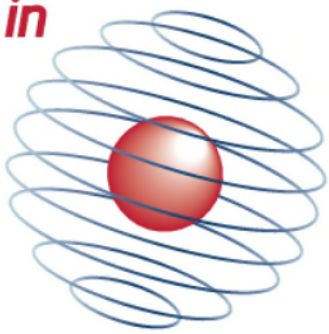
CENTRE *for* DOCTORAL TRAINING *in*
CYBER SECURITY



CDT Technical Paper

04/14

Security of Certificate Transparency

Katriel Cohn-Gordon

Security of Certificate Transparency

Katriel Cohn-Gordon

Supervised by Dr Kasper Rasmussen

Abstract

Certificate Transparency is one of a number of recent proposals to improve the public key infrastructure of the Internet, all based on the use of public, verifiable log servers to store records of certain actions. Whilst it lacks some features of alternative systems, such as handling revocation or permitting distributed verification, Certificate Transparency has the significant advantage of support from the Chromium web browser team, and thus will be enabled for large parts of the Internet by early 2015.

In this report we present an initial, informal security analysis of Certificate Transparency, identifying the implicit assumptions made elsewhere and describing the adversaries which is it designed to resist as well as those which it is not. We also suggest how this analysis could be formalised in future work, linking it to recent research on PKI in Bellare-Rogaway-style security games.

Contents

1 Introduction **2**
- 1.1 Background . . . 2
- 1.2 Related Work . . . 3

2 Adversary Model **4**
- 2.1 Reputation . . . 4
- 2.2 Bellare-Rogaway games . . . 5
- 2.3 Other Concerns . . . 5

3 Core Security Analysis **6**
- 3.1 Consistency of log servers . . . 6
- 3.2 Registration . . . 8
- 3.3 TLS handshake . . . 8
- 3.4 Verification . . . 9

4 The Gossip Protocol **9**

5 Conclusion **10**

A Examples **12**
- A.1 Compromise . . . 12

B Security Games for CDSs **14**

1 Introduction

The Internet today requires secure communication for myriad purposes. To achieve the required security properties, modern secure protocols universally rely on public-key cryptography, and hence their security depends on the reliability of the public keys they use: the public key infrastructure, or PKI. Today's PKI is based on an opaque system of chains of trust, whereby *web PKI certification authorities* (CAs) bind domains to public keys with signed certificates, and prove their own reliability with certificates from higher authorities until a "root of trust" is reached.

This approach, for all intents and purposes, generally works. However, because there is no way to detect the issuance of a certificate, CAs have a great deal of power: they may bind any domain to any public key, with all users accepting the binding transparently. There is no systematic way for the legitimate owner of a domain to list all issued certificates, nor to detect when a CA has issued a certificate which they should not have. This is not merely a theoretical problem: multiple CAs have in the past misissued[1] certificates, often detected only months later by specialised systems. For example, nothing today prevents a government from secretly compelling a CA to issue fake certificates for certain domains, which will allow silent traffic interception unless a client is using additional PKI methods.

Certificate Transparency (CT), proposed by Langley, Kasper and Laurie [LKL13, RFC 6962], is an incremental improvement to the current system in which web PKI certificates are irrevocably added to public logs upon issue, and proof of this inclusion is distributed along with the certificate. This enables interested parties, in particular domain owners, to check precisely which certificates have been issued for their domains and to ensure that these certificates abide by all applicable regulations. Dually, clients may demand proof that a certificate has been logged publicly before they are willing to use its key.

[1] For example, DigiNotar [Pri11] and the Indian CCA [Ada14] both misissued certificates for Google domains, which were only detected by certificate pinning in Chrome.

Goals The primary goal of CT is to remove the single point of failure caused by today's PKI, such that an adversary who compromises a single CA can intercept SSL traffic for any domain and an unknown period of time. Thus we primarily consider an adversary with total network control but who may corrupt at most one type of party. In fact, under certain circumstances CT can resist collusion between different types of node. We shall note the assumptions necessary for this resistance and the implementation details necessary for them to hold.

Moreover, unlike some other revocation request-type systems in which clients must give up some level of privacy or make additional requests to third parties in order to verify their received certificates, CT operates on a "herd immunity" [Goo14] basis, so that the security benefits are gained as long as some actors audit the public logs, even if any particular individual decides not to do so.

CT aims to be practical and immediately implementable, in contrast to "ground-up" rewritings of the PKI system. This means that it respects economic and practical design constraints, and chooses feasibility over security where necessary.

It does not claim to *prevent* misissue of certificates in general. Instead, it promises that malicious actions will become publicly visible and detectable. That is, if ordinary TLS is the lock on a door; CT is a burglar alarm: it does not prevent the door from being opened, but instead alerts any interested party if ever it is.

1.1 Background

> *This section briefly summarises the architecture of CT. For more details and full definitions, see [Goo14] or [LKL13].*

In the current system, domain owners request certificates from CAs, and send these certificates to clients over TLS. Under CT, compliant clients will in addition demand proof that the certificates have been submitted to a public log server, in the form of an additional signature on the certificate.

These public log servers receive certificates from any interested party, and append them to their log, periodically publishing a signed hash of the entire log known as a *signed tree head* (STH). The data structure they use, a Merkle tree, enables them to produce cryptographic proofs that STHs represent increasing logs if and only if they in fact do. The precise requirements for the data structure have been extracted by Yu, Cheval and Ryan [YCR14], and we shall see them later.

To issue a new certificate, CAs act as before, but in addition must submit their certificates to a number of logs, receiving in return inclusion promises known as *signed certificate timestamps* (SCTs). These are attached to the certificate, which is then used as normal.

When clients receive a CT-enabled certificate, they verify the signature of the CA on the entire certificate as usual, but also check the signature of the public log server on the SCT. In addition, if they so choose they may request from the log server a proof that the SCT was truly included in the log. These are of a similar form to the aforementioned proofs that STHs represent increasing logs.

