

# Automated Verification of Certificate Transparency

**Abstract**— Certificate Transparency (CT) aims to reduce the trust required in Certificate Authorities (CAs) within the TLS certificate ecosystem. It is supported by all major browsers. The protocol obliges all CAs to record the certificates they issue in a public log, which itself is monitored for compliance and consistency by third parties. Given this complex set of checks between the four roles—CA, log, monitor but also the end-user’s client—it is very hard to give a precise account of how CA eliminates trust assumptions in favour of complex infrastructure. Existing analyses, both in DY model and the computational model, only regard a very simplified model and feature definitions adapted specifically to CA, essentially capturing design features rather than the target property.

The present paper posits accountability as the main goal of CT and presents a thorough analysis in the Dolev-Yao model. We start with the vanilla PKI and, step by step, move to CT, finally analysing proposed extensions for SCT Auditing and Gossiping. We show that CT relies on an honest log, but provides accountability under this assumption. We show that the SCT Auditing extension can eliminate this assumption, while the Gossiping extension cannot.

## 1. Introduction

Certificate Transparency (CT) has become a standard security feature in all major browsers today. Its primary goal is to enable the detection of rogue Certificate Authorities (CAs) by ensuring that all issued certificates occur in publicly accessible and verifiable logs. Rogue certificate authorities can break the authenticity of the public key infrastructure (PKI) when they issue certificates with malicious intent, forgoing the rigorous ownership validation they are required to do. Publicly accessible logs (such as introduced by CT) allow anyone to look for potentially misissued or fraudulent certificates. If such a certificate is found, steps for revocation can be taken and, ultimately, the CA can be held accountable for issuing the certificate. The increasing use of such a protocol leads to the desire for a formal verification of the standard.

We apply automated analysis to CT using Tamarin in the Dolev-Yao model. Tamarin has built-in support to prove accountability: we define a set of *case tests* that identify misbehaving parties that violate a security property from inside the protocol, i.e., without an outside view of every action. We verify, formally and automatically, that this identification contains all parties it should and no party it should not. Our model of CT is the most detailed to date and the first formal model to cover maliciously maintained logs, Merkle proofs

on these logs, corrupted CAs and monitors. Our results are the first that take into account corruption scenarios involving all four roles: CA, log, monitor and web server. Compared to existing results outside formal methods, we are not only the first to be able to hold logs or monitors accountable, but the first to guarantee that (a) all responsible parties are held accountable (not just one) and (b) no innocent party is blamed.

Certificate Transparency offers an interesting case study for accountability. The primary goal of CT is not to prevent any misbehavior by CAs, but instead to ensure CAs can be audited so that their misbehavior can be identified early on. We consider accountable authenticity and show which assumptions are necessary required to prove it. Our goal is to show that if authenticity is violated, CT provides the necessary information to identify the misbehaving parties causing the violation. In the first step, we find attacks refuting this claim. In the second, we formulate assumption that prove it, but are unrealistic. In the third, we show additional infrastructure (in CT, or, later, in extension of CT) may ensure these assumptions. We perform this detailed analysis for basic CT, for SCT auditing (as implemented in the Chrome browser) and STH gossiping (an academic proposal). With SCT auditing, we can achieve the best guarantees, but at the cost of privacy for clients.

In addition to accountable authenticity, we consider transparency and accountable transparency. Similar attacks often work against both transparency and accountable authenticity, highlighting the close connection between both properties. Given transparency, we show accountable authenticity in CT and its extensions.

In Section 2 we start with necessary background, including the CT standard and its proposed extensions. Section 9 summarizes existing works that consider CT and accountability. Section 3 gives a brief overview of modeling using Tamarin and further details on the accountability notion we use in this work. We then start incrementally introducing our model and consider the PKI setting first in Section 4, where we show that authenticity can be violated and accountable authenticity achieved only given an external validator, which is not applicable in practice. Section 5 then adds Certificate Transparency to the PKI model. We show that authenticity is still violated and that accountable authenticity requires transparency, which can be violated too. In Sections 6 and 7 we consider two additional proposals, SCT auditing and gossiping, and attempt to prove the same properties. We find that SCT auditing requires the fewest additional assumptions to achieve accountability. Section 8 summarizes our findings and presents details on the proofs for every considered

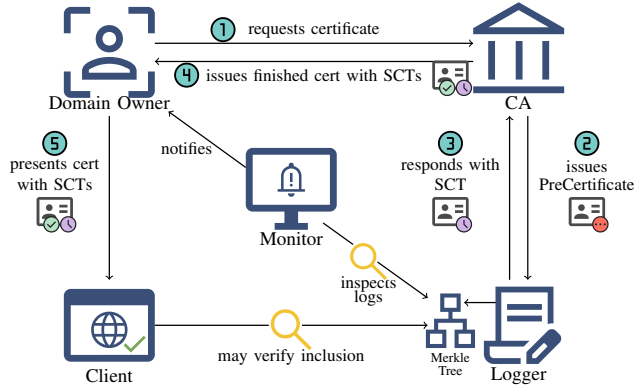


Figure 1. Different roles and usual information flow in the CT protocol with pre-certificates.

property and model.

## 2. Certificate Transparency

To ensure the authenticity and integrity of the TLS handshake used in HTTPS, hosts are required to present a certificate that maps their identity (including the domain the user typed in the URL bar) to their public key. These certificates are produced by third parties, known as Certificate Authorities (CAs). The clients trusts the CAs that are (transitively) trusted by a root CAs. These come with the browser and are all equally trusted. There are hundreds of trusted CAs out there, every one of which is a single point of failure. Root CAs can—and have in the past **obrien**, [1]—issued rogue certificates.

By virtue of being signed, rogue certificates identify their issuer, hence Certificate Transparency (CT, [2], [3]) makes them public so browser vendors can remove the CAs that issued their set of trusted root CAs. We show that the log indeed provides actionable information required to hold the (causally) responsible CAs, and only those, accountable for integrity violations.

Certificates in both the PKI and CT follow the X.509 format [4] and contain the certificate’s issuer, the object’s (e.g., website’s) identity and associated public key, and they have a validity periods. X.509 allows extensions fields, which are used for CT.

**Loggers.** Loggers maintain a publicly accessible list of certificates issued by CAs. A valid entry to the log must consist of the certificate being logged and the complete chain of certificates required to validate it. Entries can be submitted by anyone and the logger must accept them without performing further checks. When accepted, the logger responds with a Signed Certificate Timestamp (SCT), representing the logger’s signed intent to include this submission in its log.

In the context of CT, a *certificate* embeds one or more SCTs using an extension field. The CA issues the certificate given one (or more) SCTs and a *pre-certificate*, which (a)

contains all fields that the final certificate should contain except for the SCT and (b) is marked with a special ‘SCT’ extension that ensures clients will never accept it. Loggers accept pre-certificates and thus issue an SCT. The entry they add to the log is not specifically marked as a pre-certificate. Figure 1 visualizes the protocol flow.

Other flows exist (e.g., delivering SCTs via OCSP instead), but embedding is the most common method because it requires no specific implementation in the web server [5], [6].

**Signed Certificate Timestamps.** Signed Certificate Timestamps (SCTs) contain the identity of the logger that issued them, a version field and the time they when they were issued. The signature is calculated over these fields and the data fields of the certificate it has been issued for. This allows attribution of an SCT to a certificate when validating the SCTs signature.

SCTs represent the promise of a logger to include the underlying certificate in its log but are not a guarantee. A corrupted logger can issue SCTs without ever including the corresponding entry visibly in its log. Thus, other parties may verify the inclusion in the log at a later point to ensure the log is maintained honestly.

Even honest loggers can have a time gap between handing out an SCT for a certificate and including the entry in their log, which is known as the *maximum-merge delay* (MMD). It is one of the configuration parameters of a log and is usually set to 24 hours.

**Merkle trees.** CT uses Merkle trees to implement logs. Merkle trees store a sequence of entries and allow the holder to efficiently prove *inclusion* of specific entries and the *append-only* property, that two snapshots of the log are *consistent*, i.e., the second is the first with some entries appended. It is trivial to prove these properties by sending the full tree to the verifier, but Merkle trees manage to provide proofs that are only logarithmic in the size of the tree. We refer to for details.

