\mathrm{H}}(e, H, n, \vec{A}, m) \to \{0,1\}\text{:}}$

$H' \leftarrow \texttt{RootFromPath}_{\mathrm{H}}(e, n, \vec{A}, m)$
return $(H = H')$

$\underline{\texttt{RootFromPath}_{\mathrm{H}}(e, n, \vec{A}, m) \to H\text{:}}$

if $n = 1$, **return** $\mathrm{H}(0 \| e)$
$k \leftarrow 2^{\lceil \log_2(n/2) \rceil}$
if $m < k$
  $\ell \leftarrow \texttt{RootFromPath}_{\mathrm{H}}(e, k, \vec{A}[0:|\vec{A}|-1], m)$
  $r \leftarrow \vec{A}[|\vec{A}|-1]$
else $(m \geq k)$
  $\ell \leftarrow \vec{A}[|\vec{A}|-1]$
  $r \leftarrow \texttt{RootFromPath}_{\mathrm{H}}(e, n-k,$
    $\vec{A}[0:|\vec{A}|-1], m-k)$
return $\mathrm{H}(1 \| \ell \| r)$

$\underline{\texttt{ConsProof}_{\mathrm{H}}(m, n, \vec{E}) \to \vec{C}\text{:}}$

// require: $0 \leq m \leq n \leq |\vec{E}|$
if $m = n$
  return $()$
else $(m < n)$
  return $\texttt{ConsProofSub}_{\mathrm{H}}(m, \vec{E}[0:n], \text{true})$

$\underline{\texttt{ConsProofSub}_{\mathrm{H}}(m, \vec{E}, b) \to \vec{C}\text{:}}$

$n \leftarrow |\vec{E}|$
if $(m = n) \wedge (b = \text{false})$
  return $\texttt{MTH}_{\mathrm{H}}(\vec{E}[0:m])$
else
  $k \leftarrow 2^{\lceil \log_2(n)/2 \rceil}$
  if $m \leq k$
    return $\texttt{ConsProofSub}_{\mathrm{H}}(m, \vec{E}[0:k], b)$
      $\| \texttt{MTH}_{\mathrm{H}}(\vec{E}[k:n]))$
  else $(m > k)$
    return $\texttt{ConsProofSub}_{\mathrm{H}}(m-k, \vec{E}[k:n], \text{false})$
      $\| \texttt{MTH}_{\mathrm{H}}(\vec{E}[0:k]))$

$\underline{\texttt{CheckConsProof}_{\mathrm{H}}(n_0, H_0, n_1, H_1, \vec{C}) \to b\text{:}}$

if n_0 is a power of two, $\vec{C} \leftarrow H_0 \| \vec{C}$
$H'_0 \leftarrow \texttt{Root0FromConsProof}_{\mathrm{H}}(\vec{C}, n_0, n_1)$
$H'_1 \leftarrow \texttt{Root1FromConsProof}_{\mathrm{H}}(\vec{C}, n_0, n_1)$
return $((H_0 = H'_0) \wedge (H_1 = H'_1))$

$\underline{\texttt{Root0FromConsProof}_{\mathrm{H}}(\vec{C}, n_0, n_1) \to H\text{:}}$

$k \leftarrow 2^{\lceil \log_2(n_1)/2 \rceil}$
if $n_0 < k$
  return $\texttt{Root0FromConsProof}_{\mathrm{H}}(\vec{C}[0:|\vec{C}|-1], n_0, k)$
elsif $n_0 = k$, **return** $\vec{C}[|\vec{C}|-2]$
else
  $\ell \leftarrow \vec{C}[|\vec{C}|-1]$
  $r \leftarrow \texttt{Root0FromConsProof}_{\mathrm{H}}(\vec{C}[0:|\vec{C}|-1],$
    $n_0-k, n_1-k)$
  return $\mathrm{H}(1 \| \ell \| r)$

$\underline{\texttt{Root1FromConsProof}_{\mathrm{H}}(\vec{C}, n_0, n_1) \to H\text{:}}$

if $|\vec{C}| = 2$, **return** $\mathrm{H}(1 \| \vec{C}[0] \| \vec{C}[1])$
$k \leftarrow 2^{\lceil \log_2(n_1)/2 \rceil}$
if $n_0 < k$
  $\ell \leftarrow \texttt{Root1FromConsProof}_{\mathrm{H}}(\vec{C}[0:|\vec{C}|-1], n_0, k)$
  $r \leftarrow \vec{C}[|\vec{C}|-1]$
else
  $\ell \leftarrow \vec{C}[|\vec{C}|-1]$
  $r \leftarrow \texttt{Root1FromConsProof}_{\mathrm{H}}(\vec{C}[0:|\vec{C}|-1],$
    $n_0-k, n_1-k)$
return $\mathrm{H}(1 \| \ell \| r)$

Figure 6: Merkle tree algorithms

Lemma 2 (Authentication Paths Consistency)**.** *If* H *is collision-resistant, then no* $\texttt{CheckPath}_{\mathrm{H}}$ *authentication path* $\vec{A}$ *can be generated with respect to Merkle-tree hashing* $\texttt{MTH}_{\mathrm{H}}$ *for an entry* e *not contained in the Merkle tree. More precisely, if* $\mathcal{A}$ *outputs* $(e, \vec{E}, \vec{A}, m)$ *such that* $\texttt{CheckPath}_{\mathrm{H}}(e, \texttt{MTH}_{\mathrm{H}}(\vec{E}), |\vec{E}|, \vec{A}, m) = 1$ *and* $e \notin \vec{E}$*, then there exists algorithm* $\mathcal{B}_2^{\mathcal{A}}$ *given in Figure 12 in Appendix A that finds a collision in* H*. Moreover, the runtime of* $\mathcal{B}_2^{\mathcal{A}}$ *consists of the runtime of* $\mathcal{A}$*, plus at most a quadratic (in* $|\vec{E}|$*) number of hash evaluations.*

3 Logging Schemes

In this section we specify the algorithms that comprise a logging scheme and formulate CT as a logging scheme.

3.1 Definition of Logging Schemes

Our definition of a logging scheme is based around the certificate transparency functionality, but is designed to be potentially more general. We use non-CT specific language (such as "fingerprint"

instead of the CT-specific "signed tree head"), and our logging scheme is not actually about certificates—any type of object can be logged.

Definition 4 (Logging Scheme)**.** A *logging scheme* LS consists of the following algorithms, some of which are run by a logger and some of which are run by a monitor/auditor.

The following algorithm is used by a logger to initialize its log:

- KeyGen() $\xrightarrow{\$}$ (st, pk, sk): A probabilistic algorithm that returns a state st and a public key/secret key pair (pk, sk).

The following algorithms are used by a logger to add entries to its log, using a two-step process of promising to add an entry to the log and then a batch update actually adding the entries:

- PromiseEntry$(e, t, sk) \xrightarrow{\$} P$: A probabilistic algorithm that takes as input a log entry e, a time t, and the secret key sk and outputs a promise P; the promise contains the entry and time as subfields $P.e$ and $P.t$.
- UpdateLog$(st, \vec{P}, t, sk) \xrightarrow{\$} (st', F)$: A probabilistic algorithm that takes as input a state st, a potentially empty ordered list of promises $\vec{P}$ to add to the log, a time t and the secret key sk and returns an updated state st' and a fingerprint F (where the latter includes the indicated time, denoted as $F.t$)

The following algorithms are used by a logger to demonstrate various properties to monitors/auditors:

- PresentEntries$(st, F) \to \vec{E}$ or $\perp$: A deterministic algorithm that takes as input a state st and a fingerprint F and outputs an ordered list of log entries $\vec{E}$, or an error symbol $\perp$.
- ProveMembership$(st, e, F) \xrightarrow{\$} \overrightarrow{M}$ or $\perp$: A probabilistic algorithm[3] that takes as input a state st, a log entry e, and a fingerprint F and outputs a membership proof $\overrightarrow{M}$, or an error symbol $\perp$.
- ProveConsistency$(st, F_0, F_1) \xrightarrow{\$} \vec{C}$ or $\perp$: A probabilistic algorithm[3] that takes as input a state st and two fingerprints F_0 and F_1 and outputs a consistency proof $\vec{C}$, or an error symbol $\perp$.

The following algorithms are used by monitors/auditors to check a log:

- CheckPromise$(P, pk) \to \{0, 1\}$: A deterministic algorithm that takes as input a promise P (which includes an entry $P.e$) and a public key pk and outputs a bit $b \in \{0, 1\}$.
- CheckFingerprint$(F, pk) \to \{0, 1\}$: A deterministic algorithm that takes as input a fingerprint F and a public key pk and outputs a bit $b \in \{0, 1\}$.
- CheckEntries$(\vec{E}, F, pk) \to \{0, 1\}$: A deterministic algorithm that takes as input an ordered list of log entries $\vec{E}$, a fingerprint F, and a public key pk and outputs a bit $b \in \{0, 1\}$.
- CheckMembership$(F, e, \overrightarrow{M}, pk) \to \{0, 1\}$: A deterministic algorithm that takes as input a fingerprint F, an entry e, a membership proof $\overrightarrow{M}$, and a public key pk and outputs a bit $b \in \{0, 1\}$.
- CheckConsistency$(F_0, F_1, \vec{C}, pk) \to \{0, 1\}$: A deterministic algorithm that takes as input two fingerprints F_0 and F_1, a consistency proof $\vec{C}$, and a public key pk and outputs a bit $b \in \{0, 1\}$.