To ensure that the log servers do not misbehave, a network of *monitors* maintains their own copies of each log, periodically polling for new updates and verifying consistency. These monitors may also receive STHs from the wider network, and check that these views of the log servers are consistent with their own.

To illustrate the mechanism, we describe in Appendix A the end-to-end process of issuing a certificate and using it to authenticate a TLS connection in CT. For simplicity, we use a single log server; in practise, the specification requires a small number greater than one, for resilience in case any single log server is compromised. The multiple-log protocol works identically, *mutatis mutandis*.

SCTs The Merkle tree data structure allows for proofs of presence: by providing certain hash preimages, anybody can be convinced that a certain value was included in the tree represented

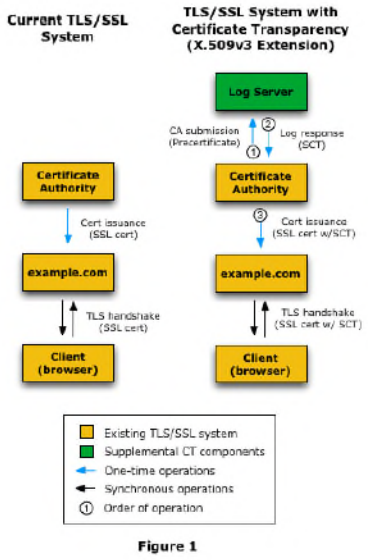


Figure 1: CT architecture [Goo14]

by a certain root hash. Theoretically, these could be transmitted with the certificates as the SCT, allowing clients to verify the inclusion offline. CT, however, specifies that log servers should not do this, instead creating a new datum which simply encodes a promise to include the given certificate in a log in a certain timeframe. There are two main reasons for this design.

Firstly, the inclusion guarantee must be verified as well as the certificate for CT to function. This means that it must be transmitted over all TLS handshakes, and hence that its size should be kept as small as possible. Although audit proofs are relatively (and asymptotically) small, they are still significantly larger than SCTs.

Secondly, log servers will only publish new logs periodically. If CAs had to wait for log servers to refresh their log for each new certificate issued, the additional time would add significant delays to the CT issuance process.

1.2 Related Work

Sovereign Keys Sovereign Keys is a related proposal to use append-only logs, this time of "sovereign" keys for a domain which must be used somehow in order to authenticate the TLS

connection. It requires trust in the mirror of the log server.

AKI Accountable Key Infrastructure [Kim+13, AKI] is a similar proposal to improve the security of the current PKI by adding public logs in which certificates must be registered. Unlike CT, however, AKI logs are ordered lexicographically by domain. This means that checking revocation is as efficient as checking membership. In contrast, CT's chronological tree is append-only, meaning that to check whether a certificate has been revoked requires reading the entire tree.

The downside to the append-only system is that it is no longer feasible for clients to verify the behaviour of other parties themselves; instead, they must rely on trusted validators to do so.

DTKI Distributed Transparent Key Infrastructure [CRY14] proposes a combination of CT and AKI, whereby both a lexical and a chronological tree are maintained ("stapled" together by including hashes from one in the other), in such a way that clients gain the benefits of CT but can also handle revocations. Unfortunately, the method by which the trees are stapled is not verifiable in logarithmic time, so that clients again rely on validators to do this work for them.

Observatories Various systems[2] use a different concept: distribute trust to a number of "notary" servers, and have these notaries repeatedly poll the certificate of a website. These solutions do not offer verifiable proofs of cheating by adversaries, but are also easier to implement currently.

2 Adversary Model

The original proposals for CT specify adversaries only very sparingly, identifying certain behaviours which it is designed to resist. There has been a significant quantity of online discussion of the threat model, but little consensus output. Roughly, CT is designed to withstand an adversary with significant – but not absolute – network control, who may compromise one or two distinct types of entities. A minimal goal is to detect any certificate misissue[3] if only a single type of entity is compromised.

However, the different subprotocols of CT have very different threat models, and require different analyses. In particular, there is a clear separation between the "core" key agreement and negotiation protocols, and the *ex post facto* dissemination and monitoring of behaviour: the former provide guarantees against general Dolev-Yau-style adversaries, while the latter require heuristic arguments on the topology of the network and the precise interference capabilities of the adversary. We discuss the latter separately in section 4, though it is equally security-critical.

For the former, the basic key exchange is unmodified from the original TLS specification, and we rely on its security results. In addition, the data structures used are asserted to provide non-repudiability, in the sense that the artefacts for a particular certificate verifiably link it to the original misbehaving party. To simplify the problem, we consider the security only of the authenticated key exchange (AKE) fragment of TLS, in which the two parties set up a session key under which data will be transferred with a symmetric encryption primitive.

2.1 Reputation

The adversaries treated in CT, unlike general AKE adversaries, have an additional constraint: they wish to maintain their public reputation. By this we mean that though they may act arbitrarily in secret, they do not wish to be detected misbehaving in a verifiable manner.

[2] Perspectives [WAP08], Convergence (`convergence.io`) and the EFF SSL Observatory (`https://www.eff.org/observatory`) included

[3] The precise definition of "misissue" is left up to the end users who are monitoring the logs: some might only care if a certificate is issued for the wrong public key, while some might consider a certificate misissued if it does not comply with the Baseline Regulations.

This distinction is important, because it significantly reduces the class of attacks which CT attempts to resist. For example, even after full deployment, a CA can always misissue a certificate, send it to a public log, and subsequently use it to intercept TLS connections. Indeed, unless anybody is watching issued certificates on that domain, this misbehaviour will not even be detected; certainly, it will take a nontrivial time to request the revocation of the malicious certificate and prove the original owner's identity.

We might call such adversaries "(nearly) omnipotent but proud": they have the power to perform many malicious actions, but do not wish to risk their reputation by doing so.

Instead of placing constraints on the actions of the adversary, we shall model the desire to maintain reputation in the lemmas we claim, asserting that if a client accepts a protocol execution as valid, then either it truly was or the client is left with an artefact proving misbehaviour.