Important for CT is that Merkle Trees provide a function `AddLog` to add an entry to an existing log and that a tree can be signed by simply signing its root (also called its *head*), which gives the *Signed Tree Head (STH)*. The logger produces the STH, which can be seen as a commitment to the log entries at some given time. Besides the straightforward way of opening this commitment by fetching all entries and recomputing the tree head and thus the STH, STH can be used in *inclusion proofs*, which verify whether a certificate was part of the log with a given STH, and *append-only* proofs, which verify whether a new STH is the result of appending items to the tree represented by the former STH. This ensures that no entries have been removed from the log. We omit these algorithms; only their correctness matters for our model.

Cheating loggers can hide a specific certificate by maintaining two views, e.g., Merkle tree  $T_1$  contains a rogue certificate and can be used to prove inclusion, while  $T_2$  is the same but without this certificate. Presenting  $T_2$  to a monitor

looking for rogue certificate, the monitor will fail to detect the misbehavior. This equivocation is known as *partition attack* and shows that, contrary to its name, CT by itself does not provide transparency [7], [8].

**Monitors.** Owners of certificates (called *subjects*) can inspect logs via monitors. They can use tools like cert spotter [14] to monitor for their own certificates, or appoint third parties to do it for them [15]. A monitor would, for instance, fetch all entries from a logger to check for certificates with a specific subject (e.g., their own) and an unknown key. For a third-party monitor, the target subject and acceptable keys needs to be communicated out of band. If a rogue certificate is found, this indicates potential misbehavior of a CA. The monitor reports to the subject, which can then take further action to get the certificate revoked, which are not defined in the CT standard. In this work, we define the concrete conditions under which the certificate and corresponding SCT correctly prove who can be held accountable. Indeed, CT logs were used to detect rogue certificate. For instance, Google’s efforts to require log inclusion for extended validation certificates leads them to find that Symantec issued multiple rogue certificates, including one for google.com in 2015 [16]. Symantec’s CA was subsequently forced to only issue CT confirming certificates and, due to further compliance failures, is entirely distrusted by Chrome, and thus a large number of Internet users, at this point [17].

## 2.1. Current deployment

We investigated the current deployment of certificate transparency in the most popular web browsers, see Table 1.

Although CTv1 was made obsolete by its successor, CTv2 [2], [3], [18], all browsers still implement CTv1. These versions only differ in certificate formats and the data structures used for implementation. As these are abstracted in our model, both our model, results and the following discussion also apply to CTv2. The only exception is that CTv2 considers the possibility for web servers to attach inclusion proofs to the SCTs it sends the client. This improves privacy, as clients would not expose to the log which certificate they are interested in when they fetch the proof from the log. With regards to authenticity, this has not security implications; our models for CT with or without extensions would stay the same.

Firefox recently started to enforce CT validation for certificates, but only for their desktop versions [18]. Only Chrome provides an additional auditing mechanism. We did not find any documentation describing client-side inclusion proof fetching. We describe Chrome’s auditing mechanism later in Section 6.

Browser vendors summarize their CT requirements in CT policies. They define acceptance criteria, e.g., how many valid SCTs are required. These policies only consist of requirements on the number of SCTs, how they are delivered (embedded or separately), and a set of trusted loggers whose SCTs are accepted.

We compare these design decisions with the guarantees that we are able to prove and required assumptions in Section 8.

## 3. Background

### 3.1. Tamarin prover

We use Tamarin to apply automated analysis in the Dolev-Yao model [19], wherein messages are described as abstract terms composed from uninterpreted function symbols that represent cryptographic primitives and are applied to *variables* representing yet unknown values and *names* serving as the base type, representing either high-entropy values like keys (then the name is *fresh*) or publicly known constants (then the name is *public*). If we want to clarify the type, we write  $\sim n$  for the former and  $\$n$  for the latter.

The behaviour of these cryptographic primitives is specified by an equational theory on terms. For example, a hash function is represented by function symbol  $h$  and the empty equational theory, essentially describing it as a random oracle. We write  $h/1$  to indicate that  $h$  takes 1 parameter. Symmetric encryption and decryption are typically written as the function symbols  $sign$ ,  $verify$ ,  $true$  and the equation  $verify(sign(m, sk), m, pk(sk)) = true$ , meaning that verifying a correctly signed message reduces to the 0-ary function symbol (i.e., constant)  $true$ .

**3.1.1. Multiset rewriting.** The overall state of a protocol execution is described as a multiset of *facts* where a fact has a fact symbol  $f$  and list of terms, typically describing the state a protocol party is in. E.g.,  $S_2(\$server, \sim nonce)$  if a server with id  $\$server$  has previously received a nonce and is ready to respond.

*Rules* then described the dynamics of the protocol. They have form

$$l \multimap [a] \rightarrow r$$

where the premises  $l$  contains a multiset of facts required to be in the current state for this rule to be applicable. These are removed and substituted with those in the conclusion  $r$ . As set of facts  $a$ , the actions, labels this transition.

Persistent facts, prefixed with  $!$ , will not be consumed. Such facts are, for instance, used to denote adversary knowledge that does not shrink over time. The builtin facts  $In$ ,  $Out$  and  $Fr$  model network input and output, respectively the choice of a fresh name.

**3.1.2. Trace properties.** Given a set of rules, a protocol execution is any chain of rewrites starting from the empty multiset and a trace the sequence of labels of an execution, skipping the empty label  $\emptyset$ . Security properties are expressed in a first-order logic with quantification over terms and time points (indexes in the trace). The atom  $f(\dots)@i$  expresses a fact  $f(\dots)$  occurs at time point  $i$ . Other atoms are equality of terms and comparison of time-points.

	Chrome [9], [10]	Edge <sup>2</sup>	Firefox (Desktop)[11]	Firefox (Mobile)[11]	Safari [12]	Brave <sup>3</sup> [13]
CT supported / enforced	✓	✓	✓	✗	✓	✓
client-side inclusion check	✗	—	✗	✗	✗	
SCT Audit [10]	● <sup>1</sup>	—	✗	✗	✗	
# of required SCTs	2 <sup>4</sup>	—	2 <sup>4</sup>	—	2 <sup>4</sup>	2 <sup>4,6</sup>
# of trusted loggers	7	—	7	—	8	7–8 <sup>6</sup>

<sup>1</sup> for random set of certificates <sup>2</sup> no public policy, hence tested or unknown (?) <sup>3</sup> follows policy of Apple and Chromium, depending on platform

<sup>4</sup> 3 if validity > 180 days but only 2 from distinct log operators.

TABLE 1. CT FEATURES PER BROWSERS.

### 3.2. Accountability

Tamarin has built-in support for formulating and proving accountability since Morio and Künnemann [20]. Their definition is protocol agnostic and based on causality; a protocol provides accountability for property  $\varphi$  if it can always correctly identify the of parties whose deviation from the protocol (jointly) caused [21] a violation  $\neg\varphi$ . Consequently, iff the protocol identifies no such set of parties, then  $\varphi$  must hold, hence accountability for  $\varphi$  implies verifiability for  $\varphi$ .

In this context, CT’s task is to find out whether  $\varphi$  was violated and who was (part of) a cause for that. Practically, this means that CT specifies that each party provides proof for their own correct behavior and collects evidence for other parties’ correct behavior. Parties might deviate, but may be detected by other parties. There is no party with a complete view of the network. We do not model how misbehavior is punished, as we consider this outside the protocol. By contrast, the precise conditions under which a party is blamed we consider very much part of the protocol, and currently underspecified. Our task is to figure out these conditions (we call them *tests*) and the precise property  $\varphi$ .