Definition 5 (Correctness of a Logging Scheme)**.** We say that a logging scheme LS is *correct* if for all $(st_0, pk, sk) \xleftarrow{\$}$ KeyGen(), all $n \in \mathbb{N}$, all $m_1, \ldots, m_n \in \mathbb{N}$, all ordered lists of entries $\vec{E}_1 = (e_{1,1}, \ldots, e_{1,m_1}), \ldots, \vec{E}_n = (e_{n,1}, \ldots, e_{n,m_n})$, all ordered lists of promised entries $\vec{P}_1 = (P_{1,1}, \ldots, P_{1,m_1}), \ldots, \vec{P}_n = (P_{n,1}, \ldots, P_{n,m_n})$, all timestamps $t_1 \leq \cdots \leq t_n$, all states $st_1, \ldots, st_n$, and all fingerprints $F_1, \ldots, F_n$ such that $P_{i,j} \xleftarrow{\$}$ PromiseEntry$(e_{i,j}, t_{i,j}, sk)$ for $i \in \{1, \ldots, n\}$, $j \in \{1, \ldots, m_i\}$, and any timestamps $t_{i,j}$ and $(st_1, F_1) \xleftarrow{\$}$ UpdateLog$(st_0, \vec{P}_1, t_1, sk), \ldots, (st_n, F_n) \xleftarrow{\$}$ UpdateLog$(st_{n-1}, \vec{P}_n, t_n, sk)$ the following holds:

[3] In CT, ProveMembership and ProveConsistency are deterministic, though in principle these could be probabilistic in a logging scheme.

1. PresentEntries$(st_n, F_n) = \vec{E}_1 \| \ldots \| \vec{E}_n$.
2. CheckEntries$(\vec{E}_1 \| \ldots \| \vec{E}_n, F_n, pk) = 1$.
3. CheckFingerprint$(F_i, pk) = 1$ for all $1 \leq i \leq n$.
4. CheckConsistency$(F_i, F_j, \vec{C}, pk) = 1$ for all $1 \leq i < j \leq n$ and $\vec{C} \xleftarrow{\$}$ ProveConsistency(st_n, F_i, F_j).
5. CheckMembership$(F_n, e, \vec{M}, pk) = 1$ for all $e \in \vec{E}_1 \| \ldots \| \vec{E}_n$ and $\vec{M} \xleftarrow{\$}$ ProveMembership(st_n, e, F_n).
6. CheckPromise$(e_{i,j}, P_{i,j}, pk) = 1$ for all $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m_i\}$.

3.2 Instantiation of Certificate Transparency as a Logging Scheme

In this subsection we formulate Certificate Transparency using H and SIG as a logging scheme $\mathtt{CT}_{\mathrm{H},\mathtt{SIG}}$ (i.e., following Definition 4).

The Certificate Transparency logging scheme $\mathtt{CT}_{\mathrm{H},\mathtt{SIG}}$ employs a cryptographic hash function $\mathrm{H} : \{0,1\}^* \to \{0,1\}^\lambda$ and a signature scheme SIG (see Section 2 for a formal definition of these). The hash function H is in particular used to store the managed entries in a Merkle hash tree [Mer79, Mer90] as introduced in Section 2.1. Its root node is used within the fingerprint value, called signed tree head *STH* in Certificate Transparency, representing the entries contained at a certain time. Certificate Transparency employs authentication paths to prove the existence of a certain entry in the tree and consistency proofs to connect subsequently published root nodes; see Section 2.1 for a detailed description of these techniques.

A log entry in CT is a chain of X.509 certificates: the certificate itself, and each intermediate CA's certificate leading to the root CA's cert. The initial certificate may be a pre-certificate, which is a partial X.509 data structure which has not yet been signed. As already mentioned, we treat entries in our formalization of logging schemes as opaque bit strings. In particular, our $\mathtt{CT}_{\mathrm{H},\mathtt{SIG}}$ scheme hence omits any syntactical checks for the entries it manages; adding these checks is independent of the logging properties.

KeyGen is used by a logger to initialize the scheme. In CT, it generates a signature public/secret key pair (pk, sk) and initializes the state st with an empty list of entries $\vec{E}$ contained in the log.

$\mathtt{CT}_{\mathrm{H},\mathtt{SIG}}.\mathrm{KeyGen}() \to (st, pk, sk)$:
1: $\vec{E} \leftarrow ()$
2: $st = (\vec{E})$
3: $(pk, sk) \xleftarrow{\$} \mathtt{SIG}.\mathrm{KeyGen}()$
4: **return** (st, pk, sk)

PromiseEntry is used by a logger to output a promise to log a particular certificate/pre-certificate chain as requested by a submitter. In CT, the promise P is called a *signed certificate timestamp* (SCT), and consists of the current timestamp t, the certificate/pre-certificate chain e, and the logger's signature σ over these values.

$\mathtt{CT}_{\mathrm{H},\mathtt{SIG}}.\mathrm{PromiseEntry}(e, t, sk) \to P$:
1: $\sigma \leftarrow \mathtt{SIG}.\mathrm{Sign}_{sk}(t \| e)$
2: **return** $P \leftarrow (t, e, \sigma)$

UpdateLog is used by a logger to incorporate previously promised entries into the log and output a new fingerprint of the log. In CT, the fingerprint F is called the *signed tree head* (STH), and consists of a timestamp t, the number of entries n currently in the log, the root H of the Merkle hash tree of all the entries $\vec{E}$ in the log, and the logger's signature σ over these values.

$\underline{\texttt{CT}_{\text{H},\texttt{SIG}}.\text{UpdateLog}(st, \vec{P}, t, sk) \to (st', F)}$:

1: **for each** $P \in \vec{P}$ **do**
2: **if** CheckPromise$(P, pk) = 0$, **return** $(st, \perp)$
3: $st.\vec{E} \leftarrow st.\vec{E} \| \vec{P}.e$
4: $n \leftarrow |st.\vec{E}|$
5: $H \leftarrow \texttt{MTH}_{\text{H}}(st.\vec{E})$
6: $\sigma \leftarrow \texttt{SIG}.\text{Sign}_{sk}(t, n, H)$
7: **return** $F \leftarrow (t, n, H, \sigma)$

PresentEntries is used by a logger to output all entries associated with a given fingerprint. In CT, the signed tree head (fingerprint) includes the number n of entries comprising that fingerprint, so this algorithm simply outputs the first n entries of the log.

$\underline{\texttt{CT}_{\text{H},\texttt{SIG}}.\text{PresentEntries}(st, F) \to \vec{E}}$:

1: **if** CheckFingerprint$(F, pk) = 0$, **return** $\perp$
2: **return** $st.\vec{E}[0 : F.n]$

ProveMembership is used by a logger to prove that a particular entry is in the log. In CT, the logger constructs an authentication path in the Merkle tree from the leaf node containing the entry to the root of the tree as described in the signed tree head (note that the tree may have subsequently grown since this particular STH was issued).

$\underline{\texttt{CT}_{\text{H},\texttt{SIG}}.\text{ProveMembership}(st, e, F) \to \vec{M}}$:

1: **if** CheckFingerprint$(F, pk) = 0$, **return** $\perp$
2: **find** $m < F.n$ such that $e = st.\vec{E}[m]$
3: **if** no such m exists, **return** $\perp$
4: $\vec{A} \leftarrow \texttt{Path}_{\text{H}}(m, \vec{E}[0 : F.n])$
5: **return** $\vec{M} \leftarrow (\vec{A}, m)$

ProveConsistency is used by a logger to output a proof of consistency between two fingerprints. In CT, this uses the algorithm `ConsProof` (cf. Section 2.1) to construct a consistency proof between the roots in two signed tree heads: the proof consists of the intermediate nodes required to construct both roots simultaneously.

$\underline{\texttt{CT}_{\text{H},\texttt{SIG}}.\text{ProveConsistency}(st, F_0, F_1) \to C}$:

1: **if** CheckFingerprint$(F_0, pk) = 0$, **return** $\perp$
2: **if** CheckFingerprint$(F_1, pk) = 0$, **return** $\perp$
3: **return** $\vec{C} \leftarrow \texttt{ConsProof}_{\text{H}}(F_0.n, F_1.n, st.\vec{E})$

CheckPromise is used by monitors/auditors to confirm that a promise was indeed issued by a specific logger. In CT, this is done by verifying the signature of the signed certificate timestamp using the logger's public key.

$\underline{\texttt{CT}_{\text{H},\texttt{SIG}}.\text{CheckPromise}(P, pk) \to b}$:

1: **return** $\texttt{SIG}.\text{Vfy}_{pk}(P.t \| P.e, P.\sigma)$

CheckFingerprint is used by monitors/auditors to confirm that a fingerprint was indeed issued by a specific logger. In CT, this is done by verifying the signature of the signed tree head using the logger's public key.

$\underline{\texttt{CT}_{\text{H},\texttt{SIG}}.\text{CheckFingerprint}(F, pk) \to b}$:

1: **return** $\texttt{SIG}.\text{Vfy}_{pk}(F.t \| F.n \| F.H, F.\sigma)$

CheckEntries is used by monitors/auditors to confirm that a list of entries corresponds to a given fingerprint. In CT, this is done by reconstructing the Merkle tree hash of the list of entries and comparing that with the root in the signed tree head.

$\texttt{CT}_{\text{H,SIG}}.\text{CheckEntries}(\vec{E}, F, pk) \rightarrow b$:
1: **if** $\text{CheckFingerprint}(F, pk) = 0$, **return** 0
2: $H' \leftarrow \texttt{MTH}_{\text{H}}(\vec{E})$
3: **return** $(|\vec{E}| = F.n) \wedge (H' = F.H)$

CheckMembership is used by monitors/auditors to confirm that a membership proof indeed prove an entry is in the log. In CT, this is done by using the provided authentication path to construct an alleged Merkle tree root, and comparing this with the root in the signed tree head.

$\texttt{CT}_{\text{H,SIG}}.\text{CheckMembership}(F, e, \overrightarrow{M}, pk) \rightarrow b$:
1: **if** $\text{CheckFingerprint}(F, pk) = 0$, **return** 0
2: **return** $\texttt{CheckPath}_{\text{H}}(e, F.H, F.n, \overrightarrow{M}.\vec{A}, \overrightarrow{M}.m)$

CheckConsistency is used by monitors/auditors to confirm the "append-onlyness" of the log has been adhered to – that the list of entries represented by one fingerprint is a prefix of the list of entries represented by another fingerprint. In CT, this is done by using the `CheckConsProof` algorithm to verify a consistency proof between the root values in two signed tree heads by reconstructing both roots from intermediate nodes.

