Secure Logging Schemes and Certificate Transparency

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

[1] Queensland University of Technology, Brisbane, Australia
[2] Technische Universität Darmstadt, Darmstadt, Germany
guenther@cs.tu-darmstadt.de
[3] McMaster University, Hamilton, ON, Canada

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).

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

I. Askoxylakis et al. (Eds.): ESORICS 2016, Part II, LNCS 9879, pp. 140–158, 2016.
DOI: 10.1007/978-3-319-45741-3_8

issue itself a valid certificate for the domain `google.com` and its subdomains from the prominent Dutch Certificate Authority DigiNotar [11]. 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 [5].

Certificate Transparency (CT) [17,18] 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 [30]. 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 [14]. 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.

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.

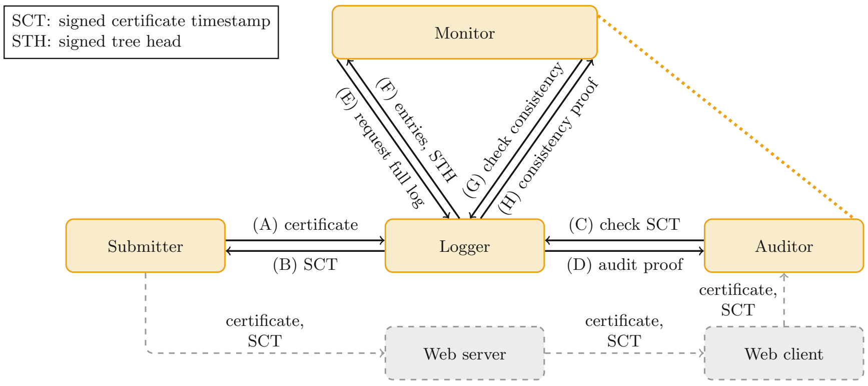

Fig. 1. Overview over the interaction between entities in Certificate Transparency; see Sect. 1.1 for details. Solid-line interactions and solid-line, orange entities are captured by the model in our work while dashed-line interactions and dashed-line, gray entities are not captured. Dotted line–connected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

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 [24,25], a form of a tamper-evident history tree [6,7]. 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 [27].

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

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 be implemented through SCT feedback (where web clients send SCTs through 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.

As a starting point for a threat model, the informational IETF draft “Attack Model for Certificate Transparency” [15] describes potential attack scenarios when Certificate Transparency is used in the context of web public-key infrastructure.

1.2 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 Sect. 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.

Security Definitions. Next, we introduce cryptographic security properties for logging schemes in Sect. 4 that are inspired by the CT threat model but reflect the corresponding 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 including a promised entry when it actually has?

Security of Certificate Transparency. Finally, we analyze the security of Certificate Transparency in Sect. 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 Sect. 3.3, we discuss the applicability of our definitions to CONIKS [23], 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.

1.3 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* [10] and *DANE* [13] are such proposals that allow domain owners to proactively and directly state their trusted public keys for the domain. Certificate Transparency takes a reactive 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 [6,7]. Two of their results [6] 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 [6, Sect. 3] are generally descriptive rather than algorithmic, whereas we state the operations fully algorithmically and provide complete algorithmic reductions for all proofs.

In recent years a few more approaches have emerged around the concept of *transparency logs*, including revocation [19,29] (which we omit in this work as they are not under consideration by the IETF Public Notary Transparency working group) or limitations on certificate issuance, validation, and update [1,16]. The Electronic Frontier Foundation's *Sovereign Keys Project* [9] combines transparency logs with cross-signing of keys. Melara et al. [23] present *CONIKS*, a system focusing on key transparency 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.

Merkle Trees. Introduced by Merkle [24], Merkle trees have been used in many areas of cryptography and computer science, including in the construction of public key signatures from hash functions [25]. 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.

There has been some work on authentication trees and more generally signatures on dynamic data sets. Bellare et al. [2,3] introduced the notion of *incremental cryptography.* Naor and Nissim [26] use dynamic Merkle trees in the context of certificate revocation and updates, Li et al. [20] apply them to authenticate index structures in outsourced databases. Villemson [31] and Ogawa et al. [28] investigated the characteristics of (generalizations of) incremental Merkle trees.