2.2 Bellare-Rogaway games

Modulo the caveats above, we use a standard setup for security definitions of AKE protocols, based on the game-based models of Bellare and Rogaway [BR94]. In these models, a probabilistic polynomial-time Turing machine $\mathcal{A}$ known as the adversary controls all communication between honest parties, implemented as oracles which $\mathcal{A}$ may query. In addition to local computation, $\mathcal{A}$ has access to a handful of special commands, or queries, which model various parts of the network infrastructure.

For example, $\mathcal{A}$ may invoke the $\texttt{send}(m, s, P)$ query to send a message m to a session s at some party P and return her response, or the $\texttt{corrupt}(P)$ or $\texttt{ephemeral-key}(s)$ queries to reveal respectively the long- or short-term secret keys of party P and session s.

The adversary plays a game where she attempts to distinguish a session key from random data. To do so, she has access to a special query $\texttt{test-session}(s)$, which she may invoke only once and only on a session satisfying certain constraints. `test-session` secretly flips a coin $b \in \{0, 1\}$; if $b = 0$ it returns the session key of s and if $b = 1$ it returns a freshly-generated session key. At some later point, $\mathcal{A}$ must output a bit b' and she wins iff $b = b'$.

Using this definition of security, we may now say what we mean by a secure key exchange protocol π: it is a protocol which is (a) sound, in that if two parties complete matching sessions then they compute the same key; (b) unique, in that the probability of a session having multiple matching counterparts is negligible; and (c) secure, in that no adversary has a nontrivial chance of winning the security game .

Adding PKI In the model described above, all parties have access to a $\texttt{public-info}()$ query which returns all generated public keys; this models the entire PKI system. Recent work by Boyd et al. [Boy+13] modifies this model to include registration of certificates explicitly. For simplicity they use only a single CA, though the definitions for multiple CAs are essentially identical.

To add CAs to the model, Boyd et al. grant $\mathcal{A}$ access to additional queries allowing her to generate and register new public keys, or even register arbitrary strings of data as public keys. As with the basic protocol, this certainly allows the adversary to break any protocol (by registering a new public key); security now means that this is the only way to learn a session key.

2.3 Other Concerns

We discuss here a slightly idealised model of CT. In practise, there are some more subtleties related to the implementation which are relevant to security.

Incremental deployment Our analyses all assume that clients require servers to use CT, and that a certificate without its corresponding SCT is considered invalid (just as a certificate with an incorrect signature would be, for instance). In practise, new systems for PKI are only as useful as they are practical, and one criterion here is that they must be deployable incrementally,

working in parallel with the current infrastructure.

Permitting servers to ignore CT and fall back to standard TLS handshakes opens the door to *downgrade attacks*, whereby an adversary intercepts traffic between a CT-enabled client and server and modifies it to indicate that one endpoint does not understand CT. For example, an adversary with network control in possession of a valid certificate C_{fake} for `google.com` could act in the middle of a TLS handshake, sending C_{fake} to the client in place of the true CT-enabled certificate.

2.3.1 Privacy

The original CT proposal instructs clients to contact log servers to request audit proofs, either between STHs or for SCTs. These requests, particularly but not exclusively the latter, reveal information about the browsing habits of the clients to the log servers. In practise, the latter will be ameliorated by permitting DNS servers – who already receive full information about browsing activities – to proxy the SCT requests and inclusion proofs.

Again, the adversaries for the gossip protocol have different goals, for instance to gain broad rather than deep information about large-scale behaviour.

3 Core Security Analysis

CT comprises a number of different protocols: *registration* between a CA (though anybody may register a certificate) and a log server, in which the CA submits the certificate to the log and receives an SCT in return; the *extended TLS handshake* between a TLS server and client, exchanging a standard certificate together with a number of SCTs and perhaps STHs; *audit* between an auditor and a log server, in which the auditor submits either two STHs or an SCT and an STH, and receives a proof that the two items are consistent; and *gossip* between all parties, to which we turn in section 4.

CT is a complex ecosystem, and it does not suit a single formal security model. Instead, we aim to identify a small core of properties which we may model formally, under certain identified assumptions, and subsequently argue that the implementation indeed satisfies these assumptions.

3.1 Consistency of log servers

A compromised entity may act arbitrarily, sending whichever messages it wishes and maintaining whichever data structures it chooses. In particular, a malicious log server need not maintain the legitimate Merkle tree of certificates: it might generate multiple trees, or revert to previous ones, or even decide not to maintain a tree at all. We shall model this by granting the adversary access to the secret keys of compromised entities, at which point she may compute arbitrarily, perhaps using the same implementation as the honest entity but perhaps not.

CT requires that honest log servers act consistently; that is, that they must respond to queries as if they are backed by a unique append-only Merkle tree of certificate hashes. Yu, Cheval and Ryan [YCR14] formalise the requirements upon such data structures.

Definition 3.1. [YCR14, based on Definition 1] A *chronological data structure* (CDS) is a data structure S admitting the operations $\texttt{contents}(S)$ (returning a sequence of values), $\texttt{digest}(S)$ (returning a constant-size value called the "digest" of S), and $\texttt{add}(S, d)$ (creating a new chronological data structure S'); such that (i) if $\texttt{digest}(S) = \texttt{digest}(S')$ then $\texttt{contents}(S) = \texttt{contents}(S')$, and (ii) $\texttt{contents}(\texttt{add}(S, x)) = \texttt{contents}(S)$ concatenated with x . For simplicity of analysis, we generally define $\texttt{add}(S, d)$ to return the digest of the subsequent structure, so that we may omit the `digest` query in the security analysis.

For CT, we require that our CDSs can generate remotely-verifiable proofs of containment and of consistency, even in the presence of a malicious adversary. We also require digests to be authen-

ticated, so that the adversary cannot produce a digest without the cooperation of the CDS.

Definition 3.2. A CDS is *verifiable* if it admits additional operations `presence`(S, s) (returning a log-size value called the "proof of presence of s in S"), and `extension`(S, d, S', d') (returning a log-size value called the "proof of extension between (S, d) and (S', d')" such that no adversary has non-negligible advantage in the games ForgePresence, ForgeExtension and ForgeDigest defined in Appendix B.