In Tamarin, tests are defined as trace properties with unbound variables for one or more party identities. For instance, the following tests check for a certificate that is later (externally) found to be incorrect. Depending on whether the intermediate CA is legitimate (`Ext_LegitimateCA`) or not (`IllegalCA`), either the root CA or the intermediate CA is blamed. We consider an intermediate CA legitimate if a root CA can provide the documents attesting its identity. Root CAs have to provide this information according to [22]

```

test CfalseAsserts_rootNotBlamed:
"∃ rootCA [..] .
// External assessment refutes cert's statement..
AssertsNot(idCA, stmt, CA_fields)@t0
// ...and was signed by legitimate CA -> blame CA
∧ Ext_LegitimateCA(rootCA, CA_fields, idCA)@t1"

test CfalseAsserts_rootBlamed:
"∃ idCA [..].
AssertsNot(idCA, stmt, CA_fields)@t0
// ...and was signed by illegitimate CA.
∧ IllegalCA(rootCA, idCA, CA_fields)@t1"

```

Observe that the underlined variables `idCA` and `rootCA` are not bound. Given a set of tests and a trace  $t$ , the *verdict* is the set of all sets of party identities for which some test is satisfied on  $t$ . This function should be correct, i.e., identify each minimal set of parties that (jointly) caused

an attack (i.e., a violation of  $\varphi$ ) and did so by deviating from their normal behavior (i.e., the protocol). For instance, if  $t$  is such that three attacks happen, two matching the first test and one matching the second, then the verdict identifies three singleton causes, two intermediary CAs and one root CA. Correctness now asserts that for each attack, the respective intermediary or root CA was indeed deviating and without them,  $\varphi$  would not have been violated (in that particular attack).

Morio and Künnemann provide us with a decomposition of accountability into trace properties, each necessary and all together sufficient. We present four of those that are intuitive and help us intuit attacks that we will see later.

- *sufficiency*: Each test has a trace that matches with only the blamed parties corrupted.
- *verifiability*: Should no test match,  $\varphi$  holds, and vice versa.
- *minimality*: No strict subset of some test’s verdict can trigger that test or any other.
- *uniqueness*: Whenever a test matches, the blamed parties must be under adversarial control.

The remaining two (injectivity and single-matchedness) are never violated and require deeper explanation, we thus refer to [20], as well as for the syntactic conditions (BR and RP) that Tamarin checks for. For BR, all tests in this work succeed. For RP, the syntactic condition in Tamarin is too strict, we thus checked this condition manually, see ??.

### 4. The standard PKI

We start by laying out the PKI used on the web, which puts unverifiable trust into the CAs. We then add an external validator, and show it can hold the previous parties accountable. Even though this validator must be online at all times and trusted by each client, and is thus unrealistic, it helps set up CT, which essentially sets up infrastructure to distribute this validation task. In the following chapters, we will proceed similarly for CT and extensions. Each time, we add new infrastructure (first monitors, then logs) and show how the protocol moves trust from a previously trusted role to a new role. Each time, trust is moved to a place that provides better efficiency or better incentives to maintain trust.

We use standard function symbols `pk/1`, `sign/2` and `verify/3` and the equation `verify(sign( $x, y$ ),  $x$ , pk( $y$ )) = true()` to model signatures. Certificates are signed 7-tuple

representing the different fields of the certificate, including the identities of issuer (1) and subject (2) and the subject's public key (3). In the equation above,  $x$  is this 7-tuple, and  $y$  the issuers secret key.

Another value in the 7-tuple signifies the type of the certificate (4). When set to 'selfCert', it represents a trusted root certificate. Issuer and subject are the same, hence the certificate is signed with the public key the tuple contains. When set to 'pubKey-CA' or 'pubKey', it is a regular public key certificates and signed with the issuer's public key, which is different from the subject's (intermediary CA's or web server's) public key. For 'pubKey-CA' the certificate can sign others (i.e., is an intermediary CA), 'pubKey' is for end-entities such as web servers. The remaining fields contain a fresh serial number (5) and a validity period, represented by two timestamps (6 and 7). We call issuer (1), subject (2), public key (3) and serial number (5) the *statement* of a certificate.

Details about the CA model are in Section A, here we present only the client, e.g., a web browser. When validating an incoming certificate, it checks:

- 1) The certificate has been issued and signed by a CA that has been signed by a trusted root CA.
- 2) The signature of the intermediate CA certificate verifies under the root CA's public key.
- 3) The signature of the certificate verifies under the CA's public key.
- 4) The certificate is still valid.

**rule** ClientValidate:

```

let
  CA_fields = <$rootCA,$idCA,pkCA,'pubKey-CA',
               snCA,$i0,$j0>
  pub_fields = <$idCA,$id,pkID,'pubKey',
               sn,$i,$j,sct1,sct2>
  stmt = <$idCA,$id,pkID,sn>
in
[ !Root_CA($rootCA,pkRootCA,root_fields,
            root_self_sig), // check 1
  // incoming certificate chain
  In(CA_fields), In(ca_sig),
  In(pub_fields), In(pub_sig),
]  $\neg$ 
ClientAcceptsChain($rootCA,CA_fields,$idCA,stmt),
// check 2) and 3)
Eq(verify(ca_sig, CA_fields, pkRootCA), true()),
Eq(verify(pub_sig, pub_fields, pkCA), true()),
Time($now), CheckValidUntil($now, $j), // check 4
]  $\mapsto$  [ ]

```

The action `ClientAcceptsChain` indicates that this rule was applied, i.e., a client accepted this certificate and the intermediate CA certificate as valid. We used it to state our security goal  $\varphi$ .

**Modelling timestamps.** For check 3), we need to model the two time stamps included in certificates: time of creation and time of expiration. Tamarin does not have native support of time beyond ordering actions. We use the approach of Morio et al [23] and model timestamps as public values that obtain a meaning through an axiomatic model. Using public variables ensures the adversary can access all time stamps, thereby also simplifying Tamarin's message deduction.

Every rules that needs to measure time obtains an event `Time($now)` and time points are related with the following restriction, i.e., a property that Tamarin assumes to hold.

**restriction** monotonicity:

```

"∀ t #t1 #t3. Time(t)@t1 ∧ Time(t)@t3 ∧ #t1 < #t3
⇒ (∀ tp #t2. Time(tp)@t2 ∧ #t1 < #t2 ∧ #t2 < #t3
   ⇒ tp = t )"

```

As monotonicity is sufficient for the properties we prove, we avoid other assumptions on timestamps. Time can occur in different parties, hence we assume a global clock.

We check if a expiry data has passed by putting the action `CheckValidUntil` in the rule that performs the check. We then constrain traces to those where the check is evaluated correctly:

**restriction** certStillValid:

```

∀ now valid_till #t1.
Time(now)@t1 ∧ CheckValidUntil(now, valid_till)@t1
⇒ ¬(∃ #t2. #t2 < #t1 ∧ Time(valid_till)@t2)

```

**Accountable authenticity.** We posit as the main goal of the PKI to provide an authentic mapping from identity to public key, or, put more generally, that the statement of a certificate, i.e., fields (1—3) and (5), is not forged.

**lemma** authenticity:

```

"∀ idL1 idL2 rootCA CA_fields idCA stmt #j.
  ClientAcceptsChain(rootCA,CA_fields,idCA,stmt)@j
  ⇒ (∃ #i. GroundTruth(stmt)@i ∧ #i < #j)"

```

We immediately find that a corrupt CA can construct a rogue certificate (authenticity). We only obtain this guarantee when we assume no CA is corrupted (`cert_auth`).

To motivate CT, assume the user does not trust all root CAs and sometimes wants to validate if a CA is still honest by checking the certificate externally, e.g., by calling the website owner on the telephone and comparing finger prints. We thus add an external validation oracle that obtains a signed certificate and asserts if its statement is correct (`Asserts_PKI`) or, with a second rule, they are not `AssertsNot`). An additional restriction asserts that the validator never contradicts itself.

Now there is a middle ground between the lemmas `authenticity` and `cert_auth`: authenticity can be violated, but we can hold the responsible CA accountable. If we reconsider the tests from Section 3.2, we can prove the following lemma:

**lemma** A\_PKI\_Ext:

```

CAfalseAsserts_rootBlamed,
  ⊢ CAfalseAsserts_rootNotBlamed accounts for
"∀ rootCA CA_fields stmt #t0 #t1.
  // if the cert's statement was checked and ..
  StatementCheckedExternally(CA_fields, stmt)@t0
  // .. in case of contradiction, the signing CA
  ∧ LegitimateCheck(rootCA, CA_fields)@t1
  ⇒ // then the statement is correct
  (∃ idCA #t. Asserts_PKI(idCA, stmt)@t)"

```

We observe that we have shifted trust from the CAs to external validation. This is, of course, inefficient. CT distributes this validation tasks to multiple monitors operating on a distributed log instead of direct input from the client. Time to start exploring accountability in CT.