Cryptographic PKI Analyses. Maurer [22] introduced a formal model for public key infrastructures (PKIs) which subsequently was further extended [4,21]. 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

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., $\frac{n}{2} \leq k = 2^i < n$.

We rely on the standard notion of signature schemes and existential unforgeability under chosen-message attacks [12], and the corresponding advantage $\mathrm{Adv}^{\mathsf{euf\text{-}cma}}_{\mathtt{SIG}}(\mathcal{A})$ of an adversary $\mathcal{A}$ breaking this notion for a scheme $\mathtt{SIG}$.

Definition 1 (Hash Collision Finding). *Let* $\mathcal{M}$ *be a set, let* $\mathrm{H} : \mathcal{M} \rightarrow \{0,1\}^\lambda$ *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* $\mathrm{H}(m) = \mathrm{H}(m')$.

2.1 Merkle Trees

The use of hash trees for authenticating large amounts of data was first proposed by Merkle [24,25]. Let $\mathrm{H} : \{0,1\}^* \rightarrow \{0,1\}^\lambda$ be a hash function. In a Merkle hash tree for $\vec{E}$, the values of $\vec{E}$ 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 Fig. 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. The

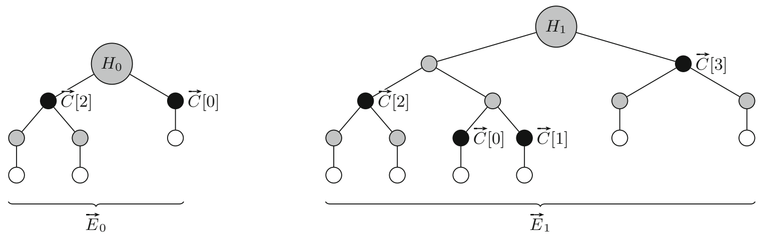

Fig. 2. Merkle tree consistency proof $\vec{C} = (\vec{C}[0], \ldots, \vec{C}[3])$ between roots H_0 (for a tree of size 3) and H_1 (for a tree of size 6). ○ denotes leaf nodes, ◉ denotes inner nodes, ● denotes nodes corresponding to consistency proof values.

authentication path generation algorithm $\texttt{Path}_{\mathrm{H}}(m, \vec{E})$ and verification algorithm $\texttt{CheckPath}_{\mathrm{H}}(e, H, n, \vec{A}, m)$ are shown in Fig. 3.

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 [6,7]. In Fig. 2, 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 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 Fig. 2). 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 Fig. 3. We have reformulated these from how they appear in the RFC [17]: 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.

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 [24,25]. For completeness, full proofs are given in the full version [8].

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 $\mathcal{B}_1^{\mathcal{A}}$ 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.*