Merkle trees present a potential implementation for verifiable CDSs, relying on the collision-resistance of a hash function to prevent forgery of proofs.

Conjecture 3.3 (Appendix B)**.** *Merkle hash trees using a collision-resistant hash function are verifiable chronological data structures, with proofs of presence and of extension defined as the sequence of values required to compute a path of hashes through the tree.*

CT log servers are then entities which implement a verifiable CDS, with digests playing the role of STHs. Without access to the private key of the log server, the conjecture above would suffice to show that the adversary cannot convince a client to accept a certificate, since one part of the certificate is a valid digest which we assumed unforgeable.

Of course, we must also consider the case where $\mathcal{A}$ gains access to the private key of a log server. In this case, the security of the CDS is no longer relevant: armed with the private key, an adversary can add any certificate or produce any digest. However, even so prepared, $\mathcal{A}$ can still not produce hash collisions or forge digests: she may only produce ones which appear legitimate. This property of non-repudiability grounds much of the security of CT.

We shall argue resilience against log server compromise in two parts: first by restricting the adversary in which actions she may perform with a compromised key, and second by arguing that it is possible to detect violations of this restriction.

We say a log server is *consistent* if every `latest` query returns one of (a) a fixed "initial" digest, (b) the same digest as returned by the previous query, or (c) a new digest b admitting a valid `extension` proof $\pi(a, b)$ of length 1.

In other words, the sequence of digests issued (to all clients combined) by a consistent log server should be of the form

$$\overbrace{s_0\ s_0 \dots s_0}^{n_0}\ \overbrace{s_1\ s_1 \dots s_1}^{n_1}\ \dots\ \overbrace{s_k\ s_k \dots s_k}^{n_k},$$

where $n_j \in \mathbb{N}$ and there exists for each $0 \leq i < k$ a `presence` proof between s_i and s_{i+1}.

We deduce from the consistency of a log server that it is effectively maintaining a verifiable CDS of something, though the entities which it stores may not be public.

Conjecture 3.4. *A consistent log server responds to all queries as if it were maintaining some chronological data structure; that is, for any consistent log server $\mathcal{L}$ there is an increasing sequence of hashes h_i such that the responses of $\mathcal{L}$ are identical to those of an honest log server storing data with hashes h_i.*

Proof idea: From consistency we get a sequence of values which comprise the publicly-visible tree, defined by the sequence of digests. This does not suffice, though, because a consistent log server might still add "extra" data to the tree. These are revealed by the extension proofs between adjacent digests, since such proofs must contain all hashes required to reconstruct the sequence.

Note that we cannot necessarily efficiently reconstruct all the values stored in the constructed CDS, only their hashes. ■

We may therefore refer to "the" log of a consistent log server, even if it is malicious.

Adversarial constraint CT may not be able to generate a proof that a log server has acted inconsistently: if there are two STHs a and b corresponding to inconsistent Merkle trees, the log server cannot generate a valid proof in response to the query `audit`(a, b). Honest clients thus cannot distinguish between a malicious log server

unable to prove consistency between a and b and an honest log server whose network traffic is being blocked by $\mathcal{A}$.

In the implementation, we solve this problem by requiring that log servers respond to all monitor queries within a short time period. (We cannot require this for clients, since in general clients may not have network connectivity to the log server.) In general, we say that artefacts which allow an honest monitor definitively to conclude that a log server is malicious comprise *proof of misbehaviour*. In this language, the aim of CT is to generate (and disseminate via the gossip protocol) proofs of misbehaviour where necessary.

This yields a constraint on adversaries in CT: we will assume that they do not wish to act in such a way as to leave proof of misbehaviour anywhere in the network. In practise, this must be backed up by a robust detection strategy so that these proofs are transmitted promptly to monitors who can deal with them. Note also that this constraint implies that all log servers must act consistently, since if an inconsistent digest is used in a client session then that digest, being unforgeable, comprises proof of misbehaviour. This is a problem for the gossip protocol of section 4.

We turn now to examine each sub-protocol of CT in turn.

3.2 Registration

Certificate registration has does not require confidentiality or authenticity, so there is little to prove: the submitter can verify the signature and the hash in the SCT and be confident that the log server has truly received her certificate. Of course, there is no guarantee that the certificate has been added to the log server's log; we must show that failing to do so will create proof of misbehaviour.

Lemma 3.5. *If a client CA invokes $\mathsf{add}(c)$ at a consistent log server $\mathcal{L}$ and receives a signed audit proof (which she need not verify), but $\mathcal{L}$ does not then add c to her log, then the proof is proof of misbehaviour.*

Proof: To complete the registration protocol, C must receive a proof corresponding to her certificate, with (a) the correct hash of her certificate, and (b) a valid signature of $\mathcal{L}$ on the hash. But the unforgeability of the signature means that a monitor can also verify it, and thus it comprises proof of misbehaviour. ■

Corollary 3.6. *SCTs for certificates that were not added to the log in time comprise proof of misbehaviour, for the same reason.*

3.3 TLS handshake

Because this protocol is implemented as an extension to TLS, we do not consider here basic protocol security notions such as authenticity; these are inferred from the equivalent properties of unextended TLS. Our first lemma is that absent a compromised log server, if an honest, CT-enabled client completes a TLS handshake then the certificate used is publicly visible.

Conjecture 3.7. *If $\mathcal{L}$ is honest and a client C accepts a sesson with certificate logged by $\mathcal{L}$, then the certificate is publicly visible in $\mathcal{L}$'s log.*

Proof idea: CT specifies that C will only accept a certificate c if accompanied by an SCT $s(c)$ signed by L; moreover, we know that L will only issue an SCT $s(c)$ if it publishes c in its log. We know that C's acceptance of $s(c)$ implies that the signature verifies, so it follows that L, acting honestly, must publish c in its log. ■

Corollary 3.8. *If the adversary does not control the server clocks, completing a TLS handshake with CT enabled implies that the certificate used will be publicly visible within a short period of time.*

Of course, we might face adversaries who can compromise the log server. We consider this case in two stages: first to show that it will produce proof of misbehaviour at the client, and second to discuss how this proof of misbehaviour can be disseminated under some limitations on adversarial power.