## 5. Certificate Transparency

Before we add loggers and monitors to the system, we have to amend certificates with ‘promises’ that the certificate has been added to the log, or will soon be.

**Signed Certificate Timestamps (SCTs).** SCTs consist of 3 fields: the issuer, usually a logger; a type field, in our case just the type ‘SCT’, and a timestamp to indicate when the SCT was issued. It represents the promise of the specified logger to add the certificate to the log by the issuance plus the MMD (usually 24h). In our model, we modify the certificates to embed two SCTs from distinct logs, as all known browser policies require two SCTs (Section 2.1). CT-conforming certificates have two embedded SCTs and carry the ‘PubKey’ type. There are also unfinished public key certificates, which do not embed SCTs and have type ‘pre-cert’, but are otherwise the same. The signature is calculated over all content fields of the certificate, including embedded SCTs.

**Monitors & Loggers.** We introduce loggers and monitors with initialisation rules. As the monitor does not have its own key pair, its initialisation rule only assigns a public variable, representing its identity. Loggers register a public key along with a self-signed certificate. Their key is later used to issue and sign SCTs. The main purpose of logs is to maintain the log structure and hand out commitments and proofs for inspecting parties. We provide these rules in more detail in Section D.

In our model, domain owners can summon a monitor (either themselves or a third-party, see ??) to check new certificates that have the domain owner as subject against a *ground truth*, i.e., a set of tuples (issuer, subject, public key, serial number). Each of these tuples we call a *statement*: the issuer asserts the subject has a specific public key and ensures uniqueness via the serial number. Following certification guidelines, honest CAs assert the statement on pre-certification in an event `GroundTruth` we can use to define rogue certificates (and, analogously, benign certificates). a

```
restriction rogueCert: "∀ idCA stmt #t1.  
  RogueCert(idCA, stmt)@t1 ⇒  
  not (∃ #t0. GroundTruth(idCA, stmt)@t0 ∧ #t0  
    < #t1)"
```

Later on, the monitor will make use of this knowledge to check for authenticity of certificates it encounters in the logs.

**Certification.** As discussed in ??, certification becomes a two step process, which we model with two separate rules. The first issues a pre-certificate and sends it out on the network for two loggers to issue an SCT. The second receives these two SCTs and, knowing the pre-certificate, creates a public key certificate that embeds them. For the client, we adapt the rule `ClientValidate` from Section 4 to additionally verify the two embedded SCTs and add a condition asserting the two loggers be different.

**Modelling logs.** We model each logger’s logs via trace properties, similar to how we modelled time. For example, the action `LoggerComputesAddLog(idlog, ...)` represents additions to the log maintained by *id<sub>log</sub>* and obtains a semantics *axiomatically*, through a restriction on traces. For the single-ledger case, earlier work [24], [25] followed the same approach, but could not represent inclusion proofs in messages on the network, as the log data structure was represented entirely on the trace. Explicit modellings of the underlying Merkle trees [26] do not scale as well (see Section 9 for discussion). Hence we instead extend the axiomatic approach to handle multiple ledgers, MMD and inclusion proofs and append-only proofs. Proofs need to be transmitted on the network and thus require an entirely new axiomatisation. In particular, corrupt loggers must be able to produce ‘incorrect’ logs, so we have a reason to hold them accountable, but if they do, they cannot forge these proofs. Honest loggers also compute these proofs, but they never break these properties.

Instead of considering the log as the outcome of a series of computations, we record the computations themselves on the log. Each logger indexes a chain of computations on the logs with an id. When a logger *L* adds an entry *f* to a log *l*, it emits an action `LoggerComputesAddLog(L, l, f)` that indicates a correct addition to the Merkle tree via the function `ADD` [27]. Removal from the log can be modelled by recomputing it from scratch with a fresh id, but honest loggers maintain only a single log id *l*, as they adhere to CT. Dishonest parties, however, can keep compute multiple logs on the fly using all the information they know and thus perform for equivocation. Any computation that produces a valid Merkle tree is thus available; computations that produce invalid Merkle trees are only represented via malformed proofs, i.e., terms that do not pass validation. Merkle trees are never transmitted, only proofs about them, hence this is w.l.o.g.

We define predicates `EntryVisible` and `EntryNotVisible`, which are used in lemmas and restrictions to define the outcome of a sequence of such computations at a point in time.

```
EntryVisible(L, f, l, #time, commit_time) <=>  
(∃ different_time #t.  
/* There has been an addition to l prior to #time  
  ↳ at MMD period different_time ... */  
  LoggerComputesAddLogTime(L, l, f,  
    ↳ different_time)@t  
    ∧ ∧ #t < #time  
/* ...MMD period has passed */  
  ¬(commit_time = different_time)  
)
```

This rule takes account of the MMD by considering the time of the addition to the log and the time of the snapshot. We consider every action with the same `Time($i)` fact to belong to the same MMD period. In this sense the addition becomes visible as soon as that time has passed. `EntryNotVisible` is in Section D.

**Inclusion proofs and snapshots.** We represent inclusion proofs and commitments as fresh variables, so they can

be linked to the unique point in time where they were computed. We use restrictions to link them to the log at that point in time (the *snapshot*) and given them meaning. The restrictions assert that iff a proof succeeds (i.e., the corresponding event is triggered on the nonce representing the proof) the proof statement holds true, i.e.,

- for an inclusion proof, the entry was visible (see predicate `EntryVisible`) at the time the proof was handed out.
- for an append-only proof, that between two snapshots of a log, no entries disappeared.

Merkle trees and proofs constructed with them are thus assumed to work perfectly, which follows Dolev-Yao model's 'perfect crypto' paradigm.

Proof fetching is a three-step process. A party, in our model either client or monitor, request a proof (Proof-Fetching) from the logger. The logger receives this request (produceSTH\_Proof) and chooses the snapshot to present, that is, a log of their choice (identified by `log_id`). The log has to belong to them and the snapshot represents the time the proof is made. Attackers can present old snapshots of the log by producing a new log (with a new `log_id`) that contains the entries of the older snapshot. In the third step, the snapshot is received and checked for the property of interest. We use different rules and restrictions representing every possible outcome of that check. Note that we assume a private channel for this communication step. This is necessary, because otherwise the network attacker could replay an honest log's old message, thus produce a failing inclusion test that implicate the (honest!) log. This attack transfers to the real world, and indeed, CT demands the communication with the logger be implemented via HTTPS GET and POST requests.

There are four restrictions; two for each proof type for failure and success. We list and explain them in Section E.5 and ??; here, we only quote the failing case for inclusion proofs.

```
restriction inclusionProofFails:
"∀ IdFetcher IdL stmt commit session #t1.
  ↳ InclusionProofFails(IdFetcher, IdL, stmt,
    ↳ commit, session)@t1
    ⇒ (∃ log_id time #t0.
      LoggerPresentsSnapshot(IdL, IdFetcher,
        ↳ log_id, time, commit, session)@t0
        /* Logger handed out an STH to the
        ↳ recipient... */
        ∧ #t0 < #t1 /* ...prior to the check */
        ∧ EntryNotVisible_MMD(IdL, stmt, log_id, #
        ↳ t0, time)
        /* ...and the entry is not visible in
        ↳ that snapshot */
    )"
```

**Accountable Authenticity.** As discussed before, we assume the monitor knows the ground truth and thus add a rule permitting it to raise an alarm. Nevertheless, the monitor needs to be active to perform this check. We thus weaken the accountability lemma.