$\texttt{CT}_{\text{H,SIG}}.\text{CheckConsistency}(F_0, F_1, \vec{C}, pk) \rightarrow b$:
1: **if** $\text{CheckFingerprint}(F_0, pk) = 0$, **return** 0
2: **if** $\text{CheckFingerprint}(F_1, pk) = 0$, **return** 0
3: **return** $\texttt{CheckConsProof}(F_0.n, F_0.H, F_1.n, F_1.H, \vec{C})$

3.3 Gossiping Protocol in CT

Certificate Transparency includes an additional mechanism called *gossiping* [NGR15], which allows monitors, auditors, and web clients to share information they receive from log servers, with the goal of collectively detecting misbehaviour of log servers while limiting the damage to user privacy. Gossiping can take the following three forms.

1) *SCT feedback:* Web clients record the signed certificate timestamp values they receive from web servers, and replay those SCT values back to the web server on later visits. If the web client was initially subject to a man-in-the-middle during which it received a fraudulent certificate/SCT, the hope is that MITM attack will eventually cease, and the client will eventually communicate with the real web server, at which point in time the real server will receive the past fraudulently issued SCT from which it can then take action. This message flow is designed to preserve client privacy, as it does not communicate SCT values to a third-party monitor, thereby not informing third-parties of browsing habits.

2) *STH pollination:* Web clients, auditors, and monitors exchange signed tree hashes (fingerprints) they have observed for loggers. To avoid privacy stripping attacks that attempt to uniquely tag a particular client with a particular STH, logs are restricted from issuing STHs too frequently.

3) *Trusted auditor:* Web clients may record SCTs, certificates, and STHs, and send them directly to a trusted monitor/auditor. This is essentially a combination of the previous two streams, but with the drawback that it does not keep the browsing habits of the end-user private from the monitor, and thus end-users are advised to only set up a trusted auditor stream if it is reasonable from a privacy perspective.

The overall goal of gossiping in CT is that interested parties such as monitors and auditors will eventually agree on their view of a log's behaviour. In our model, then, we will assume that gossiping works as intended, and assume that monitors and auditors share all STH values among them (even if this may take a while in practice).

3.4 CONIKS as a Logging Scheme

CONIKS [MBB+15] is a recent transparency log scheme that aims to enable privacy-preserving transparency logging for end-user keys, for applications such as secure messaging. Our definition of logging scheme can capture several aspects of CONIKS' functionality and security, but also serves to highlight some significant differences between CT and CONIKS.

CONIKS also uses a Merkle tree structure to aggregate users' public keys. Whereas CT fills out the leaves of the tree sequentially as log entries are added, CONIKS uses a Merkle *prefix* tree in which an attribute of the log entry (e.g., the user's identity) determines the position of that entry in the tree. The root of the tree is then calculated using both the entries that are present, as well as placeholder values for roots of completely empty subtrees, allowing for efficient root calculation over a very large and mostly empty tree. The root of the tree is signed and published by the logger; this is known as the *signed tree root (STR)*. Membership proofs can be performed in the standard way using Merkle authentication paths. Signed tree roots are linked over time using a hash chain: the current signed tree root includes in its computation a hash of the previous signed tree root. However, this does not enable consistency proofs in the same way as in CT: to verify that a key that was present in STR_i is also present in STR_j, one would have to ask for a fresh membership proof of that key's presence in STR_j. Two core security properties of CONIKS are *non-equivocation*, meaning an identity provider cannot present diverging views to different users; and *privacy-preserving consistency proofs*, meaning consistency of some entries be checked over time without leaking information about other entries.

Concretely, CONIKS can be mapped onto the following notions in our definition of a logging scheme (Definition 4). There is a KeyGen algorithm run by the logger. CONIKS has no separate notion of promise and log entry, so PromiseEntry and UpdateLog could be combined; and CheckFingerprint will verify a signed tree root similarly. CONIKS aims to provide privacy of keys, so by design it does not include PresentEntries and CheckEntries. ProveMembership and CheckMembership are supported. ProveConsistency and CheckConsistency are not directly supported; as noted above, an auditor would need to use ProveMembership and CheckMembership for each entry to be checked for consistency, though there is a mechanism to create a chain of STRs.

In terms of security properties, none of our security properties directly maps onto CONIKS' non-equivocation or privacy-preserving consistency proofs, the primary reason being our definitions include CheckEntries. However, some notions are similar. Non-equivocation is similar to proof-coll, except that it involves two CheckMembership computations, rather than one CheckMembership and one CheckEntries computation (our entry-coll and proof-coll together imply this new notion). Our promise-incl property is also relevant, albeit with a similar change from CheckEntries to CheckMembership, and ignoring maximum merge delay for promises. Consistency of STRs in CONIKS is quite a bit different from our entry-cons property, as CONIKS' auditing scheme for consistency involves probabilistic spot-checks using membership proofs.

4 Security goals

4.1 Threat Model for Certificate Transparency

Recall that in Section 1.2, we summarized the threats against CT in the context of the web PKI as considered in the informational IETF draft [Ken15] which defines potential threats for Certificate Transparency in web public-key infrastructure. Threats are considered regarding

$\underline{\text{Exp}^{\text{entry-coll}}_{\text{LS}}(\mathcal{A})}$:
1: $(\vec{E}_0, \vec{E}_1, F, pk) \xleftarrow{\$} \mathcal{A}()$
2: **return** 1 iff $(\text{CheckEntries}(\vec{E}_0, F, pk) = 1) \ \wedge \ (\text{CheckEntries}(\vec{E}_1, F, pk) = 1) \ \wedge \ (\vec{E}_0 \neq \vec{E}_1)$

$\underline{\text{Exp}^{\text{proof-coll}}_{\text{LS}}(\mathcal{A})}$:
1: $(e, \vec{E}, F, \overrightarrow{M}, pk) \xleftarrow{\$} \mathcal{A}()$
2: **return** 1 iff $(\text{CheckEntries}(\vec{E}, F, pk) = 1) \ \wedge \ (\text{CheckMembership}(e, F, \overrightarrow{M}, pk) = 1) \ \wedge \ (e \notin \vec{E})$

$\underline{\text{Exp}^{\text{entry-cons}}_{\text{LS}}(\mathcal{A})}$:
1: $(\vec{E}_0, \vec{E}_1, F_0, F_1, \vec{C}, pk) \xleftarrow{\$} \mathcal{A}()$
2: **return** 1 iff $(\text{CheckConsistency}(F_0, F_1, \vec{C}, pk) = 1) \ \wedge \ (\text{CheckEntries}(\vec{E}_0, F_0, pk) = 1)$
$\wedge \ (\text{CheckEntries}(\vec{E}_1, F_1, pk) = 1) \ \wedge \ (\vec{E}_0 \not\prec \vec{E}_1)$

Figure 7: Security properties of a logging scheme LS against a malicious logger.

each of the interacting entities: malicious log servers, malicious monitors, malicious CAs, and malicious web clients. We then clustered the threats into three categories: those involving validity or syntax of log entries, those involving failures to notify an affected party of misbehaviour, and those involving attempts to present false information. Our model of logging schemes does not assume a PKI context, so we do not assume that log entries must have a particular syntax, and thus we leave the first class of threats to existing analyses on certificate validity. Similarly, we omit consideration of threats where an entity *fails* to act.

Thus, in our model we focus on threats where an entity attempts to present false information. There are two perspectives: threat (L2), which is that a malicious logger may attempt to present different views of the log to different entities, and threat (M3), where a malicious monitor may attempt to issue false warnings to a domain owner about fake certificates, thereby framing an honest logger or certificate authority for dishonest behaviour.

4.2 Cryptographic Security Properties

For the security properties of logging schemes that can be proved cryptographically, our security definitions follow a provable security game-based approach. We consider three properties involving security against a malicious logger, in which the experiment acts as an honest monitor/auditor which the logger is trying to fool. We also consider one security property involving security against a malicious monitor/auditor, in which the experiment acts as an honest logger which the monitor/auditor is trying to frame for bad behaviour.

4.2.1 Security Against a Malicious Logger

Since the fingerprint (signed tree hash in CT) is used to concisely represent the contents of the log, the first two cryptographic security properties against a malicious logger, shown in Figure 7, concern the ability of the logger to make the fingerprint represent different, conflicting information. *Collision resistance of entries*, defined in the experiment entry-coll, requires that it is hard for a malicious logger to come up with a single fingerprint representing two different sets of entries. *Collision resistance of proofs*, formalized in the experiment proof-coll, is about the difficulty for a malicious logger to create a proof that an entry is represented by a fingerprint while simultaneously claiming that the set of entries represented by that fingerprint does not include that particular entry. A scheme that satisfies both of these ensures that a malicious logger cannot make parties who use the same fingerprint believe different things about the log entries represented by that fingerprint.