Consider the scenario above but where $\mathcal{A}$ may also compromise the log server without violating consistency. That is, $\mathcal{A}$ may perform arbitrary operations with $PK(L)$, including or refusing submitted certificates, but the sequence of her responses to `latest` queries must be consistent.

Conjecture 3.9. *If C completes a TLS handshake with S in the presence of $\mathcal{A}$ at time t_0, receiving an SCT $s = \{SCT, t, h\}$ then either (a) c is included in the tree represented by τ_h, or (b) s is proof of misbehaviour, where τ_h is any tree head returned by the `latest` query at some time $t > t_0 + MMD$.*

Proof idea: As before, for C to accept the TLS connection L must have issued an SCT $s = s(c)$. L has a unique sequence of trees because it is consistent; let τ_h be some sufficiently-late one such. Either τ_h represents a tree containing c or it does not.

If it does, then case (a) holds. If it does not, then it is impossible to generate a valid `SCTProof` between $s(c)$ and τ_h, so that τ_h is proof of misbehaviour and case (b) holds. ■

3.4 Verification

Conjecture 3.10. *In the same security model as above, if C receives an STH from a consistent log server L and subsequently receives an audit proof for an SCT with timestamp t, then c is contained in all logs of L from time $t + MMD$.*

This should follow in a relatively straightforward fashion from the conjectures of Appendix B, using the fact that $\mathcal{A}$ cannot falsify audit proofs.

4 The Gossip Protocol

We saw above that under certain assumptions, CT promises to provide proof of misbehaviour; that is, an artefact which proves to honest monitor nodes that a log server has been compromised. This is only part of the problem, however: if the adversary can prevent these proofs from ever reaching a monitor node than she can cheat with impunity.

Indeed, under the naive protocol in which clients simply send their proofs of misbehaviour directly to monitor nodes, this attack is both simple and practical: the adversary simply blocks all traffic between clients and monitors, thus preventing public detection. Of course, a client could choose to audit the proof of misbehaviour themselves, and perhaps even stop using the log, but without the ability to publicise the misbehaviour the detection mechanism is extremely limited in scope. Thus, in order to ensure that proofs of misbehaviour do indeed reach parties who can act on them, CT requires some form of dissemination of client artefacts.

Note also that a pair of not-provably-consistent STHs, even at different clients, also comprises proof of misbehaviour. Thus such a dissemination protocol, if working correctly, should also enforce the aforementioned consistency property of log servers.

Unfortunately, CT as it stands does not specify a dissemination protocol except in very broad strokes. We briefly review a proposed implementation, and then suggest one possible format for the protocol.

Privacy SCTs received from TLS connections must also be disseminated, to prevent the attack whereby a malicious log server issues an SCT which it does not honour, and subsequently ignores audit requests for that SCT from the client. This poses a privacy issue, however, since SCTs directly reveal browsing history. One proposed solution is to overlay the SCT audit requests onto DNS lookups, since DNS servers already have access to browsing history. This functions in the normal case but does not specify what a DNS server should do with an SCT it sees but cannot audit.

Adversary model The adversary of the previous sections, who has total control of the network, is no longer applicable to an analysis of any gossip protocol: indeed, it is trivial to block dissemination if you have the power to block all

traffic! Instead, we consider an adversary with significant but not unconstrained power to control traffic, such an ISP or a government. Such an adversary has the ability to block or intercept any particular connection, but cannot block all connections permanently. (For example, it might be feasible permanently to block a monitor server, but not all major websites.)

This constraint leads us to a protocol design in which clients re-use reliable connections in order to transmit certificate auditing data to participating servers. To this end, the CT project has set up a TLS extension for gossip, which allows clients to send STHs to participating servers over otherwise-unrelated TLS connections. In such a scenario, STH dissemination from the client cannot be blocked without also blocking communication with all CT-enabled servers.

5 Conclusion

Certificate Transparency is a protocol designed to enable detection of misissued web PKI certificates, using public append-only logs to enable any participant to monitor the certificates issued for their domains. It was designed to resist various adversaries with differing abilities, but which subprotocols offered which levels of security was never formally written down.

In this project, we present the early stages of such a specification, describing the different subprotocols proposed and the adversaries which they may resist. In particular, we separate the core security properties from the gossip protocol, since their adversary models differ.

The project was brief by construction, and left open more questions than it has answered. We mention here some points which deserve more research or investigation.

- Completing the conjectured security proof of Merkle trees against forgery (Appendix B) would be useful not just for CT but for all the related protocols which rely on append-only logs.
- The use of SCTs in place of audit proofs is unique to CT, and not modelled in our analysis – but of course in practise it leaves a fine avenue for potential exploits, since clients are required to trust the signature of the log server that a proof exists, and by verifying it reveal details of their browsing history.
- The dissemination protocol is necessarily dependent on many difficult-to-model features of the Internet topology, but even so it would be useful to provide lower and upper bounds on how much connectivity is required to ensure a robust dissemination of information. In particular, little thought has been given to the abilities of an adversary to disrupt this propagation, say by selectively manipulating the connectivity of clients, or by blocking certain servers from the Internet.

Acknowledgements

I'd like to thank my supervisor Kasper for his plentiful advice and guidance, as well as Cas Cremers for his explanations of AKI. Thanks also to Google for the original project proposal concept, and in particular to Eran Messeri and Ben Laurie for their insight and prompt responses to questions.

References

[Ada14] Adam Langley. *Maintaining digital certificate security*. July 2014. URL: http://googleonlinesecurity.blogspot.com/2014/07/maintaining-digital-certificate-security.html (visited on 14/08/2014).