**Lemma** `A_CT_1 [heuristic=0 "acc_oracle"]:`

```
ca_auth_on_log, root_ca_auth_on_log accounts
↳ for
  "∀ idL1 idM idL2 rootCA CA_fields idCA stmt
  ↳ idL commit session #t0 #t1 #t2 #t3.
    ClientAcceptsChain(idL1, idL2, rootCA,
  ↳ CA_fields, idCA, stmt)@t0
    ∧ FetchLogSnapshot(idM, idL, session)@t1
    ∧ MonitorInspectsLog(idM, idL, stmt,
  ↳ commit, session)@t2
    ∧ LegitimateCheck(rootCA, CA_fields)@t3
    ∧ ((idL = idL1) ∨ (idL = idL2))
    ∧ ProofRunsAfterMMD(idL, commit, session)
    ∧ #t0 < #t1
    ⇒ (∃ #t. GroundTruth(idCA, stmt)@t)"
```

The following test blames any legitimate intermediate CA (`idCA`) that issued a certificate which the monitor flagged as rogue.

```
test root_ca_auth_on_log:
  "∃ idL idL1 idL2 idM [...].
    ClientAcceptsChain(idL1, idL2, rootCA,
  ↳ CA_fields, idCA, stmt)@t0
    ∧ FetchLogSnapshot(idM, idL, session)@t1
    ∧ MonitorBlamesCA_CT(idCA, idL, stmt,
  ↳ commit, session)@t2 /* idCA blamed */
    ∧ Ext_LegitimateCA(rootCA, CA_fields,
  ↳ idCA)@t3
    ∧ ((idL = idL1) ∨ (idL = idL2))
    ∧ ProofRunsAfterMMD(idL, commit, session)
    ∧ #t0 < #t1
  "
```

The second test, omitted here, does the same, but for illegitimate intermediate CAs, consequently blaming any root CA that certified an illegitimate intermediate CA.

Despite our trust in at least one monitor to confirm the certificate in question, we do not manage to hold the CAs accountable: a counterexample shows that a corrupt CA can issue a rogue certificate which is accepted by the client (`ClientAcceptsCert_Key`). Even if the honest monitor becomes active afterwards and fetches one of the logs that ought contain the entry for that certificate, a corrupt logger can hide this entry from the monitor. The monitor would thus fail to blame the CA (let alone the log) and CT would not even ensure verifiability.

**Transparency.** We can, however, prove this property, if we additionally assume the logger honest (`acc_CT_monitor_fetches_honest_log`). This motivates the analysis of *transparency*, i.e., the property that the log contains a certificate after the responsible logger issued an SCT for it. Transparency is also of independent interest to third-parties that rely on the correctness of the logs, e.g., researchers [28]. Given CT's name, one could even argue that user's might expect this property.

In Section F, we formulate transparency as a stand-alone requirement: all entries corresponding to certificates that are actively in use and validated by a some client need to be visible after the MMD, i.e., clients and monitors should be able to prove their inclusion after the MMD. We show that transparency does not hold for dishonest loggers. Assuming honest loggers, we find transparency to hold, but only after the MMD.

To mitigate this problem, we consider two methods for holding the log account: SCT auditing (implemented in Chrome) in the next chapter, and Gossiping [8] afterwards.

## 6. SCT auditing

*SCT auditing* was adapted as an opt-in mechanism in Google Chrome [10], [29]. Instead of the clients, the monitors request inclusion proofs from the loggers. Clients upload a random selection of SCTs and certificate chains they have encountered to a particular version of the monitor that is currently maintained by Google. Having a large set of SCTs and underlying certificates at hand puts the monitor in a much stronger position: now, the monitor can validate a logger's entries and commitments w.r.t. those. If the monitor knows the ground truth, it can use these certificates for authenticity checks. It can also enhance transparency by treating SCT auditing as a secondary source for discovering new certificates, rather than relying solely on logs. Finally, the received SCTs serve as promises that the monitor can verify. If there are SCTs that do not have a counterpart in the log the monitor retrieves, this could indicate potential misbehavior.

The downside to this proposal is that clients that opted in expose parts of their browsing history, as the certificates they emit contain the websites the visited they as the subject.

**Modelling.** We add a rule for clients to report the certificates they encountered to a trusted monitor. As only clients that opt do this, the rule needs not to be used. Appropriately, the properties below talk about the guarantees for clients that participate in SCT auditing, but considering a model where not every client does that.

We add rules to the monitor that use incoming certificates and perform an authenticity check on them. As in the previous versions, the monitor relies for this on the ground truth it has from the actual key owner stored in the `!TrackSubject` fact. There are two possible outcomes of this check, modeled in a rule each: Either the monitor found a contradicting certificate, which does not match its ground truth knowledge, or the given certificate does not contradict the monitor's knowledge, i.e., it is authentic. Moreover, a dishonest monitor can find a rogue certificate, but *sleep on it*, meaning they neither inform the subject nor blame the CA. See Section I.1 for these rules.

**Accountable authenticity.** With SCT auditing, we use `Audit_CAFakesCert` as a case test and assume that every considered certificate in this statement is audited. Likewise as before, we distinguish on a second test whether the root CA or intermediate CA is blamed.

```
test ca_rogue_audit:
  "∃ idL rootCA [...].
    AuditCertFake(idCA, idL, CA_fields, stmt)@t0
    ∧ Ext_LegitimateCA(rootCA, CA_fields, idCA)@t1"

test root_ca_rogue_audit:
  "∃ idL idCA [...].
    AuditCertFake(idCA, idL, CA_fields, stmt)@t0
    ∧ IllegalCA(rootCA, idCA, CA_fields)@t1"
```

```
lemma A_CTAudit_1:
  ca_rogue_audit, root_ca_rogue_audit account for
  "∀ CA_fields stmt rootCA idL #t0 #t1.
    // For every certificate that is audited to the
    monitor
    AuditAuthCheck(idL, CA_fields, stmt)@t0
    ∧ LegitimateCheck(rootCA, CA_fields)@t1
    ⇒ (∃ idCA #t. GroundTruth(idCA, stmt)@t)"
```

This property holds; the monitor identifies misbehaving CAs correctly, even without trust in the loggers. Note, though, the monitor is assumed to have ground truth.

Moreover, the monitor can use the inclusion check to hold loggers accountable. We add a second test that blames the logger and the rogue CA if a rogue certificate is found while the inclusion check fails.

```
test ca_rogue_audit_logger_honest:
  "∃ CA_fields rootCA stmt idM idL commit
    session #t0 #t1 #t2 #t3.
    AuditCertFake(idCA, idL, CA_fields, stmt)
    @t0
    ∧ #t0 < #t1
    ∧ FetchLogSnapshot(idM, idL, session)@t1
    ∧ MonitorFindsCert(idM, idL, stmt, commit,
    session)@t2
    ∧ Ext_LegitimateCA(rootCA, CA_fields, idCA
    )@t3
    ∧ ProofRunsAfterMMD(idL, commit, session)"

test ca_rogue_audit_logger_rogue:
  "∃ CA_fields rootCA stmt idM commit session #
    t0 #t1 #t2 #t3.
    AuditCertFake(idCA, idL, CA_fields, stmt)
    @t0
    ∧ #t0 < #t1
    ∧ FetchLogSnapshot(idM, idL, session)@t1
    ∧ CannotFindCert(idM, stmt, idL, commit,
    session)@t2
    ∧ Ext_LegitimateCA(rootCA, CA_fields, idCA
    )@t3
    ∧ ProofRunsAfterMMD(idL, commit, session)
    ∧ not [ca_rogue_audit_logger_honest]"
```

We can now prove the property below. Monitors can correctly blame only the CA, or CA + logger, depending on the case. Cases can, of course, overlap, i.e., in the same trace,  $CA_1$  and  $\{CA_2, L_2\}$  trigger independent violations. This holds for premises that are audited and where the monitor was able (and did) run an inclusion proof after sufficient time relative to the MMD.

```
lemma A_CTAudit_2:
  ca_rogue_audit_logger_honest,
  ca_rogue_audit_logger_rogue,
  root_ca_rogue_audit_logger_honest,
  root_ca_rogue_audit_logger_rogue
  accounts for
  "∀ stmt CA_fields rootCA idM idL commit
    session #t0 #t1 #t2 #t3.
    AuditAuthCheck(idL, CA_fields, stmt)@t0
    /* Monitor receives a cert via
    auditing */
    ∧ #t0 < #t1
    ∧ FetchLogSnapshot(idM, idL, session)@t1
    ∧ MonitorInspectsLog(idM, idL, stmt,
    commit, session)@t2
    /* then after auditing looks for
    precisely that entry in the log */
    ∧ LegitimateCheck(rootCA, CA_fields)@t3"
```

$$\begin{aligned} & \wedge \text{ProofRunsAfterMMD}(\text{idL}, \text{commit}, \text{session}) \\ & \Rightarrow \\ & \vdash \text{stmt}(\text{t})'' \quad (\exists \text{ idCA } \#t. \text{GroundTruth}(\text{idCA}, \\ & \quad \text{idCA}, \text{t}))'' \end{aligned}$$

SCT Auditing mitigates the threat from malicious loggers hiding log entries. Assuming that *all clients* audit is a strong assumption, but a probabilistic check as implemented in Chrome is obviously realistic enough that it was deployed. The above lemma is easily raised to the probabilistic setting, as it specifies its guarantees per audited certificate. E.g., if Chrome clients audit a certificate with probability  $p$  and  $n$  rogue certificates are sent to such clients, then an attack carries the risk of detection with a probability of  $1 - (1 - p)^n$ .