Logs are updated over time, but are meant to be append-only. However, since logs are only represented by fingerprints, consistency proofs are used to connect two fingerprints and are meant to prove that the set of entries represented by one fingerprint is a subset of the set of entries represented by a second fingerprint—in other words, that the fingerprints are representative of

$\underline{\text{Exp}^{\text{promise-incl}}_{\text{LS,MMD}}(\mathcal{A})}$:
1: $T \leftarrow 0$
2: $\vec{E}_{promised} \leftarrow ()$
3: $(st, pk, sk) \xleftarrow{\$} \text{KeyGen}()$
4: $(F, P, \vec{E}) \xleftarrow{\$} \mathcal{A}^{\mathsf{OTick},\mathsf{OPromiseEntry},\mathsf{OUpdateLog},\mathsf{OProveConsistency},\mathsf{OProveMembership}}(pk)$
5: **return** 1 iff $(\text{CheckFingerprint}(F, pk) = 1) \ \wedge \ (\text{CheckPromise}(P.e, P, pk) = 1)$
$\wedge \ (\text{CheckEntries}(\vec{E}, F, pk) = 1) \wedge \ (P.e \notin \vec{E}) \ \wedge \ (P.t + \mathsf{MMD} \leq F.t)$

$\underline{\mathsf{OTick}()}$:
1: $T \leftarrow T + 1$
2: $\vec{P} \leftarrow \{P \in \vec{E}_{promised} : P.t + \mathsf{MMD} \leq T\}$
3: **if** $\vec{P} \neq ()$,
4: $\quad F \xleftarrow{\$} \mathsf{OUpdateLog}(\vec{P})$
5: $\quad \vec{E}_{promised} \leftarrow \vec{E}_{promised} \setminus \vec{P}$
6: $\quad$ **return** (T, F)
7: **else return** T

$\underline{\mathsf{OPromiseEntry}(e)}$:
1: $(st, P) \xleftarrow{\$} \text{PromiseEntry}(st, e, T, sk)$
2: $\vec{E}_{promised} \leftarrow \vec{E}_{promised} \,||\, \{P\}$
3: **return** P

$\underline{\mathsf{OUpdateLog}(\vec{P})}$:
1: $(st, F) \xleftarrow{\$} \text{UpdateLog}(st, \vec{P}, T, sk)$
2: **return** F

$\underline{\mathsf{OProveConsistency}(F_0, F_1)}$:
1: $(st, \vec{C}) \xleftarrow{\$} \text{ProveConsistency}(st, F_0, F_1)$
2: **return** $\vec{C}$

$\underline{\mathsf{OProveMembership}(e, F)}$:
1: $(st, \vec{M}) \xleftarrow{\$} \text{ProveMembership}(st, e, F)$
2: **return** $\vec{M}$

Figure 8: Security properties of a logging scheme LS against a malicious monitor/auditor framing a log for failing to include a promised entry.

an append-only log. The final security property in Figure 7 captures the *consistency of entries*, i.e., the difficulty for a malicious logger to remove an entry from a log: experiment entry-cons is concerned with two fingerprints connected by a single consistency proof. A "multi-hop" version, concerned with a chain of fingerprints connected by consistency proofs, can easily be formulated and shown to follow directly from the "single-hop" version.

4.2.2 Security Against a Malicious Monitor/Auditor

The security properties described above are *cryptographic*, meaning that (under some computational assumptions) it is not possible for a malicious logger to perform certain actions. However, there are some security goals of CT that are not cryptographic. For example, a log could choose to omit an entry that it has promised to log, and no amount of cryptography can prevent it from doing so. Should a log issue a fingerprint after the time by which it has promised to log an entry but the log does not contain an entry, that constitutes evidence of the log's misbehaviour.

However, to protect honest loggers, it should not be possible to frame an honest logger for misbehaviour that did not actually happen, which is the security guarantee formalized as *inclusion of promises* in experiment promise-incl in Figure 8. Here the experiment plays the role of an honest logger against a malicious monitor/auditor, so we allow the adversary (the malicious monitor/logger) to interact with experiment oracles that carry out the actions of an honest log, such as adding entries or proving membership. The experiment includes a global time which advances at the adversary's command, and is parameterized by a *maximum merge delay* $\mathsf{MMD} > 0$, within which an honest log is expected to include a promised entry. The list $\vec{E}_{promised}$ tracks entries that the log has promised to include; in calls to OTick the experiment (acting as the honest log) automatically adds the list of promised entries by the end of the maximum merge delay window.

5 Security of Certificate Transparency

We are now ready to prove the security results on Certificate Transparency, namely that its instantiation $\mathtt{CT}_{\mathrm{H},\mathtt{SIG}}$ within our logging scheme frameworks guarantees collision resistance of

entities and proofs, consistency of entries, and inclusion of promises.

Theorem 1 (Collision resistance of entries)**.** *If hash function* H *is collision-resistant, then, in Certificate Transparency (with hash function* H*), no malicious logger can present different log entries for the same fingerprint. More precisely, if* $\mathcal{A}$ *wins* $\mathrm{Exp}^{\mathsf{entry\text{-}coll}}_{\mathtt{CT}_{\mathrm{H},\mathtt{SIG}}}$*, then algorithm* $\mathcal{B}^{\mathcal{A}}$*, which runs* $\mathcal{A}$ *and then returns the first two components of* $\mathcal{A}$*'s output, finds a collision in* $\mathtt{MTH}_{\mathrm{H}}$*. Moreover, the runtime of* $\mathcal{B}^{\mathcal{A}}$ *is the same as that of* $\mathcal{A}$*.*

By Lemma 1, a collision in $\mathtt{MTH}_{\mathrm{H}}$ leads to a collision in H, which is infeasible if H is collision-resistant.

Proof. We show that a successful adversary $\mathcal{A}$ effectively outputs two colliding entry sets under the same root of the Merkle tree used in Certificate Transparency, which, by Lemma 1, leads to a hash collision.

Suppose $\mathcal{A}$ wins $\mathrm{Exp}^{\mathsf{entry\text{-}coll}}_{\mathtt{CT}_{\mathrm{H},\mathtt{SIG}}}$. Then $\mathcal{A}$ will output $(\vec{E}_0, \vec{E}_1, F, pk)$ such that

$$\mathrm{CheckEntries}(\vec{E}_0, F, pk) = \mathrm{CheckEntries}(\vec{E}_1, F, pk) = 1$$

but $\vec{E}_0 \neq \vec{E}_1$. Based on the definition of CheckEntries for $\mathtt{CT}_{\mathrm{H},\mathtt{SIG}}$ (cf. Section 3.2), this implies that $\mathtt{MTH}_{\mathrm{H}}(\vec{E}_0) = \mathtt{MTH}_{\mathrm{H}}(\vec{E}_1)$ but $\vec{E}_0 \neq \vec{E}_1$. This is immediately a collision in $\mathtt{MTH}_{\mathrm{H}}$, which is what $\mathcal{B}$ outputs. □

Theorem 2 (Collision resistance of proofs)**.** *If hash function* H *is collision-resistant then, in Certificate Transparency (with hash function* H*) no malicious logger can present a list of log entries under some fingerprint and a membership proof under the same fingerprint for an entry not contained in this list. More precisely, if* $\mathcal{A}$ *wins* $\mathrm{Exp}^{\mathsf{proof\text{-}coll}}_{\mathtt{CT}_{\mathrm{H},\mathtt{SIG}}}$ *by outputting* $(e, \vec{E}, F, \vec{M}, pk)$*, then algorithm* $\mathcal{B}^{\mathcal{A}}$*, which runs* $\mathcal{A}$ *and then returns* $(e, \vec{E}, \vec{M}.\vec{A}, \vec{M}.m)$*, breaks authentication path consistency in the sense of Lemma 2. Moreover, the runtime of* $\mathcal{B}^{\mathcal{A}}$ *is the same as that of* $\mathcal{A}$*.*

By Lemma 2, a break of authentication path consistency in $\mathtt{MTH}_{\mathrm{H}}$ leads to a collision in H, which is infeasible if H is collision-resistant.

Proof. Suppose $\mathcal{A}$ wins $\mathrm{Exp}^{\mathsf{proof\text{-}coll}}_{\mathtt{CT}_{\mathrm{H},\mathtt{SIG}}}$ by outputting $(e, \vec{E}, F, \vec{M}, pk)$ such that $\mathrm{CheckEntries}(\vec{E}, F, pk) = 1$ and $\mathrm{CheckMembership}(F, e, \vec{M}, pk) = 1$, but $e \notin \vec{E}$. Based on the definition of CheckEntries and CheckMembership for $\mathtt{CT}_{\mathrm{H},\mathtt{SIG}}$ (cf. Section 3.2), this implies that $|\vec{E}| = F.n$, $\mathtt{MTH}_{\mathrm{H}}(\vec{E}) = F.H$, and $\mathtt{CheckPath}_{\mathrm{H}}(e, F.H, F.n, \vec{M}.\vec{A}, \vec{M}.m) = 1$. Outputting the values $(e, \vec{E}, \vec{M}.\vec{A}, \vec{M}.m)$ breaks authentication path consistency in the sense of Lemma 2. □

Theorem 3 (Consistency of entries)**.** *If hash function* H *is collision-resistant, then, in Certificate Transparency (with hash function* H*), no malicious logger can present two lists of entries, two fingerprints, and a consistency proof such that each list corresponds to the fingerprint, and the fingerprints are connected via the consistency proof, but the first list of entries is not a prefix of the second list of entries. More precisely, if* $\mathcal{A}$ *wins* $\mathrm{Exp}^{\mathsf{entry\text{-}cons}}_{\mathtt{CT}_{\mathrm{H},\mathtt{SIG}}}$*, then algorithm* $\mathcal{B}_3^{\mathcal{A}}$ *given in Figure 9 finds a collision in* H*. Moreover, the runtime of* $\mathcal{B}_3^{\mathcal{A}}$ *consists of the runtime of* $\mathcal{A}$*, plus at most a quadratic (in the size of the second list) number of hash evaluations.*

Proof. Suppose $\mathcal{R}_3$ receives as input values $\vec{D}_0, \vec{D}_1, \vec{C}$ such that

$$\mathtt{Root0FromConsProof}_{\mathrm{H}}(\vec{C}, n_0, n_1) = \mathtt{MTH}_{\mathrm{H}}(\vec{D}_0) \tag{1}$$

$$\mathtt{Root1FromConsProof}_{\mathrm{H}}(\vec{C}, n_0, n_1) = \mathtt{MTH}_{\mathrm{H}}(\vec{D}_1) \tag{2}$$

(where $n_0 = |\vec{D}_0|$ and $n_1 = |\vec{D}_1|$) but

$$\vec{D}_0 \not\prec \vec{D}_1. \tag{3}$$

$\mathcal{B}_3^{\mathcal{A}}()$:

1: $(\vec{E}_0, \vec{E}_1, F_0, F_1, \vec{C}, pk) \leftarrow \mathcal{A}()$
2: **if** $F_0.n$ is a power of two, $\vec{C} \leftarrow F_0.H \| \vec{C}$
3: *// assume* $\vec{E}_0 \not\prec \vec{E}_1$
and $\texttt{MTH}_{\texttt{H}}(\vec{E}_0) = \texttt{Root0FromConsProof}_{\texttt{H}}(\vec{C}, F_0.n, F_1.n)$
and $\texttt{MTH}_{\texttt{H}}(\vec{E}_1) = \texttt{Root1FromConsProof}_{\texttt{H}}(\vec{C}, F_0.n, F_1.n)$
4: **return** $\mathcal{R}_3(\vec{E}_0, \vec{E}_1, \vec{C})$

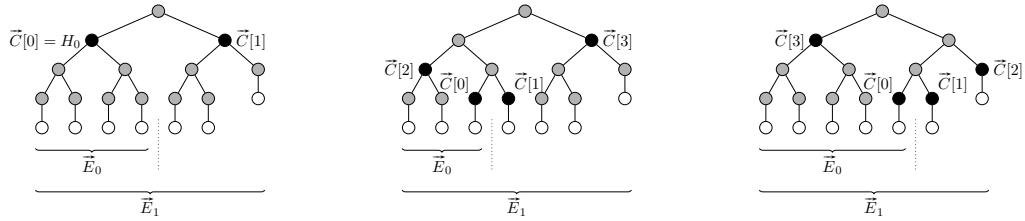
$\mathcal{R}_3(\vec{D}_0, \vec{D}_1, \vec{C})$:

1: $n_0 \leftarrow |\vec{D}_0|$
2: $n_1 \leftarrow |\vec{D}_1|$
3: *// require:* $\texttt{Root0FromConsProof}_{\texttt{H}}(\vec{C}, n_0, n_1) = \texttt{MTH}_{\texttt{H}}(\vec{D}_0)$
4: *// require:* $\texttt{Root1FromConsProof}_{\texttt{H}}(\vec{C}, n_0, n_1) = \texttt{MTH}_{\texttt{H}}(\vec{D}_1)$
5: $k \leftarrow 2^{\lceil \log_2(n_1)/2 \rceil}$
6: $\vec{C}' \leftarrow \vec{C}[0 : |\vec{C}| - 1]$
7: **if** $n_0 < k$
8: $\quad \ell_1 \leftarrow \texttt{MTH}_{\texttt{H}}(\vec{D}_1[0 : k])$
9: $\quad \ell_1' \leftarrow \texttt{Root1FromConsProof}_{\texttt{H}}(\vec{C}', n_0, k)$
10: $\quad r_1 \leftarrow \texttt{MTH}_{\texttt{H}}(\vec{D}_1[k : n_1])$
11: $\quad r_1' \leftarrow \vec{C}[|\vec{C}| - 1]$
12: $\quad$ **if** $(\ell_1 \neq \ell_1') \vee (r_1 \neq r_1')$
13: $\quad\quad$ **return** $(1 \| \ell_1 \| r_1, 1 \| \ell_1' \| r_1')$
14: $\quad$ **else return** $\mathcal{R}_3(\vec{D}_0, \vec{D}_1[0 : k], \vec{C}')$
15: **elsif** $n_0 = k$
16: $\quad \ell_0 \leftarrow \texttt{MTH}_{\texttt{H}}(\vec{D}_0)$
17: $\quad \ell_1 \leftarrow \texttt{MTH}_{\texttt{H}}(\vec{D}_1[0 : k])$
18: $\quad$ **if** $\ell_0 = \ell_1$ **return** $\mathcal{R}_1(\vec{D}_0, \vec{D}_1[0 : k])$
19: $\quad$ **elsif** $\ell_0 \neq \vec{C}[|\vec{C}| - 2]$
20: $\quad\quad$ *// will not occur due to line 2 of* $\mathcal{B}_3$
21: $\quad$ **else** $(\ell_1 \neq C[|C| - 2])$
22: $\quad\quad$ **return** $(1 \| \ell_1 \| \texttt{MTH}_{\texttt{H}}(\vec{D}_1[k : n_1],$
$1 \| \vec{C}[|\vec{C}| - 2] \| \vec{C}[|\vec{C}| - 1]))$

$\mathcal{R}_3$ continued:

23: **else** $(n_0 > k)$
24: $\quad$ **if** $\vec{D}_0[0 : k] = \vec{D}_1[0 : k]$
25: $\quad\quad \ell \leftarrow \texttt{MTH}_{\texttt{H}}(\vec{D}_0[0 : k])$
26: $\quad\quad$ **if** $\ell = \vec{C}[|\vec{C}| - 1]$
27: $\quad\quad\quad r_0 \leftarrow \texttt{MTH}_{\texttt{H}}(\vec{D}_0[k : n_0])$
28: $\quad\quad\quad r_0' \leftarrow \texttt{Root0FromConsProof}(\vec{C}', n_0 - k, n_1 - k)$
29: $\quad\quad\quad$ **if** $r_0 \neq r_0'$, **return** $(1 \| \ell \| r_0, 1 \| \ell \| r_0')$
30: $\quad\quad\quad r_1 \leftarrow \texttt{MTH}_{\texttt{H}}(\vec{D}_1[k : n_1])$
31: $\quad\quad\quad r_1' \leftarrow \texttt{Root1FromConsProof}(\vec{C}', n_0 - k, n_1 - k)$
32: $\quad\quad\quad$ **if** $r_1 \neq r_1'$, **return** $(1 \| \ell \| r_1, 1 \| \ell \| r_1')$
33: $\quad\quad\quad$ **return** $\mathcal{R}_3(\vec{D}_0[k : n_0], \vec{D}_1[k : n_1], \vec{C}')$
34: $\quad\quad$ **else** $(\ell \neq \vec{C}[|\vec{C}| - 1])$
35: $\quad\quad\quad$ **return** $(1 \| \ell \| \texttt{MTH}_{\texttt{H}}(\vec{D}_0[k : n_0]),$
$1 \| \vec{C}[|\vec{C}| - 1]$
$\| \texttt{Root0FromConsProof}_{\texttt{H}}(\vec{C}', n_0 - k, n_1 - k))$
36: $\quad$ **else** $(\vec{D}_0[0 : k] \neq \vec{D}_1[0 : k])$
37: $\quad\quad \ell_0 \leftarrow \texttt{MTH}_{\texttt{H}}(\vec{D}_0[0 : k])$
38: $\quad\quad \ell_1 \leftarrow \texttt{MTH}_{\texttt{H}}(\vec{D}_1[0 : k])$
39: $\quad\quad$ **if** $\ell_0 = \ell_1$, **return** $\mathcal{R}_1(\vec{D}_0[0 : k], \vec{D}_1[0 : k])$
40: $\quad\quad$ **else** $(\ell_0 \neq \ell_1)$
41: $\quad\quad\quad$ **if** $\ell_0 \neq \vec{C}[|\vec{C}| - 1]$
42: $\quad\quad\quad\quad$ **return** $(1 \| \ell_0 \| \texttt{MTH}_{\texttt{H}}(\vec{D}_0[k : n_0]),$
$1 \| \vec{C}[|\vec{C}| - 1]$
$\| \texttt{Root0FromConsProof}_{\texttt{H}}(\vec{C}', n_0 - k, n_1 - k))$
43: $\quad\quad\quad$ **else** $(\ell_1 \neq \vec{C}[|\vec{C}| - 1])$
44: $\quad\quad\quad\quad$ **return** $(1 \| \ell_1 \| \texttt{MTH}_{\texttt{H}}(\vec{D}_1[k : n_1]),$
$1 \| \vec{C}[|\vec{C}| - 1]$
$\| \texttt{Root1FromConsProof}_{\texttt{H}}(\vec{C}', n_0 - k, n_1 - k))$

Figure 9: Algorithm $\mathcal{B}_3$ for Theorem 3.



(a) Case 1: $|\vec{C}| = 2$ (i.e., $n_0 = k$).

(b) Case 2(a): $|\vec{C}| > 2$, $n_0 < k$.

(c) Case 2(b): $|\vec{C}| > 2$, $n_0 > k$.

Figure 10: Tree and consistency path for cases in proof of Theorem 3. $|\vec{E}_0| = n_0$, $|\vec{E}_1| = n_1$, $k = 2^{\lceil \log_2(n_1)/2 \rceil}$. ○ denotes leaf nodes, ● denotes inner nodes, and ● denotes nodes corresponding to consistency proof values.

We claim that $\mathcal{R}_3$ outputs a collision for H when run with values satisfying (1)–(3). The proof of the claim proceeds by induction on the size of $\vec{C}$. $\mathcal{R}_3$ will recursively descend through the trees induced by $\vec{D}_0$ and $\vec{D}_1$, as well as the reconstructions of the two roots from the consistency path. At some point, $\mathcal{R}_3$ will find a collision. The three main cases are represented in Figures 10(a)–10(c).

Case 1. First suppose $|\vec{C}| = 2$. Assume that $n_0 = 2^{\lceil \log_2(n_1)/2 \rceil}$. (See Figure 10(a).) (This is true in honestly constructed consistency proofs; if lengths mismatch in malicious proofs, the algorithms in Figure 6 will access invalid memory addresses and are assumed to abort.)

Now,

$$\texttt{Root1FromConsProof}_{\texttt{H}}(\vec{C}, n_0, n_1) = \text{H}(1 \| \vec{C}[0] \| \vec{C}[1])$$

by `Root1FromConsProof` line 1 in Figure 6. By definition, we have

$$\texttt{MTH}_{\texttt{H}}(\vec{D}_1) = \text{H}(1 \| \texttt{MTH}_{\texttt{H}}(\vec{D}_1[0 : k]) \| \texttt{MTH}_{\texttt{H}}(\vec{D}_1[k : n_1])) \ .$$

By (2),

$$\texttt{Root1FromConsProof}_{\mathrm{H}}(\vec{C}, n_0, n_1) = \texttt{MTH}_{\mathrm{H}}(\vec{D}_1) \ .$$

If $\texttt{MTH}_{\mathrm{H}}(\vec{D}_1[0:k]) \neq \vec{C}[0]$ or $\texttt{MTH}_{\mathrm{H}}(\vec{D}_1[k:n_1]) \neq \vec{C}[1]$, then we have a collision in H: this is what is output by $\mathcal{R}_3$ on line 22.