[Boy+13] Colin Boyd et al. *ASICS: Authenticated Key Exchange Security Incorporating Certification Systems*. Tech. rep. 398. 2013. URL: http://eprint.iacr.org/2013/398 (visited on 03/09/2014).

[BR94] Mihir Bellare and Phillip Rogaway. 'Entity Authentication and Key Distribution'. en. In: *Advances in Cryptology — CRYPTO' 93*. Ed. by Douglas R. Stinson. Lecture Notes in Computer Science 773. Springer Berlin Heidelberg, Jan. 1994, pp. 232–249. ISBN: 978-3-540-57766-9, 978-3-540-48329-8. URL: http://link.springer.com/chapter/10.1007/3-540-48329-2_21 (visited on 21/05/2014).

[CRY14] Vincent Cheval, Mark D. Ryan and Jiangshan Yu. 'DTKI: Distributed Transparent Key Infrastructure'. In: 2014.

[Goo14] Google. *Certificate Transparency website*. 2014. URL: http://www.certificate-transparency.org/ (visited on 05/08/2014).

[Kim+13] Tiffany Hyun-Jin Kim et al. 'Accountable Key Infrastructure (AKI): A Proposal for a Public-key Validation Infrastructure'. In: *Proceedings of the 22Nd International Conference on World Wide Web*. WWW '13. Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2013, pp. 679–690. ISBN: 978-1-4503-2035-1. URL: http://dl.acm.org/citation.cfm?id=2488388.2488448 (visited on 14/07/2014).

[LKL13] Adam Langley, Emilia Kasper and Ben Laurie. *Certificate Transparency*. 2013. URL: http://tools.ietf.org/html/rfc6962 (visited on 16/07/2014).

[Pri11] J. R. Prins. 'DigiNotar Certificate Authority breach 'Operation Black Tulip''. In: *Fox-IT, November* (2011). URL: https://www.ncsc.nl/binaries/en/current-topics/news/interim-report-on-diginotar-digital-breach-published/1/interim+report+on+diginotar+digital+certificate+breach.pdf (visited on 16/07/2014).

[SMVB01] Sjouke Mauw and Victor Bos. 'Drawing Message Sequence Charts with LaTeX'. In: *TUGBoat* 22 (June 2001), pp. 87–92.

[WAP08] Dan Wendlandt, David G. Andersen and Adrian Perrig. 'Perspectives: Improving SSH-style Host Authentication with Multi-path Probing'. In: *USENIX 2008 Annual Technical Conference on Annual Technical Conference*. ATC'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 321–334. URL: http://dl.acm.org/citation.cfm?id=1404014.1404041 (visited on 21/05/2014).

[YCR14] Jiangshan Yu, Vincent Cheval and Mark Ryan. 'DTKI: a new formalized PKI with no trusted parties'. In: *arXiv:1408.1023 [cs]* (Aug. 2014). arXiv: 1408.1023. URL: http://arxiv.org/abs/1408.1023 (visited on 28/08/2014).

A Examples

We briefly explain the normal use of the different sub-protocols used in CT to issue and use a certificate. Message sequence charts drawn with the LaTeX `msc` package [SMVB01].

Certificate Issuance The owner `server` of some domain requests a CT-enabled certificate from her certification authority `ca`. `ca` generates and signs a standard X.509 certificate C, then sends C to a log server `log`. `log` signs C and the current time to generate an SCT, sending it to `ca`, who staples the SCT to C and returns it to `user`. The standard presents multiple ways to staple the SCT to the certificate: via an X.509 extension, via OCSP stapling, or in the TLS handshake. For the former case, `ca` in fact generates an SCT not on the entire certificate (since part of that certificate is the very same SCT) but on a related datum called a precertificate.

TLS handshake TLS clients work with trusted auditors; in practice, for instance, web browsers will include an audit component. A TLS client `client` begins a connection with `server` and indicates CT support in the protocol negotiation. `client` may query its trusted auditor `audit` for a current STH and transmit it to `server`, but `server` must transmit its certificate C with stapled SCT to `client`. `client` verifies `ca`'s signature on C and `log`'s signature on the SCT, then sends the SCT and certificate to its auditor.

Asynchronous verification Auditors communicate with TLS clients as well as asynchronously checking that the information they receive is consistent. An auditor `audit` may query `log` for the latest STH, together with a proof of consistency between it and `audit`'s current one. While it has queued STHs, `audit` fetches an SCT from its queue and may query `log` for an audit proof between the SCT and its current STH.