Lemma 2 (Authentication Paths Consistency). *If* H *is collision-resistant, then no* $\texttt{CheckPath}_{\mathrm{H}}$ *authentication path $\vec{A}$ can be generated*

```
$\mathtt{MTH}_{\mathrm{H}}(\vec{E}) \to H$:
1: $n \leftarrow |\vec{E}|$
2: if $n = 1$, return $\mathrm{H}(0\|\vec{E}[0])$
3: else $(n > 1)$
4:     $k \leftarrow 2^{\lceil \log_2(n/2) \rceil}$
5:     return $\mathrm{H}(1\|\mathtt{MTH}_{\mathrm{H}}(\vec{E}[0:k])$
6:                 $\|\mathtt{MTH}_{\mathrm{H}}(\vec{E}[k:n])$)

$\mathtt{Path}_{\mathrm{H}}(m, \vec{E}) \to \vec{A}$:
1: $n \leftarrow |\vec{E}|$
2: if $n = 1$, return $()$
3: else $(n > 1)$
4:     $k \leftarrow 2^{\lceil \log_2(n/2) \rceil}$
5:     if $m < k$
6:         return $\mathtt{Path}_{\mathrm{H}}(m, \vec{E}[0:k])$
7:                 $\|\mathtt{MTH}_{\mathrm{H}}(\vec{E}[k:n])$
8:     else $(m \geq k)$
9:         return $\mathtt{Path}_{\mathrm{H}}(m-k, \vec{E}[k:n])$
10:                $\|\mathtt{MTH}_{\mathrm{H}}(\vec{E}[0:k])$

$\mathtt{CheckPath}_{\mathrm{H}}(e, H, n, \vec{A}, m) \to \{0,1\}$:
1: $H' \leftarrow \mathtt{RootFromPath}_{\mathrm{H}}(e, n, \vec{A}, m)$
2: return $(H = H')$

$\mathtt{RootFromPath}_{\mathrm{H}}(e, n, \vec{A}, m) \to H$:
1: if $n = 1$, return $\mathrm{H}(0\|e)$
2: $k \leftarrow 2^{\lceil \log_2(n/2) \rceil}$
3: if $m < k$
4:     $\ell \leftarrow \mathtt{RootFromPath}_{\mathrm{H}}(e, k, \vec{A}[0:|\vec{A}|-1], m)$
5:     $r \leftarrow \vec{A}[|\vec{A}|-1]$
6: else $(m \geq k)$
7:     $\ell \leftarrow \vec{A}[|\vec{A}|-1]$
8:     $r \leftarrow \mathtt{RootFromPath}_{\mathrm{H}}(e, n-k,$
9:                 $\vec{A}[0:|\vec{A}|-1], m-k)$
10: return $\mathrm{H}(1\|\ell\|r)$

$\mathtt{ConsProof}_{\mathrm{H}}(m, n, \vec{E}) \to \vec{C}$:
1: // require: $0 \leq m \leq n \leq |\vec{E}|$
2: if $m = n$
3:     return $()$
4: else $(m < n)$
5:     return $\mathtt{ConsProofSub}_{\mathrm{H}}(m, \vec{E}[0:n], \mathrm{true})$

$\mathtt{ConsProofSub}_{\mathrm{H}}(m, \vec{E}, b) \to \vec{C}$:
1: $n \leftarrow |\vec{E}|$
2: if $(m = n) \wedge (b = \mathrm{false})$
3:     return $\mathtt{MTH}_{\mathrm{H}}(\vec{E}[0:m])$
4: else
5:     $k \leftarrow 2^{\lceil \log_2(n)/2 \rceil}$
6:     if $m \leq k$
7:         return $\mathtt{ConsProofSub}_{\mathrm{H}}(m, \vec{E}[0:k], b)$
8:                 $\|\mathtt{MTH}_{\mathrm{H}}(\vec{E}[k:n])$
9:     else $(m > k)$
10:        return $\mathtt{ConsProofSub}_{\mathrm{H}}(m-k, \vec{E}[k:n], \mathrm{false})$
11:                $\|\mathtt{MTH}_{\mathrm{H}}(\vec{E}[0:k])$

$\mathtt{CheckConsProof}_{\mathrm{H}}(n_0, H_0, n_1, H_1, \vec{C}) \to b$:
1: if $n_0$ is a power of two, $\vec{C} \leftarrow H_0\|\vec{C}$
2: $H_0' \leftarrow \mathtt{Root0FromConsProof}_{\mathrm{H}}(\vec{C}, n_0, n_1)$
3: $H_1' \leftarrow \mathtt{Root1FromConsProof}_{\mathrm{H}}(\vec{C}, n_0, n_1)$
4: return $((H_0 = H_0') \wedge (H_1 = H_1'))$

$\mathtt{Root0FromConsProof}_{\mathrm{H}}(\vec{C}, n_0, n_1) \to H$:
1: $k \leftarrow 2^{\lceil \log_2(n_1)/2 \rceil}$
2: if $n_0 < k$
3:     return $\mathtt{Root0FromConsProof}_{\mathrm{H}}(\vec{C}[0:|\vec{C}|-1], n_0, k)$
4: elsif $n_0 = k$, return $\vec{C}[|\vec{C}|-2]$
5: else
6:     $\ell \leftarrow \vec{C}[|\vec{C}|-1]$
7:     $r \leftarrow \mathtt{Root0FromConsProof}_{\mathrm{H}}(\vec{C}[0:|\vec{C}|-1],$
8:                 $n_0-k, n_1-k)$
9:     return $\mathrm{H}(1\|\ell\|r)$

$\mathtt{Root1FromConsProof}_{\mathrm{H}}(\vec{C}, n_0, n_1) \to H$:
1: if $|\vec{C}| = 2$, return $\mathrm{H}(1\|\vec{C}[0]\|\vec{C}[1])$
2: $k \leftarrow 2^{\lceil \log_2(n_1)/2 \rceil}$
3: if $n_0 < k$
4:     $\ell \leftarrow \mathtt{Root1FromConsProof}_{\mathrm{H}}(\vec{C}[0:|\vec{C}|-1], n_0, k)$
5:     $r \leftarrow \vec{C}[|\vec{C}|-1]$
6: else
7:     $\ell \leftarrow \vec{C}[|\vec{C}|-1]$
8:     $r \leftarrow \mathtt{Root1FromConsProof}_{\mathrm{H}}(\vec{C}[0:|\vec{C}|-1],$
9:                 $n_0-k, n_1-k)$
10: return $\mathrm{H}(1\|\ell\|r)$
```