So we now assume that $\texttt{MTH}_{\mathrm{H}}(\vec{D}_1[0:k]) = \vec{C}[0]$ and $\texttt{MTH}_{\mathrm{H}}(\vec{D}_1[k:n_1]) = \vec{C}[1]$. By line 4 of `Root0FromConsProof` in Figure 6, we have that $\texttt{Root0FromConsProof}_{\mathrm{H}}(\vec{C}, n_0, n_1) = \vec{C}[0]$. By (1), this is equal to $\texttt{MTH}_{\mathrm{H}}(\vec{D}_0)$. And by the assumption at the start of this paragraph, this is also equal to $\texttt{MTH}_{\mathrm{H}}(\vec{D}_1[0:k])$. Since $\vec{D}_0 \not\prec \vec{D}_1$ and $|\vec{D}_0| = k$, we must have that $\vec{D}_0 \neq \vec{D}_1[0:k]$. Thus, $(\vec{D}_0, \vec{D}_1[0:k])$ constitutes a collision for $\texttt{MTH}_{\mathrm{H}}$, and by Lemma 1, this leads to a collision in H: this is what is output by $\mathcal{R}_3$ in its call to $\mathcal{R}_1$ on line 18.

Case 2. Now suppose $|\vec{C}| > 2$.

Case 2(a). First, suppose $n_0 < k$. (See Figure 10(b).) Use the definitions of $\ell_1, \ell_1', r_1, r_1'$ on lines 8–11 of $\mathcal{R}_3$. By lines 4, 5, and 8 of `Root1FromConsProof`, $\mathrm{H}(1\|\ell_1'\|r_1')$ is the value returned by $\texttt{Root1FromConsProof}_{\mathrm{H}}(\vec{C}, n_0, n_1)$. By definition of `MTH`, $\mathrm{H}(1\|\ell_1\|r_1)$ is the value returned by $\texttt{MTH}_{\mathrm{H}}(\vec{D}_1)$. By equation (2), these hashes are equal. If $\ell_1 \neq \ell_1'$ or $r_1 \neq r_1'$, we have a collision in H: this is what is output by $\mathcal{R}_3$ on line 13. Suppose $\ell_1 = \ell_1'$ and $r_1 = r_1'$. Then in particular

$$\texttt{MTH}_{\mathrm{H}}(\vec{D}_1[0:k]) = \texttt{Root1FromConsProof}_{\mathrm{H}}(\vec{C}[0:|\vec{C}|-1]) \ .$$

By line 3 of `Root0FromConsProof` and equation (1), we have that

$$\texttt{MTH}_{\mathrm{H}}(\vec{D}_0) = \texttt{Root0FromConsProof}_{\mathrm{H}}(\vec{C}, n_0, n_1) = \texttt{Root0FromConsProof}_{\mathrm{H}}(\vec{C}[0:|\vec{C}|-1], n_0, k) \ .$$

Moreover, $\vec{D}_0 \not\prec \vec{D}_1[0:k]$. Thus, $\vec{D}_0, \vec{D}_1[0:k, \vec{C}[0:|\vec{C}|-1]$ satisfy conditions (1)–(3). By induction, $\mathcal{R}_3$'s recursive call on line 14 will yield a collision in H.

Case 2(b). Second, suppose $n_0 > k$. (See Figure 10(c).) While we know that $\vec{D}_0 \neq \vec{D}_1[0:n_0]$, there are two possibilities: either $\vec{D}_0[0:k] = \vec{D}_1[0:k]$ but $\vec{D}_0[k:n_0] \neq \vec{D}_1[k:n_0]$, or $\vec{D}_0[0:k] \neq \vec{D}_1[0:k]$.

Suppose $\vec{D}_0[0:k] = \vec{D}_1[0:k]$ but $\vec{D}_0[k:n_0] \neq \vec{D}_1[k:n_0]$. This means that $\texttt{MTH}_{\mathrm{H}}(\vec{D}_0[0:k]) = \texttt{MTH}_{\mathrm{H}}(\vec{D}_1[0:k])$, which is ℓ on line 25 of $\mathcal{R}_3$. Now, either $\ell = \vec{C}[|\vec{C}|-1]$ or not. If $\ell \neq \vec{C}[|\vec{C}|-1]$, then we already have a collision in equation (1), and this is output by line 35. If $\ell = \vec{C}[|\vec{C}|-1]$, then—intuitively—we have to look on the right side of the trees for the collision. If r_0 (the right side of the $\vec{D}_0$ tree) is not equal to r_0' (the right side of the reconstructed path to root zero), then we have a collision:

$$\mathrm{H}(1\|\ell\|r_0) = MTH_{\mathrm{H}}(\vec{D}_0) = \texttt{Root0FromConsProof}_{\mathrm{H}}(\vec{C}, n_0, n_1) = \mathrm{H}(1\|\ell\|r_0') \ ,$$

and this is what $\mathcal{R}_3$ outputs on line 29. Similarly if r_1 (the right side of the $\vec{D}_1$ tree) is not equal to r_1' (the right side of the reconstructed path to root one): this is what $\mathcal{R}_3$ outputs on line 32. Otherwise, $r_0 = r_0'$ and $r_1 = r_1'$, and thus $\vec{D}_0[k:n_0], \vec{D}_1[k:n_1], \vec{C}'$ satisfy conditions (1)–(3): by induction, the recursive call to $\mathcal{B}_3$ on line 33 will return a collision.

Finally, suppose $\vec{D}_0[0:k] \neq \vec{D}_1[0:k]$. Either these two lists hash to the same value under $\texttt{MTH}_{\mathrm{H}}$ or they do not. If they do, then this constitutes a collision in $\texttt{MTH}_{\mathrm{H}}$, and by Lemma 1, this leads to a collision in H: this is what is output by $\mathcal{R}_3$ in its call to $\mathcal{R}_1$ on line 40. If these two lists hash to different values ℓ_0, ℓ_1 under $\texttt{MTH}_{\mathrm{H}}$, then at least one of ℓ_0, ℓ_1 must be different from $\vec{C}[|\vec{C}|-1]$. Suppose $\ell_0 \neq \vec{C}[|\vec{C}|-1]$. By condition (1), the definition of `MTH`, and lines 6–8 of `Root0FromConsProof` in Figure 6, $1\|\ell_0\|\texttt{MTH}_{\mathrm{H}}(\vec{D}_0[k:n_0])$ and $1\|\vec{C}[|\vec{C}|-1]\|\texttt{Root0FromConsProof}_{\mathrm{H}}(\vec{C}', n_0-k, n_1-k)$ hash to the same value but are different strings, constituting a collision for H: this is what $\mathcal{R}_3$ outputs on line 42. Similarly, if $\ell_1 \neq \vec{C}[|\vec{C}|-1]$, we obtain a collision related to condition (2), which is what $\mathcal{R}_3$ outputs on line 44. □

Theorem 4 (Inclusion of promises)**.** *If hash function* H *is collision-resistant and signature scheme* SIG *is existentially unforgeable under chosen-message attacks, then, in Certificate Transparency (with hash function* H *and signature scheme* SIG*), no malicious monitor/auditor can frame an honest logger of not including a promised entry within the maximum merge delay. More precisely, if algorithm* $\mathcal{A}$ *wins* $\mathrm{Exp}^{\mathsf{promise\text{-}incl}}_{\mathtt{CT}_{\mathrm{H},\mathtt{SIG}}}$*, then there exist algorithms* $\mathcal{B}^{\mathcal{A}}$ *and* $\mathcal{C}^{\mathcal{A}}$*, described in the proof, that find a collision in* $\mathtt{MTH}_{\mathrm{H}}$ *or a forgery in* SIG*, respectively. Moreover, the runtimes of* $\mathcal{B}^{\mathcal{A}}$ *and* $\mathcal{C}^{\mathcal{A}}$ *are approximately the same as that of* $\mathcal{A}$*.*

Proof. By definition of OTick (cf. Figure 8, the simulated honest logger will keep track of any promise P issued through OPromiseEntry and will include the P through OUpdateLog by time $T = P.t + \mathsf{MMD}$. As in particular $\mathsf{MMD} > 0$, this ensures that any fingerprint issued by the honest logger at time $T' \geq T$ will include the promised entry $P.e$.

Assume $\mathcal{A}$ wins by outputting $(F, P, \vec{E})$, i.e., F is a valid fingerprint representing entries $\vec{E}$ and P is a promise for an entry $e \notin \vec{E}$ although $P.t + \mathsf{MMD} \leq F.t$. This means either one of the promise P or the fingerprint F (or both) were not issued by the simulated honest logger through an invocation of OPromiseEntry or OUpdateLog, or that $\mathcal{A}$ repeated an honest F that matches an entry list $\vec{E}$ different from the entry list $\vec{E}'$ hold by the honest logger when creating the fingerprint.

The second case constitutes a Merkle-tree hash collision (as $\mathtt{MTH}_{\mathrm{H}}(\vec{E}) = \mathtt{MTH}_{\mathrm{H}}(\vec{E}')$, but $\vec{E} \neq \vec{E}'$). Hence $\mathcal{A}$'s advantage in winning through this case can be bound by the advantage of an algorithm $\mathcal{B}$ (that simulates the oracles and simply outputs the colliding $\vec{E}$ and $\vec{E}'$) against the collision resistance of $\mathtt{MTH}_{\mathrm{H}}$. (Applying Lemma 1 leads to a collision in H.)