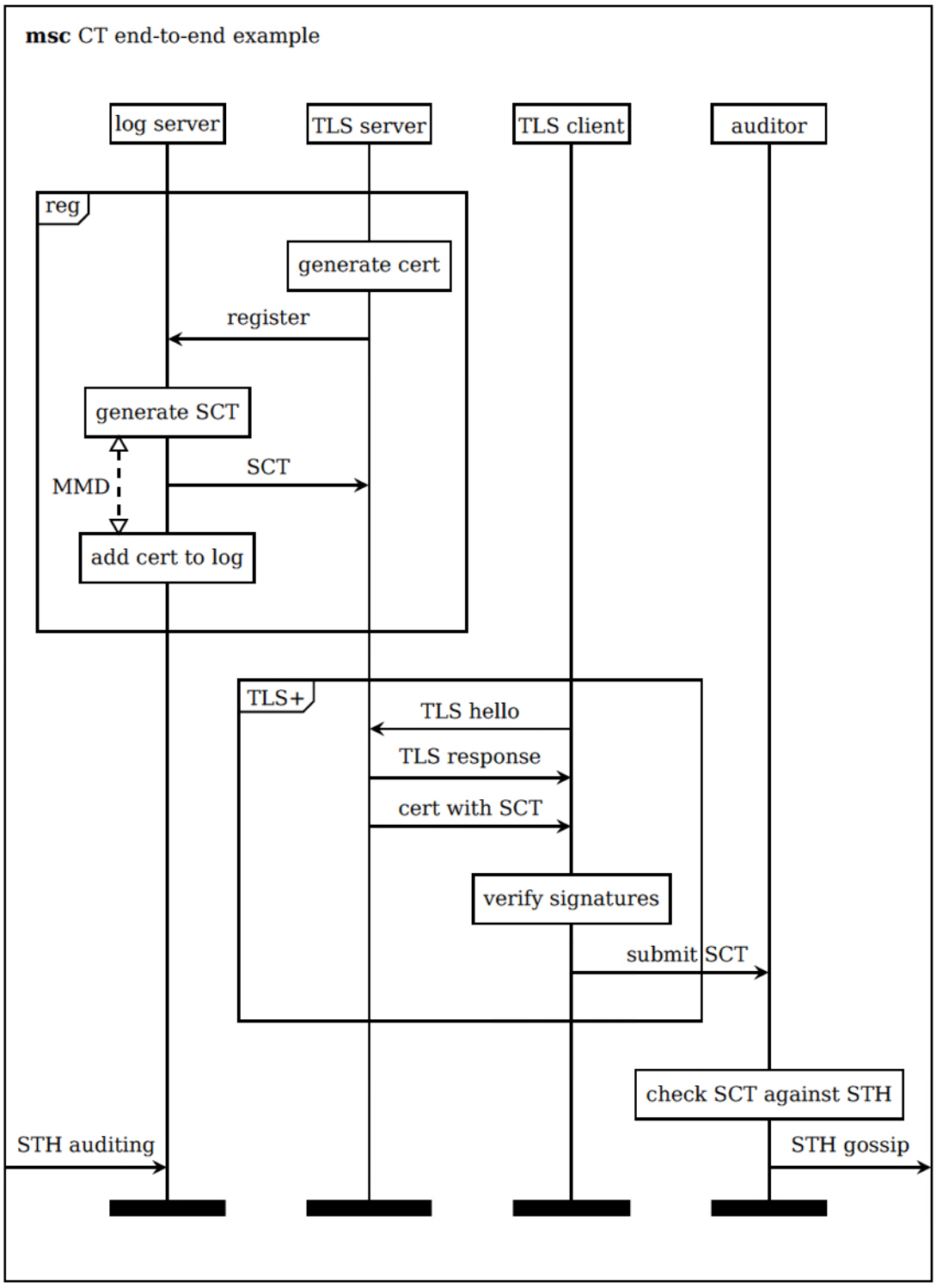
Distributed consistency checking At least auditors, but generally all willing parties, gossip information between each other, with the final goal of ensuring that every proof artefact eventually reaches an uncompromised monitor server. As part of this gossip, clients may send artefacts to servers over existing TLS connections, relying on the fact that these connections are difficult for an adversary to disrupt permanently.

A.1 Compromise

If an adversary compromises a CA root certificate, it may use it to sign other certificates for arbitrary domains. As part of this generation process, the adversary must provide an SCT with the certificate, signed by a log server the intended client trusts.

The only way a log server will issue such an SCT, unless compromised, is upon receiving the relevant certificate for addition to its logs. In this case, the misissuance will be publicly visible; it is up to the owner of the relevant domain to detect and revoke it.

Alternatively, the adversary might compromise the log server's signing key in order to generate the SCT. This will create an apparently-valid certificate, which clients will accept – but as part of the asynchronous verification they will store the SCT and certificate to request inclusion proofs in their views of the compromised log. Since anyone can request verification of an SCT, this artefact represents proof of the compromised log's behaviour. Once it is disseminated to the relevant authorities, it is therefore publicly verifiable and the log server and CA will be removed from trusted lists.


msc CT end-to-end example
log server
TLS server
TLS client
auditor
reg
generate cert
register
generate SCT
SCT
MMD
add cert to log
TLS+
TLS hello
TLS response
cert with SCT
verify signatures
submit SCT
check SCT against STH
STH auditing
STH gossip

B Security Games for CDSs

Recall that a verifiable chronological data structure (vCDS) is a data structure storing an append-only sequence of values and admitting short, remotely-verifiable proofs of behaviour. We model this verification as a game between a probabilistic, polynomial-time (PPT) adversary $\mathcal{A}$ and a vCDS oracle. Formally:

Definition B.1. A *verifiable CDS algorithm* is a tuple $(\mathsf{Types}, \mathsf{SetupG}, \Psi, \mathsf{Verify})$, where

- $\mathsf{Types} = (H, C, D, P_{\mathsf{presence}}, P_{\mathsf{extension}}, \bot)$ is a tuple of sets denoting possible values for contents of the CDS, digests of the CDS, proofs of presence and of extension, and the special value $\bot$ representing an error value in a computation.
- SetupG is a key generation algorithm which produces a public/private key pair.
- $\Psi = (\Psi_{\texttt{contents}}, \Psi_{\texttt{add}}, \Psi_{\texttt{presence}}, \Psi_{\texttt{extension}})$ is a tuple of PPT algorithms implementing the behaviour for each required operation. Each Ψ_q takes as input the local state of a machine as well as the arguments to its query, and returns an updated state as well as a value in the relevant set or $\bot$.
- $\mathsf{Verify} = (\mathsf{Verify}_{\mathsf{digest}}, \mathsf{Verify}_{\mathsf{presence}}, \mathsf{Verify}_{\mathsf{extension}})$ is a tuple of PPT algorithms describing how to verify digests and proofs of presence and extension. Each takes public information (such as the security parameter or certain public keys) and returns a boolean value $\{0, 1\}$.