Fig. 3. Merkle tree algorithms

with respect to Merkle-tree hashing $\mathtt{MTH}_{\mathrm{H}}$ *for an entry* e *not contained in the Merkle tree. More precisely, if* $\mathcal{A}$ *outputs* $(e, \vec{E}, \vec{A}, m)$ *such that* $\mathtt{CheckPath}_{\mathrm{H}}(e, \mathtt{MTH}_{\mathrm{H}}(\vec{E}), |\vec{E}|, \vec{A}, m) = 1$ *and* $e \notin \vec{E}$*, then there exists algorithm* $\mathcal{B}_2^{\mathcal{A}}$ *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 2 (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$(st, F) \rightarrow \vec{E}$ *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) $\xrightarrow{\$}$ $\vec{M}$ *or* $\perp$: *A probabilistic algorithm*[2] *that takes as input a state* st*, a log entry* e*, and a fingerprint* F *and outputs a membership proof* $\vec{M}$*, or an error symbol* $\perp$.
- ProveConsistency(st, F_0, F_1) $\xrightarrow{\$}$ $\vec{C}$ *or* $\perp$: *A probabilistic algorithm*[2] *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$(P, pk) \rightarrow \{0, 1\}$: *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* $b \in \{0, 1\}$.

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

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

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

$\texttt{CT}_{\mathrm{H},\texttt{SIG}}$.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: $H \leftarrow \texttt{MTH}_{\mathrm{H}}(st.\vec{E})$
6: $\sigma \leftarrow \texttt{SIG}.\mathrm{Sign}_{sk}(t, n, H)$
7: return $F \leftarrow (t, n, H, \sigma)$

$\texttt{CT}_{\mathrm{H},\texttt{SIG}}$.PresentEntries$(st, F) \to \vec{E}$:
1: if CheckFingerprint$(F, pk) = 0$, return $\perp$
2: return $st.\vec{E}[0 : F.n]$

$\texttt{CT}_{\mathrm{H},\texttt{SIG}}$.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: $\vec{A} \leftarrow \texttt{Path}_{\mathrm{H}}(m, \vec{E}[0 : F.n])$
5: return $\vec{M} \leftarrow (\vec{A}, m)$

$\texttt{CT}_{\mathrm{H},\texttt{SIG}}$.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 $\vec{C} \leftarrow \texttt{ConsProof}_{\mathrm{H}}(F_0.n, F_1.n, st.\vec{E})$
```

Fig. 4. Certificate Transparency: algorithms run by loggers.

- 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$(F, e, \vec{M}, pk) \to \{0,1\}$: *A deterministic algorithm that takes as input a fingerprint F, an entry e, a membership proof $\vec{M}$, 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 $b \in \{0,1\}$.*

Correctness of a logging scheme is defined in the natural way and is omitted due to space constraints; see the full version [8].

3.2 Instantiation of Certificate Transparency as a Logging Scheme

Figures 4 and 5 formulate Certificate Transparency using H and SIG as a logging scheme $\texttt{CT}_{\mathrm{H},\texttt{SIG}}$ (i.e., following Definition 2). A log entry in CT is a chain of X.509 certificates: the certificate (or partially completed pre-certificate) itself, and each intermediate CA's certificate leading to the root CA's cert. We treat entries in our formalization of logging schemes as opaque bit strings: our fomulation hence omits any syntactical checks for the entries it manages; adding these checks is independent of the logging properties. The promise P is called a *signed certificate timestamp* (SCT). The fingerprint F is called the *signed tree head* (STH).