We also trust that the monitor performs the audit and knows the ground truth. This was a realistic assumption in previous scenarios, where companies could run their own monitors. Here, however, monitors receive certificates from all clients that opted in, which is (a) a lot of load, (b) prohibitively slow for the clients if the monitor is not fast enough or has bad latency. Finally, (c), if each server would also act as a monitor and be trusted to do so, then we would have eliminated the needs for a PKI, let alone CT, altogether. It is thus not surprising that Chrome’s monitor is run by Google and trusted by users running Google’s web browser Chrome. Anyone can become a monitor, but not every monitor can perform SCT auditing.

**Transparency.** We adapt our notion of transparency to SCT auditing. Instead of stating the correctness of the log, we ensure the monitor can learn about an entry by finding it in the log or receiving it via auditing. We add `MonitorLearns` to all rules where the monitor learns an entry. If we assume auditing for a certificate, we immediately get that transparency for this certificate holds. (See Section J for more details.)

**Holding the monitor accountable with receipts.** So far, we excluded monitor misbehavior from our accountability analysis, as CT does not provide us with a mechanism suited for accountability tests. Other parties cannot verify whether the monitor purposefully ignores a rogue certificate or just did not receive it.

To see if that is possible at all, we extend SCT auditing (the only model actually managing to keep the logger accountable) with a rule that has honest monitors handover a signed receipt to the client, confirming the monitor has seen a particular snapshot of the log. In case of a dispute or suspicion, another role has the client forward this receipt to the domain owner (alternatively, the domain owner could act as the client for testing).

In case a rogue certificate is in active use and found by the domain owner, an additional test can detect the monitor’s misbehavior. While at that point, the domain owner could now just as well serve as an external validator like in section 4, the point here is that they can oversee the monitor service, which they contracted to do that for them. Moreover, SCT auditing works best if some powerful monitor receives many SCTs and certificates, which would be definition be

an external service. Combined with the previous tests we can now show accountable authenticity in the case where servers, CAs, loggers and monitors can be corrupted.

The model demonstrates thus a mechanism to also hold monitors achievable. But note that it neither represents a published proposal, nor is it thought out at that level of detail.

## 7. STH Gossiping

We now turn to a different, yet similar approach to the equivocation problem, STH Gossiping. While in SCT auditing, certificates had to be shared with the monitor to validate SCTs for the inclusion property, STHs can be validated for the append-only property, but without the certificate. This means STHs can be shared more widely: gossiping clients can share STHs with each other and verify append-only-ness between the snapshots they received and their own. To attack transparency, the logger must hide the entry from every gossiping client it interacts with, which prevents partition attacks on the gossiping clients. Direct communication is, however, impractical, since there is standard communication method between browsers, and end users are often difficult to address due to NAT.

Who should the STH then be gossiped to? Nordberg et al. [8] present various options, which can be classified by where the gossip is finally received and then analysed, which is either the auditor, or the monitor. We model specifically the case where STHs are sent to the monitor (comparable to *STH Pollination* [8]) or where the monitor doubles as a trusted auditor (*Trusted Auditor Relationship* [8]). Compared to *STH Pollination* [8] (or *SCT Feedback* [8] which is the same but additionally for SCTs), we assume direct communication between client and monitor, instead of a relay via the web server, which was implemented as the HTTP header [7], [30] but is now deprecated. As we do not trust the webserver, this abstraction comes w.l.o.g.

A client gossiping about its fetched STH can be accompanied by an inclusion proof of a certificate the client encountered. If that proof succeeds, the certificate is guaranteed to be included in the STH and thus more likely to become visible to the monitor. Sharing these STHs with the monitor results in a different trade-off for privacy. Instead of trusting the central monitor that clients share SCTs with as seen in SCT Auditing, with gossiping this trust is not needed and clients can share their STHs more broadly. However, as an inclusion proof is required to achieve the guarantee that the certificate is included in the STH, the client reveals the certificate of interest to the logger instead.

**Monitoring.** So far, we implemented inclusion proofs using facts and, by doing so, excluded the network attacker there. Our approach for modeling gossiping is to use the network again. Every commit (STH) handed out by a logger at some point will be sent out to the network. Monitors that rely on gossiping can retrieve them from the network.

We add authenticity checks to the monitor gossip rules, similarly to the authenticity checks in CT and CT with

SCT auditing. Both append-only and inclusion checks are implemented with a gossiping variant that considers an incoming STH. As we know, append-only checks are between two existing STHs. Later, when considering properties that include append-only checks, we will assume that one such STH has been gossiped (the older one), while the second STH is freshly fetched by the monitor that runs the append-only check. This way, a monitor can ensure that it can see at least every entry that a client has seen and gossiped to the monitor.

We use similar restrictions to axiomatically assert append-only proofs (Section E.6), but this time the gossiped proof is not necessarily handed out to the monitor but to any party to reflect the nature of gossiping.

For inclusion checks, in an earlier version, we simply reused the restrictions for the action facts `InclusionProofSucceeds` and `InclusionProofFails` where `commit` and `session` are replaced by the gossiped information.

This misses a simple attack that is now worth considering with the changed situation. So far, we assumed that handing out a commit STH to some party that contains an entry we are looking for means that the recipient will be able to find the entry using this STH. This assumption was fine, as we abstracted what exactly has been handed out by the logger, and the logger was in control to choose who to share the snapshot with. Recall that for gossiping, we are sharing only the STH of a snapshot retrieved by a client and not specific certificates. Let us consider a monitor that wants to fetch all entries that were included in an STH it retrieved via gossiping, i.e., open the commitment. If the logger is rogue, it could refuse to disclose its entries and hide potential rogue certificates. The monitor will be able to blame the logger, since this is clearly a violation, but it will not be able to learn the entries. To encode this kind of attack, we run the inclusion proof not on the gossiped commit but on a newly fetched snapshot from the log, allowing it to hide the entry again.

```

rule monitorCannotFindCert_Gossiped:
  let
    stmt = <iss, su, ke, ser>
  in
    [ !LoggerDict($IdM, $IdL, logcert), !
    ↪ St_Monitor($IdM), !Logger($IdL, pkL),
      !TrackSubject($IdM, su), In(stmt),
    ↪ RequestFetched(session),
      !TLS_C_to_M($idC, $idM, $gos_snapshot,
    ↪ gos_commit, gos_session),
      !TLS_L_to_M($idL, $idM, $snapshot, commit,
    ↪ session) ]
    [
      Gos_MonInclCheck($IdM, fields, gos_commit,
    ↪ gos_session, commit, session),
      Gossiped($IdM, $IdL, fields, <gos_commit,
    ↪ gos_session>),
      InclusionProofFails($IdL, fields, commit,
    ↪ session),
      /* certificate with fields is not visible
    ↪ in the presented snapshot */
      Gos_CannotFindCert($IdM, fields, commit,
    ↪ session, gos_commit, gos_session),
      Time($now)
  ]

```

```

  ]
  [ ]

```

If a monitor wants to ensure that it can see the entries behind a gossiped commit, it additionally needs to run an append-only proof between both snapshots it considers. We introduce a new append-only check to express an additional append-only check between a gossiped snapshot and a fetched snapshot by the monitor.

```

rule monitorFindsAppendOnlyViolation_Gossiped:
  [
    !LoggerDict($IdM, $IdL, logcert), !
    ↪ St_Monitor($IdM),
      !TLS_C_to_M($idC, $idM, $gos_snapshot,
    ↪ gos_commit, gos_session),
      !TLS_L_to_M($idL, $idM, $snapshot, commit,
    ↪ session)
  ]
  [
    Gos_AppendOnlyViol($IdM, $IdL, commit,
    ↪ session, gos_commit, gos_session),
    /* Append only is violated between
    ↪ both snapshots -> $IdL blamed */
    Gos_AppendOnlyCheck($IdM, $IdL, commit,
    ↪ session, gos_commit, gos_session),
    Time($i)
  ]
  ]
  [ ]

```

The corresponding restriction relaxes the assumption that both snapshots have been handed out to the monitor to reflect the idea of gossiping.