The security game ForgeDigest (resp. ForgePresence, ForgeExtension) is a game played between a log server $\mathcal{L}$ and an adversary $\mathcal{A}$, defined as follows:

initialisation SetupG is executed to generate a key pair, of which the public key is given to both parties and the secret key only to $\mathcal{L}$

computation $\mathcal{A}$ computes with access to `contents`, `add`, `presence` and `extension` queries. The response to a query q is the result of executing Ψ_q at $\mathcal{L}$ on the arguments given to q.

result $\mathcal{A}$ outputs a value v, and wins iff $\mathsf{Verify}_{\mathsf{digest}}(v)$ (resp. $\mathsf{Verify}_{\mathsf{presence}}(v)$, $\mathsf{Verify}_{\mathsf{extension}}(v)$) is true but $\mathcal{Q}(v)$ is not, defined by

- $v \in C$ and $\mathcal{Q}(v)$ iff v was a response to an `add` query, for VerifyDigest;
- $v = (c, p, d) \in C \times P_{\mathsf{presence}} \times D$ is a proof p of presence of c in the CDS denoted by d, where $\mathcal{Q}(v)$ iff c was an argument to an `add` query, for VerifyPresence; and
- $v = (d_0, d_1, p) \in D \times D \times P_{\mathsf{extension}}$ is a proof of extension between the CDSs denoted by d_0 and d_1, where $\mathcal{Q}(v)$ iff both d_0 and d_1 were responses to `add` queries, for VerifyExtension

The CT logs are implemented by Merkle hash trees; digests are given by a signature on the root hash of the tree together with a timestamp, and the tree size. We can reduce forgery of these digests to forgery of the signature, as follows, assuming the signature scheme is existentially unforgeable under a chosen-message attack.

Lemma B.2. *CT logs have unforgeable digests, in the sense that a PPT adversary $\mathcal{A}$ who wins* ForgeDigest *with non-negligible probability gives rise to a PPT adversary $\mathcal{A}'$ violating security of the signature used.*

Proof: Recall that existential unforgeability under a chosen-message attack means that $\mathcal{A}'$ has access to a signing oracle Sign, and may adaptively request signatures on arbitrary data, winning if she can output a signature which passes verification but which was not produced by the oracle. To reduce ForgeDigest to this game, we must define for every adversary $\mathcal{A}$ in ForgeDigest a corresponding adversary $\mathcal{A}'$ in Sig-forge.

Suppose $\mathcal{A}$ has probability of success $\epsilon(1^k) > \mathsf{negl}(k)$ at ForgeDigest. Define $\mathcal{A}'$ as follows:

- $\mathcal{A}'$ receives a public key pk, and initialises a simulated copy of $\mathcal{A}$. $\mathcal{A}'$ also creates a list C and an empty Merkle tree τ in its own memory, preparing to act as the oracle for $\mathcal{A}$.
- $\mathcal{A}'$ responds to oracle queries by $\mathcal{A}$ as follows:
  - for a `contents` query, $\mathcal{A}'$ returns C
  - for a $\mathsf{add}(v)$ query, $\mathcal{A}'$ appends v to C, adds v to τ to produce a new root hash h, submits $(h, \text{length}(C))$ to the signing oracle to produce $d = \{h, \text{length}(C)\}_{\mathsf{pk}}$, and returns d
  - for a $\mathtt{presence}(v)$ query, if $v \in C$ then $\mathcal{A}'$ returns the sequence of hashes required to prove the presence of v in τ; otherwise returning $\perp$
  - for a $\mathtt{extension}(d, d')$ query, if d' is an extension of d then $\mathcal{A}'$ returns the sequence of hashes required to prove it; otherwise returning $\perp$
- eventually, $\mathcal{A}$ produces an output v which, with probability $\epsilon(1^k)$, is a signature by pk on $(h, \text{length}(C)$ not returned by an add query. Since the only submissions to the signing oracle are as a result of add queries, v also wins Sig-forge with probability $\epsilon(1^k)$.

Thus, if $\mathcal{A}$ wins ForgeDigest then $\mathcal{A}'$ wins Sig-forge, as required. ■

This result is intuitively obvious, and does not use the Merkle hash structure at all: since digests are signed, an adversary wishing to forge them must necessarily forge the signature. More interestingly, we would like to consider the proofs of presence and of extension, and to show that forging them implies computing a hash collision.

Conjecture B.3. *CT logs have unforgeable proofs of presence, in the sense that a PPT adversary $\mathcal{A}$ who wins ForgePresence with non-negligible probability gives rise to an PPT adversary $\mathcal{A}'$ violating collision resistance of the hash.*

Proof idea: Recall that collision resistance means that $\mathcal{A}'$ is given a particular hash function (generally a "key" s indexing one hash function from a fixed family), and must output a pair of values x, x' such that $H^s(x) = H^s(x')$. To reduce ForgePresence to this game, we must define for every adversary $\mathcal{A}$ in ForgePresence a corresponding adversary $\mathcal{A}'$ in Hash-coll.

Suppose $\mathcal{A}$ has probability of success $\epsilon(1^k) > \mathsf{negl}(k)$ at ForgePresence. We construct an adversary $\mathcal{A}'$ who can produce a hash collision. Indeed, consider the definition of success at ForgePresence: $\mathcal{A}$ must produce a proof of presence for a Merkle tree i.e. an item c, a digest d and a sequence of values $v_1, v_2, \ldots, v_k$ such that $H(\cdots H(H(c, v_1), v_2) \cdots, v_k) = d$ but c was never added to the log. Recall that $\mathcal{A}$ must produce a value v, a proof p and a digest d for which the proof shows containment. If d was not the response to an add query, then this adversary succeeds at ForgeDigest by outputting just d, contradicting the previous lemma. Hence we may assume that the digest was returned by a query to the (honest) oracle.

Let N be the size of the tree maintained by the oracle after executing the add query which returned d. We proceed by induction on N, with hypothesis that there does not exist an adversary which can win ForgePresence with a digest for a tree of size N.

If $N = 1$ then the tree holds a single element v', and hence the digest is defined to be $H(0, v')$, where v' is the argument to the add query. Since VerifyPresence returns true, we have by its definition that $H(0, v) = d$. But $v \neq v'$ by definition of $\mathcal{Q}$, so that the tuples $(0, v)$ and $(0, v')$ are a hash collision for H.

The inductive case is conjectured similarly. ■

Conjecture B.4. *CT logs have unforgeable proofs of extension, in the sense that a PPT adversary $\mathcal{A}$ who wins ForgeExtension with non-negligible probability gives rise to a PPT adversary $\mathcal{A}'$ violating collision resistance of the hash.*