```
$\texttt{CT}_{\mathrm{H},\texttt{SIG}}$.CheckPromise$(P, pk) \to b$:
  1: return $\texttt{SIG}.\mathrm{Vfy}_{pk}(P.t \| P.e, P.\sigma)$

$\texttt{CT}_{\mathrm{H},\texttt{SIG}}$.CheckFingerprint$(F, pk) \to b$:
  1: return $\texttt{SIG}.\mathrm{Vfy}_{pk}(F.t \| F.n \| F.H, F.\sigma)$

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

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

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

Fig. 5. Certificate Transparency: algorithms run by monitors/auditors.

3.3 CONIKS as a Logging Scheme

CONIKS [23] 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, but in contrast to CT uses a Merkle *prefix* tree in which some attribute of an entry (e.g., the user's identity) determines its position. The tree root is computed both from present entries and placeholder values for empty subtrees, allowing efficient calculation over very large but mostly empty trees. It is signed and published by the logger 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, including the previous signed tree root. However, this does not enable consistency proofs as in CT: verification that a key that was present in STR_i is also present in STR_j requires fresh membership proof of that key's presence in STR_j. Two core security properties of CONIKS are *non-equivocation* (a provider cannot present diverging views) and *privacy-preserving consistency proofs* (privacy here meaning with respect to other entries' information).

CONIKS can be mapped onto the following notions in our definition of a logging scheme. The KeyGen algorithm is run by the logger. CONIKS has no separate notion of promise and log entry, combining PromiseEntry and UpdateLog. CheckFingerprint will verify a signed tree root similarly. Aiming at privacy, CONIKS 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.

In terms of security properties, none of ours directly map onto CONIKS' notions, primarily because of including 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 matches with a similar change from CheckEntries to CheckMembership, and ignoring maximum merge delays. Consistency of STRs in CONIKS is quite a bit different from our entry-cons property, as CONIKS' involves probabilistic spot-checks using membership proofs.

4 Security Goals

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.

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 Fig. 6, 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.

```
$\underline{\mathrm{Exp}^{\textsf{entry-coll}}_{\mathrm{LS}}(\mathcal{A})}$:
  1: $(\vec{E}_0, \vec{E}_1, F, pk) \xleftarrow{\$} \mathcal{A}()$
  2: return 1 iff $(\mathrm{CheckEntries}(\vec{E}_0, F, pk) = 1) \wedge (\mathrm{CheckEntries}(\vec{E}_1, F, pk) = 1) \wedge (\vec{E}_0 \neq \vec{E}_1)$
$\underline{\mathrm{Exp}^{\textsf{proof-coll}}_{\mathrm{LS}}(\mathcal{A})}$:
  1: $(e, \vec{E}, F, \vec{M}, pk) \xleftarrow{\$} \mathcal{A}()$
  2: return 1 iff $(\mathrm{CheckEntries}(\vec{E}, F, pk) = 1) \wedge (\mathrm{CheckMembership}(e, F, \vec{M}, pk) = 1) \wedge (e \notin \vec{E})$
$\underline{\mathrm{Exp}^{\textsf{entry-cons}}_{\mathrm{LS}}(\mathcal{A})}$:
  1: $(\vec{E}_0, \vec{E}_1, F_0, F_1, \vec{C}, pk) \xleftarrow{\$} \mathcal{A}()$
  2: return 1 iff $(\mathrm{CheckConsistency}(F_0, F_1, \vec{C}, pk) = 1) \wedge (\mathrm{CheckEntries}(\vec{E}_0, F_0, pk) = 1)$
                 $\wedge (\mathrm{CheckEntries}(\vec{E}_1, F_1, pk) = 1) \wedge (\vec{E}_0 \not\preceq \vec{E}_1)$
```

Fig. 6. Security properties of a logging scheme LS against a malicious logger.

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 an append-only log. The final security property in Fig. 6 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.

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 Fig. 7. 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

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

$\mathsf{OTick}()$:
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:     $F \xleftarrow{\$} \mathsf{OUpdateLog}(\vec{P})$
5:     $\vec{E}_{promised} \leftarrow \vec{E}_{promised} \setminus \vec{P}$
6:     return $(T, F)$
7: else return $T$

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

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

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

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

Fig. 7. Security properties of a logging scheme LS against a malicious monitor/auditor framing a log for failing to include a promised entry.