The other rules consider the opposite case and are left out here: If the certificate can be found, we distinguish between whether the certificate contradicts the known key for its subject or not, similar to the authenticity checks in Section H.1 or Section I.1 and blame the CA accordingly. We do not consider a sleep variant, where a monitor ignores found evidence in this section, assuming that every monitor involved in gossiping is honest.

**Accountable authenticity.** We modify authenticity to only provide guarantees for certificates that are gossiped to the monitor and for which some client has successfully proven inclusion of some certificate with fields.

```

lemma A_CTGossip_1:
  ca_rogue_gossip, root_ca_rogue_gossip accounts
  ↪ for
    "∀ idM idL stmt commit session rootCA
  ↪ CA_fields #t0 #t1 #t2.
    ClientSideInclusionProved(idL, stmt,
  ↪ commit, session)@t0
    ∧ Gossiped(idM, idL, stmt, commit, session
  ↪ )@t1
    ∧ LegitimateCheck(rootCA, CA_fields)@t2
    ⇒ (∃ idCA #t. GroundTruth(idCA, stmt
  ↪ )@t)"

```

The monitor fetches a new snapshot from the log in order to retrieve the same certificate and perform an authenticity check with it. As soon as the monitor sees the entry, it will either blame the CA if the certificate is rogue and add the `Gossiped_CAFakesCert` action fact to the trace, or, in the positive case, this action will not occur. Thus, we use `Gossiped_CAFakesCert` in the case test below

and further decide based on the legitimacy check whether to blame the root CA or intermediate CA.

```
test ca_rogue_gossip:
"∃ idL [...].
  ClientSideInclusionProved(idL, stmt, commit,
    session)@t0
  ∧ GossipedCertFake(idCA, idL, stmt, commit,
    session)@t1
  ∧ Ext_LegitimateCA(rootCA, CA_fields, idCA)@t2"
```

Accountable authenticity does not hold; we find an attack against verifiability. The monitor cannot blame the responsible CA, because without the entry, the monitor is unable to identify (let alone provide proof of misbehavior) the misbehaving CA. Note that, in contrast to CT, the monitor actually detects equivocation, as the received snapshot contradicts the old snapshot received as gossip earlier and thus the append-only proof between both STHs fails. We conclude that only sharing STHs with monitors is not sufficient to achieve accountable authenticity.

Note that, like with SCT Auditing, we require at least one trusted monitor. As SCT auditing leaks browsing history, STH gossiping can not only improve on privacy, but also weaken the trust assumption, as STH can be shared more broadly.

The underlying proposal for this version of gossiping has been dropped and is not used anymore, which makes the used tests less realistic. While this approach improves transparency and provides accountable transparency, it is not sufficient to show accountable authenticity. In Section N we consider accountability with a different security property that assumes a simple revocation model in CT and CT with SCT auditing.

**Transparency.** We find that transparency does not hold if the logger is misbehaving. If we assume that a client has a successful inclusion proof for a certificate in a given STH, then gossips this STH with the monitor, transparency becomes violated if the logger refuses to cooperate and disclose the entries behind that fetched STH to the monitor and instead equivocates. While transparency to the client holds in this scenario, the monitor does not gain a guarantee to achieve the same result if gossiping is used.

However, a monitor can detect this situation if it additionally checks for append-only. We derive a test that instead achieves accountable transparency.

```
test LoggerViolatesInclusion_monitor_gossip:
"∃ IdM [...].
  ClientSideInclusionProved(IdL, fields, <
    gos_commit, gos_session>)@t1
  /* client verified that fields is visible in
    gos_commit */
  ∧ SomeoneFetchesLogSnapshot(IdM, IdL, session)
  @t2
  /* monitor starts own inclusion proof */
  ∧ LoggerPresentsSnapshot(IdL, IdM, logid,
    time2, commit, session)@t3
  /* receives commit from logger */
  ∧ #t1 < #t2
  ∧ Gos_CannotFindCert(IdM, fields, commit,
    session, gos_commit, gos_session)@t4
```

```
/* but inclusion in this commit is not
  falsified */
  ∧ Gos_AppendOnlyViol(IdM, IdL, commit, session
    , gos_commit, gos_session)@t5"
/* the monitor finds that append-only between
  gos_commit and commit is
  violated and blames the logger */
```

Now we can show that accountable transparency holds. (See Section K for all lemmas in detail.)

## 8. Results

We summarize our results and provide details about our approach and the proof effort. All results were computed on an Intel 13th Gen Core i7-13700H with 10 allocated cores and 6GB of memory. The proof files are available at <https://anonymous.4open.science/api/repo/formal-model-certificate-transparency-6F77/file/ct.html?v=a43f4693>; links to lemmas in this document are clickable and lead to this document.

**Sanity Lemmas.** To show our models internally consistent we show that each rule is reachable, thus ensuring that security statements are non-vacuous. We prove 39 such lemmas, one for each rule. Each is proven automatically, but we sometimes add restrictions to limit the search space (only for sanity lemmas and other lemmas that check for satisfiability, e.g., sufficiency). Their total verification time is 226.05 seconds.

Proof details are given accumulated in Table 3 in Section B.

**Protocol mechanics.** To convince ourselves that we understand the mechanics of the protocol, we proved authenticity properties that found the trust between parties. Overall, these lemmas take less than a minute to prove in total.

**Lemma 1 ( cert\_auth ).** When a client accepts a certificate, then the CA asserts the value of this certificate, unless this particular CA was corrupted.

**Lemma 2 ( cert\_sct\_auth ).** If a CA asserts a public key certificate with two SCTs, then two distinct loggers have previously issued them honestly, unless some party was corrupted.

**Lemma 3 ( sct\_auth ).** If a client accepts a certificate with two SCTs then both loggers marked in their have added an entry to their log, unless one of them is corrupt.

**Lemma 4 ( incl\_proof\_auth ).** When a client accepts an SCT by some log, and later, after the MMD has passed, another party fetches a view that the log provides and the inclusion test between the SCT and the log fails, then that log must be compromised.

In Table 4 in Section B, we give the corresponding proof details of every lemma presented here.

**Accountability.** We summarize the results discussed in the previous sections in Table 2.

protocol	authenticity with trust in ...				transparency
	noone	monitor	+ logger	+ external validator	(no trust)
PKI	—	—	—	● (4)	—
CT	✗	✗ (H.2)	● (H.2)	● (4)	✗ (F, G)
SCT Auditing	✗ (with receipt: ●, ??)	● (G)	● (H.2)	● (4)	✓ (J)
Gossiping	✗	✗ (7)	● (H.2)	● (4)	● (L,M)

(✗ = attack   ● = accountable authenticity / transparency   ✓ = property proven   — = not applicable )

TABLE 2. RESULTS: ACCOUNTABILITY AND TRANSPARENCY IN CT.

**Transparency.** Our results show that transparency in CT is essential for achieving accountable authenticity. The mechanism of transparency—whether through an honest logger or a secondary source like SCT auditing—is less important than *what* is made transparent. To hold CAs accountable, log entries must include the full certificate, i.e., all fields signed by the CA; without them, loggers may be able to wrongly accuse CAs. More broadly, the data made transparent determines who can be held accountable: full certificates enable accountability of CAs, SCTs suffice to hold loggers accountable for omissions, and SCT auditing supports both, as it considers sharing SCTs and the full certificate. Gossiping reveals different STHs to monitors, which is comparable to SCT sharing, but without CA signatures, holding CAs accountable becomes impossible. As seen with gossiping, accountable transparency, as a weaker notion of transparency where detectable misbehavior is allowed, allows blaming loggers correctly but is not strong enough to achieve accountable authenticity.