For the first case, we show how this allows constructing a signature forgery attacker $\mathcal{C}$ against the euf-cma security of SIG, which works as follows. First of all, $\mathcal{C}$ creates an initial state with empty list of entries. It then simulates experiment $\mathrm{Exp}^{\mathsf{promise\text{-}incl}}_{\mathtt{CT}_{\mathrm{H},\mathtt{SIG}},\mathsf{MMD}}$ for $\mathcal{A}$, providing the public key pk from its euf-cma game as input for $\mathcal{A}$. It furthermore uses its euf-cma signing oracle OSign when required to generate a signature in the simulations of the OPromiseEntry and OUpdateLog oracles and keeps a list of all the values queried to the signing oracle.

If $\mathcal{A}$ halts (outputting $(F, P, \vec{E})$) and wins, as argued above, at least one of P or F was not output through $\mathcal{C}$'s simulation of OPromiseEntry and OUpdateLog (as we excluded the case of a Merkle-tree hash collision). Hence, in particular, the according value was not queried to the euf-cma signing oracle, so $\mathcal{C}$ checks which of the two values is not contained in its list of queries and outputs this as its valid signature forgery. □

Remark 1. Our security results for CT depend on security of Merkle tree hashing, which in turn depend on collision resistance. Some hash-based signature schemes are able to rely on second-preimage resistance [DOTV08, BDH11] or pseudorandomness [BDE+11] by XORing (pseudo)random values into intermediate node computations, but these techniques are not used in CT's use of traditional Merkle trees so we rely solely on collision resistance.

6 Conclusion and Future Work

Certificate Transparency is a promising approach for providing assurances in the web PKI by using untrusted auditable public logs to detect fraudulently issued certificates. We introduced a generic model for logging schemes and captured Certificate Transparency as one specific instance of our model. Based on the security notions we formalized, we were able to analyze the cryptographic aspects of CT and show how its cryptographic mechanisms prevent both undetected misbehaviour of log servers as well as false accusations of honest loggers.

Although cryptography plays an essential role to establish the trust necessary in a public and auditable logging scheme like Certificate Transparency, there are other components involved that are difficult or even impossible to capture in a cryptographic model. For example, under various conditions on adversary control of the network and with various patterns of honest entity

behaviour, how long does it take for the CT gossiping protocol to propagate SCTs and STHs to ensure detection of dishonest log behaviour? Once misbehaviour is detected, what organizational measures should be taken to ensure an appropriate response? Analyzing these components in general as well as their specific relevance in the CT framework is an important task for future work.

Acknowledgements

B.D. and D.S. are supported by Australian Research Council (ARC) Discovery Project grant DP130104304. F.G. is supported by the DFG as part of project S4 within the CRC 1119 CROSSING.

References

[BCK+14] David A. Basin, Cas J. F. Cremers, Tiffany Hyun-Jin Kim, Adrian Perrig, Ralf Sasse, and Pawel Szalachowski. ARPKI: Attack resilient public-key infrastructure. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 14*, pages 382–393. ACM Press, November 2014.

[BDE+11] Johannes Buchmann, Erik Dahmen, Sarah Ereth, Andreas Hülsing, and Markus Rückert. On the security of the Winternitz one-time signature scheme. In Abderrahmane Nitaj and David Pointcheval, editors, *AFRICACRYPT 11*, volume 6737 of *LNCS*, pages 363–378. Springer, Heidelberg, July 2011.

[BDH11] Johannes A. Buchmann, Erik Dahmen, and Andreas Hülsing. XMSS - A practical forward secure signature scheme based on minimal security assumptions. In Bo-Yin Yang, editor, *4th International Workshop on Post-Quantum Cryptography (PQCrypto) 2011*, volume 7071 of *LNCS*, pages 117–129. Springer, 2011.

[BGG94] Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. Incremental cryptography: The case of hashing and signing. In Yvo Desmedt, editor, *CRYPTO'94*, volume 839 of *LNCS*, pages 216–233. Springer, Heidelberg, August 1994.

[BKH13] Johannes Braun, Franziskus Kiefer, and Andreas Hülsing. Revocation and non-repudiation: When the first destroys the latter. In *10th European Workshop on Public Key Infrastructures, Services and Applications (EuroPKI) 2013*, volume 8341 of *LNCS*, pages 31–46. Springer, 2013.

[BM97] Mihir Bellare and Daniele Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In Walter Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 163–192. Springer, Heidelberg, May 1997.

[Com11] Comodo Group. Comodo fraud incident: Update 31-Mar-2011. https://www.comodo.com/Comodo-Fraud-Incident-2011-03-23.html, March 2011.

[Cro09] Scott A. Crosby. *Efficient Tamper-Evident Data Structures for Untrusted Servers*. PhD thesis, Rice University, Houston, Texas, USA, 2009.

[CW09] Scott A. Crosby and Dan S. Wallach. Efficient data structures for tamper-evident logging. In *18th USENIX Security Symposium 2009*, pages 317–334. USENIX Association, 2009.

[DOTV08] Erik Dahmen, Katsuyuki Okeya, Tsuyoshi Takagi, and Camille Vuillaume. Digital signatures out of second-preimage resistant hash functions. In Johannes A. Buchmann

and Jintai Ding, editors, *2nd International Workshop on Post-Quantum Cryptography (PQCrypto) 2008*, volume 5299 of *LNCS*, pages 109–123. Springer, 2008.

[Ele] Electronic Frontier Foundation. Sovereign Keys. https://www.eff.org/sovereign-keys.

[EPS15] C. Evans, C. Palmer, and R. Sleevi. Public Key Pinning Extension for HTTP. RFC 7469 (Proposed Standard), April 2015.

[Fox12] Fox IT. Black Tulip: Report of the investigation into the DigiNotar certificate authority breach. http://www.rijksoverheid.nl/bestanden/documenten-en-publicaties/rapporten/2012/08/13/black-tulip-update/black-tulip-update.pdf, August 2012.

[HS12] P. Hoffman and J. Schlyter. The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA. RFC 6698 (Proposed Standard), August 2012.

[Hua16] David Huang. Early impacts of Certificate Transparency. https://www.facebook.com/notes/protect-the-graph/early-impacts-of-certificate-transparency/1709731569266987/, April 2016.

[Ken15] S. Kent. Attack model and threat for Certificate Transparency. https://tools.ietf.org/html/draft-ietf-trans-threat-analysis-03, October 2015.

[KHP+13] Tiffany Hyun-Jin Kim, Lin-Shung Huang, Adrian Perrig, Collin Jackson, and Virgil D. Gligor. Accountable key infrastructure (AKI): a proposal for a public-key validation infrastructure. In *22nd International World Wide Web Conference (WWW) 2013*, pages 679–690. ACM, 2013.

[Lau14] Ben Laurie. Certificate Transparency. *ACM Queue - Security*, 12(8):10, 2014.

[LHKR06] Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin. Dynamic authenticated index structures for outsourced databases. In *ACM SIGMOD International Conference on Management of Data 2006*, pages 121–132. ACM, 2006.

[LK12] Ben Laurie and Emilia Kasper. Revocation Transparency. http://www.links.org/files/RevocationTransparency.pdf, 2012.

[LKMS04] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. Secure untrusted data repository (sundr). In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 9–9, Berkeley, CA, USA, 2004. USENIX Association.

[LLK13] B. Laurie, A. Langley, and E. Kasper. Certificate Transparency. RFC 6962 (Experimental), June 2013.

[Mau96] Ueli M. Maurer. Modelling a public-key infrastructure. In Elisa Bertino, Helmut Kurth, Giancarlo Martella, and Emilio Montolivo, editors, *ESORICS'96*, volume 1146 of *LNCS*, pages 325–350. Springer, Heidelberg, September 1996.

[MBB+15] Marcela S Melara, Aaron Blankstein, Joseph Bonneau, Edward W Felten, and Michael J Freedman. CONIKS: Bringing key transparency to end users. In *USENIX Security 2015*, pages 383–398. USENIX Association, 2015.

[Mer79] Ralph C. Merkle. Secrecy, authentication, and public key systems. Technical Report 1979-1, Information Systems Laboratory, Stanford University, June 1979.

[Mer90] Ralph C. Merkle. A certified digital signature. In Gilles Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 218–238. Springer, Heidelberg, August 1990.

[MS05] John Marchesini and Sean Smith. Modeling public key infrastructures in the real world. In *2nd European Workshop on Public Key Infrastructures, Services and Applications (EuroPKI) 2005*, volume 3545 of *LNCS*, pages 118–134. Springer, 2005.

[NGR15] L. Nordberg, D. Gillmor, and T. Ritter. Gossiping in CT. https://tools.ietf.org/html/draft-ietf-trans-gossip-00, August 2015.

[NN98] Kobbi Nissim and Moni Naor. Certificate revocation and certificate update. In *USENIX Security 1998*. USENIX Association, 1998.

[OHO05] Mizuhito Ogawa, Eiichi Horita, and Satoshi Ono. Proving properties of incremental Merkle trees. In *20th International Conference on Automated Deduction (CADE) 2005*, volume 3632 of *LNAI*, pages 424–440, 2005.

[Rya14] Mark D. Ryan. Enhanced certificate transparency and end-to-end encrypted mail. In *NDSS 2014*. The Internet Society, February 2014.

[SE15] S. Somogyi and A. Eijdenberg. Improved digital certificate security. http://googleonlinesecurity.blogspot.de/2015/09/improved-digital-certificate-security.html, September 2015.

[Szy04] Michael Szydlo. Merkle tree traversal in log space and time. In Christian Cachin and Jan Camenisch, editors, *EUROCRYPT 2004*, volume 3027 of *LNCS*, pages 541–554. Springer, Heidelberg, May 2004.

[Vil02] Jan Villemson. *Size-efficient interval time stamps*. PhD thesis, Tartu, 2002.

A Proofs of Merkle Tree Security Properties

We now provide proofs for the facts about Merkle tree hashing stated in Section 2.2.