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.

Theorems 1 and 2 below connect rather immediately with the security properties of the underlying Merkle tree hash, so we omit the arguments due to space constraints; they appear in the full version [8]. Lemmas 1 and 2 then connect the Merkle tree hash properties to finding a collision in H, which is infeasible if H is collision-resistant. For Theorem 3 we also provide the proof in the full version due to space restrictions; the proof for Theorem 4 is given in Appendix A.

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* $\mathcal{A}$*.*

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* $\mathcal{A}$*.*

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 the full version [8] 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.*

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* $\mathcal{A}$*.*

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. Benjamin Dowling and Douglas Stebila are supported by Australian Research Council (ARC) Discovery Project grant DP130104304. Felix Günther is supported by the DFG as part of project S4 within the CRC 1119 CROSSING.

A Proof of Theorem 4 (Inclusion of promises)

Proof. By definition of OTick (cf. Fig. 7), the simulated honest logger will keep track of any promise P issued through $\mathsf{OPromiseEntry}$ and will include the P through $\mathsf{OUpdateLog}$ 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 $\mathsf{OPromiseEntry}$ or

$\mathsf{OUpdateLog}$, 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 $\mathsf{euf\text{-}cma}$ security of $\mathtt{SIG}$, which works as follows. First of all, $\mathcal{C}$ creates an initial state with empty list of entries. It then simulates experiment $\mathrm{Exp}^{\mathsf{promise\text{-}incl}}_{\mathsf{CT}_{\mathrm{H},\mathtt{SIG}},\mathsf{MMD}}$ for $\mathcal{A}$, providing the public key pk from its $\mathsf{euf\text{-}cma}$ game as input for $\mathcal{A}$. It furthermore uses its $\mathsf{euf\text{-}cma}$ signing oracle OSign when required to generate a signature in the simulations of the $\mathsf{OPromiseEntry}$ and $\mathsf{OUpdateLog}$ 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 $\mathsf{OPromiseEntry}$ and $\mathsf{OUpdateLog}$ (as we excluded the case of a Merkle-tree hash collision). Hence, in particular, the according value was not queried to the $\mathsf{euf\text{-}cma}$ 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. $\square$

References

1. Basin, D.A., Cremers, C.J.F., Kim, T.H.J., Perrig, A., Sasse, R., Szalachowski, P.: ARPKI: attack resilient public-key infrastructure. In: Ahn, G.J., Yung, M., Li, N. (eds.) ACM CCS 2014, pp. 382–393. ACM Press, November 2014
2. Bellare, M., Goldreich, O., Goldwasser, S.: Incremental cryptography: the case of hashing and signing. In: Desmedt, Y.G. (ed.) CRYPTO 1994. LNCS, vol. 839, pp. 216–233. Springer, Heidelberg (1994)
3. Bellare, M., Micciancio, D.: A new paradigm for collision-free hashing: incrementality at reduced cost. In: Fumy, W. (ed.) EUROCRYPT 1997. LNCS, vol. 1233, pp. 163–192. Springer, Heidelberg (1997)
4. Braun, J., Kiefer, F., Hülsing, A.: Revocation and non-repudiation: when the first destroys the latter. In: Katsikas, S., Agudo, I. (eds.) EuroPKI 2013. LNCS, vol. 8341, pp. 31–46. Springer, Heidelberg (2014)
5. Comodo Group: Comodo fraud incident, 31 Mar 2011. https://www.comodo.com/Comodo-Fraud-Incident-2011-03-23.html
6. Crosby, S.A.: Efficient Tamper-Evident Data Structures for Untrusted Servers. Ph.D. thesis, Rice University, Houston, Texas, USA (2009)
7. Crosby, S.A., Wallach, D.S.: Efficient data structures for tamper-evident logging. In: 18th USENIX Security Symposium 2009, pp. 317–334. USENIX Association (2009). http://www.usenix.org/events/sec09/tech/full_papers/crosby.pdf
8. Dowling, B., Günther, F., Herath, U., Stebila, D.: Secure logging schemes and Certificate Transparency (full version). Cryptology ePrint Archive, Report 2016/452 (2016). http://eprint.iacr.org/2016/452
9. Electronic Frontier Foundation: Sovereign Keys. https://www.eff.org/sovereign-keys