Proof of Lemma 1. Suppose $\mathcal{A}$ outputs sets $\vec{E}_0$ and $\vec{E}_1$ such that $\texttt{MTH}_\text{H}(\vec{E}_0) = \texttt{MTH}_\text{H}(\vec{E}_1)$ but $\vec{E}_0 \neq \vec{E}_1$. $\mathcal{B}_1$ then runs $\mathcal{R}_1$ on $\vec{E}_0$, $\vec{E}_1$. $\mathcal{R}_1$ is a recursive algorithm which takes as input two lists that are distinct but hash to the same value under $\texttt{MTH}_\text{H}$. Somewhere in the recursive computation of $\texttt{MTH}_\text{H}(\vec{E}_0)$ and $\texttt{MTH}_\text{H}(\vec{E}_1)$, a collision occurs in H, and $\mathcal{R}_1$ recurses until it finds that collision. □

Proof of Lemma 2. Suppose $\mathcal{R}_2$ receives as input values $e, \vec{D}, \vec{A}, m$ such that

$$\texttt{RootFromPath}_\text{H}(e, n, \vec{A}, m) = \texttt{MTH}_\text{H}(\vec{D}) \tag{4}$$

(where $n = |\vec{D}|$) but

$$e \notin \vec{D} \ . \tag{5}$$

We claim that $\mathcal{R}_2$ outputs a collision for H when run with values satisfying (4) and (5). The proof of this claim proceeds by induction on n.

First suppose $n = 1$. Then $\texttt{MTH}_\text{H}(\vec{D}) = \text{H}(0\|\vec{D}[0])$ and $\texttt{RootFromPath}_\text{H}(e, 1, \vec{A}, m) = \text{H}(0\|e)$, and these are equal by assumption on $\mathcal{R}_2$'s inputs. Since $e \notin \vec{D}$, we have that $e \neq \vec{D}[0]$ and thus $0\|\vec{D}[0] \neq 0\|e$, but $\text{H}(0\|\vec{D}[0]) = \text{H}(0\|e)$, which is a collision for H. This is what is output by $\mathcal{R}_2$ on line 3.

$\underline{\mathcal{B}_1^{\mathcal{A}}()}$:
1: $(\vec{E}_0, \vec{E}_1) \leftarrow \mathcal{A}()$
2: *// assume* $\mathtt{MTH}_{\mathrm{H}}(\vec{E}_0) = \mathtt{MTH}_{\mathrm{H}}(\vec{E}_1)$ *but* $\vec{E}_0 \neq \vec{E}_1$
3: **return** $\mathcal{R}_1(\vec{E}_0, \vec{E}_1)$

$\underline{\mathcal{R}_1(\vec{D}_0, \vec{D}_1)}$:
1: *// require:* $\vec{D}_0 \neq \vec{D}_1$ *but* $\mathtt{MTH}_{\mathrm{H}}(\vec{D}_0) = \mathtt{MTH}_{\mathrm{H}}(\vec{D}_1)$
2: $n_0 \leftarrow |\vec{D}_0|$, $n_1 \leftarrow |\vec{D}_1|$
3: $k_0 \leftarrow 2^{\lceil \log_2(n_0)/2 \rceil}$, $k_1 \leftarrow 2^{\lceil \log_2(n_1)/2 \rceil}$
4: **if** $n_0 = n_1 = 1$, **return** $(0\|\vec{D}_0[0], 0\|\vec{D}_1[0])$
5: **elseif** $(n_0 = 1) \wedge (n_1 > 1)$
6: **return** $(0\|\vec{D}_0[0], 1\|\mathtt{MTH}_{\mathrm{H}}(\vec{D}_1[0 : k_1])\|\mathtt{MTH}_{\mathrm{H}}(\vec{D}_1[k_1 : n_1]))$.
7: **elseif** $(n_0 > 1) \wedge (n_1 = 1)$
8: **return** $(1\|\mathtt{MTH}_{\mathrm{H}}(\vec{D}_0[0 : k_0])\|\mathtt{MTH}_{\mathrm{H}}(\vec{D}_0[k_0 : n_0]), 0\|\vec{D}_1[0])$
9: **else** $(n_0, n_1 > 1)$
10: **if** $(\mathtt{MTH}_{\mathrm{H}}(\vec{D}_0[0 : k_0]) \neq \mathtt{MTH}_{\mathrm{H}}(\vec{D}_1[0 : k_1]))$
11: $\vee\ (\mathtt{MTH}_{\mathrm{H}}(\vec{D}_0[k_0 : n_0]) \neq \mathtt{MTH}_{\mathrm{H}}(\vec{D}_1[k_1 : n_1]))$
12: **return** $(1\|\mathtt{MTH}_{\mathrm{H}}(\vec{D}_0[0 : k_0])\|\mathtt{MTH}_{\mathrm{H}}(\vec{D}_0[k_0 : n_0]),$
13: $1\|\mathtt{MTH}_{\mathrm{H}}(\vec{D}_1[0 : k_1])\|\mathtt{MTH}_{\mathrm{H}}(\vec{D}_1[k_1 : n_1]))$
14: **elseif** $\vec{D}_0[0 : k_0] \neq \vec{D}_1[0 : k_1]$
15: **return** $\mathcal{R}_1(\vec{D}_0[0 : k_0],\ \vec{D}_1[0 : k_1])$
16: **elseif** $\vec{D}_0[k_0 : n_0] \neq \vec{D}_1[k_1 : n_1]$
17: **return** $\mathcal{R}_1(\vec{D}_0[k_0 : n_0],\ \vec{D}_1[k_1 : n_1])$

Figure 11: Algorithm $\mathcal{B}_1$ for Lemma 1.

Now suppose $n > 1$. Let $k = 2^{\lceil \log_2(n)/2 \rceil}$.

Suppose $m < k$, and let ℓ, r, ℓ', and r' be as on lines 5, 6, 8, and 9 of $\mathcal{R}_2$ in Figure 12. Then $\mathtt{MTH}_{\mathrm{H}}(\vec{D}) = \mathrm{H}(1\|\ell\|r)$ and $\mathtt{RootFromPath}_{\mathrm{H}}(e, n, \vec{A}, m) = \mathrm{H}(1\|\ell\|r)$, and these are equal by assumption on $\mathcal{R}_2$'s inputs. If $\ell \neq \ell'$ or $r \neq r'$, then we immediately have a collision for H: $(1\|\ell\|r, 1\|\ell'\|r')$, which is what $\mathcal{R}_2$ outputs on line 10. If these are all equal, then $e, \vec{D}[0 : k], \vec{A}[|\vec{A}| - 1], m$ satisfy (4) and (5). By induction, this recursive call to $\mathcal{R}_2$ outputs a collision.

Suppose $m \geq k$, and let ℓ, r, ℓ', and r' be as on lines 5, 6, 13, and 14 of $\mathcal{R}_2$ in Figure 12. Then $\mathtt{MTH}_{\mathrm{H}}(\vec{D}) = \mathrm{H}(1\|\ell\|r)$ and $\mathtt{RootFromPath}_{\mathrm{H}}(e, n, \vec{A}, m) = \mathrm{H}(1\|\ell\|r)$, and these are equal by assumption on $\mathcal{R}_2$'s inputs. If $\ell \neq \ell'$ or $r \neq r'$, then we immediately have a collision for H: $(1\|\ell\|r, 1\|\ell'\|r')$, which is what $\mathcal{R}_2$ outputs on line 15. If these are all equal, then $e, \vec{D}[k : n], \vec{A}[|\vec{A}| - 1], m$ satisfy (4) and (5). By induction, this recursive call to $\mathcal{R}_2$ outputs a collision. □

$\underline{\mathcal{B}_2^{\mathcal{A}}()}$:

1: $(e, \vec{E}, \vec{A}, m) \leftarrow \mathcal{A}()$
2: *// assume* $\texttt{RootFromPath}_{\mathrm{H}}(e, n, \vec{A}, m) = \texttt{MTH}_{\mathrm{H}}(\vec{E})$ *but* $e \notin \vec{E}$
3: **return** $\mathcal{R}_2(e, \vec{E}, \vec{A}, m)$

$\underline{\mathcal{R}_2(e, \vec{D}, \vec{A}, m)}$:

1: *// require:* $\texttt{RootFromPath}_{\mathrm{H}}(e, |\vec{D}|, \vec{A}, m) = \texttt{MTH}_{\mathrm{H}}(\vec{D})$
2: $n \leftarrow |\vec{D}|$
3: **if** $n = 1$, **return** $(0\|\vec{D}[0], 0\|e)$
4: $k \leftarrow 2^{\lceil \log_2(n)/2 \rceil}$
5: $\ell \leftarrow \texttt{MTH}_{\mathrm{H}}(\vec{D}[0:k])$
6: $r \leftarrow \texttt{MTH}_{\mathrm{H}}(\vec{D}[k:n])$
7: **if** $m < k$
8: $\quad \ell' \leftarrow \texttt{RootFromPath}_{\mathrm{H}}(e, k, \vec{A}[0 : |\vec{A}| - 1], m)$
9: $\quad r' \leftarrow A[|A| - 1]$
10: $\quad$ **if** $(\ell \neq \ell') \vee (r \neq r')$, **return** $(1\|\ell\|r, 1\|\ell'\|r')$
11: $\quad$ **else return** $\mathcal{R}_2(e, \vec{D}[0:k], \vec{A}[0 : |\vec{A}| - 1], m)$
12: **else** $(m \geq k)$
13: $\quad \ell' \leftarrow \vec{A}[|\vec{A}| - 1]$
14: $\quad r' \leftarrow \texttt{RootFromPath}(e, n - k, \vec{A}[0 : |\vec{A}| - 1], m - k)$
15: $\quad$ **if** $(\ell \neq \ell') \vee (r \neq r')$, **return** $(1\|\ell\|r, 1\|\ell'\|r')$
16: $\quad$ **else return** $\mathcal{R}_2(e, \vec{D}[k:n], \vec{A}[0 : |\vec{A}| - 1], m - k)$

Figure 12: Algorithm $\mathcal{B}_2$ for Lemma 2.