10. Evans, C., Palmer, C., Sleevi, R.: Public Key Pinning Extension for HTTP. RFC 7469 (Proposed Standard), April 2015. http://www.ietf.org/rfc/rfc7469.txt
11. Fox, I.T.: Black Tulip: Report of the investigation into the DigiNotar certificate authority breach, August 2012. http://www.rijksoverheid.nl/bestanden/documenten-en-publicaties/rapporten/2012/08/13/black-tulip-update/black-tulip-update.pdf
12. Goldwasser, S., Micali, S., Rivest, R.L.: A digital signature scheme secure against adaptive chosen-message attacks. SIAM J. Comput. **17**(2), 281–308 (1988)
13. Hoffman, P., Schlyter, J.: The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA. RFC 6698 (Proposed Standard), August 2012. http://www.ietf.org/rfc/rfc6698.txt
14. Huang, D.: Early impacts of Certificate Transparency, April 2016. https://www.facebook.com/notes/protect-the-graph/early-impacts-of-certificate-transparency/1709731569266987/
15. Kent, S.: Attack model and threat for Certificate Transparency, October 2015. https://tools.ietf.org/html/draft-ietf-trans-threat-analysis-03
16. Kim, T.H., Huang, L., Perrig, A., Jackson, C., Gligor, V.D.: Accountable key infrastructure (AKI): a proposal for a public-key validation infrastructure. In: 22nd International World Wide Web Conference (WWW) 2013, pp. 679–690. ACM (2013)
17. Laurie, B., Langley, A., Kasper, E.: Certificate Transparency. RFC 6962 (Experimental), June 2013. http://www.ietf.org/rfc/rfc6962.txt
18. Laurie, B.: Certificate transparency. ACM Queue Secur. **12**(8), 10 (2014)
19. Laurie, B., Kasper, E.: Revocation Transparency (2012). http://www.links.org/files/RevocationTransparency.pdf
20. Li, F., Hadjieleftheriou, M., Kollios, G., Reyzin, L.: Dynamic authenticated index structures for outsourced databases. In: ACM SIGMOD International Conference on Management of Data 2006, pp. 121–132. ACM (2006)
21. Marchesini, J.C., Smith, S.: Modeling public key infrastructures in the real world. In: Chadwick, D., Zhao, G. (eds.) EuroPKI 2005. LNCS, vol. 3545, pp. 118–134. Springer, Heidelberg (2005)
22. Maurer, U.M.: Modelling a public-key infrastructure. In: Bertino, E., Kurth, H., Martella, G., Montolivo, E. (eds.) Computer Security – ESORICS '96. LNCS, vol. 1146, pp. 325–350. Springer, Heidelberg (1996)
23. Melara, M.S., Blankstein, A., Bonneau, J., Felten, E.W., Freedman, M.J.: CONIKS: bringing key transparency to end users. In: USENIX Security 2015, pp. 383–398. USENIX Association (2015)
24. Merkle, R.C.: Secrecy, authentication, and public key systems. Technical report 1979–1, Information Systems Laboratory, Stanford University, June 1979
25. Merkle, R.C.: A certified digital signature. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 218–238. Springer, Heidelberg (1990)
26. Nissim, K., Naor, M.: Certificate revocation and certificate update. In: USENIX Security 1998. USENIX Association (1998)
27. Nordberg, L., Gillmor, D., Ritter, T.: Gossiping in CT, August 2015. https://tools.ietf.org/html/draft-ietf-trans-gossip-00
28. Ogawa, M., Horita, E., Ono, S.: Proving properties of incremental merkle trees. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 424–440. Springer, Heidelberg (2005)
29. Ryan, M.D.: Enhanced certificate transparency and end-to-end encrypted mail. In: NDSS 2014, The Internet Society, February 2014

30. Somogyi, S., Eijdenberg, A.: Improved digital certificate security, September 2015. http://googleonlinesecurity.blogspot.de/2015/09/improved-digital-certificate-security.html
31. Villemson, J.: Size-efficient interval time stamps. Ph.D. thesis, Tartu (2002)