**Applicability to current web browser.** In Section 2.1, we discussed how different browsers implement CT and extensions. SCT auditing, implemented in Chrome, leads to strong transparency and (accountable) authenticity guarantees, but only if we assume every certificate is audited. In practice, not every Chrome client is configured to do this and those that are only audit a randomized portion of certificates [10]. Nevertheless, we can state that every handshake that is actually audited enjoys these guarantees (acc\_CTAudit\_fakeCheck).

The other browsers do not implement SCT auditing and provide no client-side inclusion proofs. As we have seen, client-side inclusion proofs by themselves are not enough to guarantee transparency—and thus accountability without trust in at least one logger—as a dishonest logger can equivocate. All browser require multiple, say  $n$ , SCTs from distinct loggers (currently, for all of them the number is two) but these can be different for each certificate, albeit that they must come from a pool of  $N$  trusted loggers (see Table 1). Assuming the attacker knows the user’s browser, they can pick  $n$  dishonest loggers specifically, hence the trust assumption is that at least  $N - (n - 1)$  loggers are trusted (or at least, not colluding).

## 9. Related Work

So far, formal analyses of CT have either been conducted manually, in the cryptographic standard model [31], or with automated tools in the Dolev-Yao model but with significant simplifications [20], [26].

**Cryptographic analyses of CT.** Wrótniak et al. [31] modeled CT and conducted manual analysis to show transparency and accountability. They define accountability as a 23-line-long game in which the attacker wins if they are able to produce a rogue certificate that passes validation but does not point to a *responsible* CA, i.e., a trusted root CA that has signed the rogue certificate’s fields or issued a certificate for some intermediate CA that is responsible in the same sense. They show that this notion of accountability holds for CT.

First, this cryptographic game is not only specific to accountability for PKI schemes but also specific to their formulation, and encodes protocol-specific concepts like root certificates and root CAs, specific fields in certificates, and even the data flow of the scheme. This definition is not easy to adapt to formal analysis outside the cryptographic model, as it is game-based and very complex.

Secondly, because the game requirement is so specific, it is not clear if it achieves accountability in a broader sense. Comparing with the definition we employ, their definition at most guarantees that some responsible party can be identified. It does not ensure that this party indeed bear responsibility nor that everyone else who should bear responsibility is identified. Indeed, there is not even a mechanism to blame other parties than CAs and only one. Many works on accountability in protocol-like settings highlight the ability to blame (all) the responsible participants [32], [33], [34].

By contrast, our definition of accountability was tested on OCSP-Stapling and Mixnets [20] and can ensure all parties blamed a causes to the violation and all such causes are indeed blamed. For this reason, we can find that accountability only holds under strong assumptions. Nevertheless, their work was a motivation for the present analysis and helped formulate some tests (see below).

**Automated analysis of accountability in CT.** CT was first verified for accountability by Bruni et al. [24]. Morio and Künnemann [20] extend the model and analyze it with respect to the same accountability definition we use here, pointing out that their definition allows blaming a party for

a violation of a security property that is always true, i.e., has not been violated by any party and may miss violations. We do not further extend their model but instead obtain a broader model that captures more of the details of the actual specification while continuing to work with support for an unbounded number of participants.

Cheval et al. [26] analysed CT in ProVerif. Their focus is on accurately modelling the Merkle tree structure and they extend ProVerif to support an axiomatic modeling of ledgers. Besides CT, ledgers are also used in voting systems [35]. They first show the Merkle tree structure secure and then, assuming a Merkle tree interface, the properties of CT.

For instance, they model the proof of inclusion (they call this a proof of presence) using inductive clauses. They allow computing the proof recursively starting from the leaf that contains the entry we are interested in, whose proof would be the empty list. Then, depending on whether the right or left child node is missing, the hash of that node is appended to the proof list together with a label `left` or `right`, indicating the branch taken to traverse the tree from the root to the leaf we started at.

Our trace-based model of the log (Section E) also allows for equivocation and representing proofs as messages that can be sent around but builds on restrictions that axiomatically assert the properties of the log in relation to inclusion or append-only proofs. We do not prove those axioms.

Despite the more detailed modeling of Merkle trees, their model of CT is simpler than ours in arguably more pertinent aspects: they only consider browser-side validation (in a simplified manner) and browser-side monitoring (which none of the popular browsers implement). I.e., neither the loggers nor the monitors appear as parties distinct from the client or website and hence cannot be individually corrupted.

Summarizing, for accountability we had to design our model from scratch. Moreover, we are the first to model time to represent the certificate validity period and represent the pre-certification process.

**SCT Auditing.** Wrótniak et al. [31]’s model also considers SCT auditing [31, CTwAudit, Sec. D.2] (see Section 6) but in form of an algorithm that given a certificate with SCTs returns the identity of the loggers that are deemed responsible for the lack of transparency if the corresponding entry is not visible in the log. They show that auditing can be used to achieve "audited transparency," i.e., either transparency holds or they are able to blame the responsible logger if transparency is violated. Similarly, we can show accountable transparency (for STH gossiping) or even transparency itself (for SCT auditing) under lighter assumptions. Our accountability tests for transparency in STH Gossiping are similar to their algorithm for audited transparency, but they strictly focus on transparency with respect to the log. Instead of using the audited certificate in order to learn an entry, their algorithm ‘magically’ hands a certificate to the monitor, who has a snapshot from *every* log, and compares it with the existing knowledge. We do not assume that a monitor stores every certificate it has ever seen but instead

fetches another snapshot from a log and blame them if inclusion is violated.

## 10. Conclusion

We discussed the authenticity problem in PKIs that rely on a central authority. Our results support that CT solves this problem and successfully delegates the required trust from CAs to loggers. This helps—we gave the Symantec CA misbehavior as an example in Section 2. By logging the whole certificate and chain, log entries provide enough evidence to hold CAs accountable for their issued certificates. If the logger is corrupted and violates transparency, the evidence can stay hidden, which constitutes to a known attack against accountable authenticity. If one of the used loggers is honest, transparency for that particular logger holds. Assuming an honest monitor that knows the ground truth (every domain owner can monitor themselves), we can hold the CA accountable for maliciously issued certificates.

Assuming an honest logger is not as benign as it looks and requires further research from the policy perspective. The current pool of loggers consists of 7-8 logs (see Table 1) with almost complete intersect (currently, IPnG is included in some but not all). With Chrome’s high market share, there is no incentive for websites to include other logs (certificates are transmitted before the user agent is known) let alone target loggers that are not trusted in Chrome. Essentially, Google decides which set of loggers is trusted, even for other browsers, and is included in this set.

With SCT auditing, implemented in Chrome, we can remove the log from the trust assumption and instead hold misbehaving loggers and CAs simultaneously accountable. We find, however, that this strong guarantee only holds for certificates that are audited; for other certificates, loggers can get away with hiding the corresponding entry. Certificates are only audited randomly and if the client uses Chrome and opted in, thereby sharing them with one specific, trusted Google monitor. We discussed the privacy implications in Section 6. On the policy-level, it is worthwhile to explore how to counter-act the inherent tendency to centralise

With STH gossiping, not implemented and deprecated, clients also share their log snapshot, but they additionally perform an inclusion proof with the corresponding logger. This adds further cost and effectively eliminates the privacy advantage gained from sharing only the log snapshot instead of full certificates. Future work could explore privacy preserving primitives to compute inclusion proofs. Only accountable transparency holds; loggers can refuse to show the entries corresponding to an STH they have handed out but this misbehavior is identifiable. The same attack violates accountable authenticity.

Our model has significant improvements compared to existing works, most notably the approach to Merkle proofs and that it covers dishonest intermediary CAs, root CAs, loggers and monitors. It currently does not cover certificate revocation. While CT does not oblige logs to store revocation request, researchers are considering integrating revocation proofs into the CT log, e.g., via postcertificates [36]

or similar entries [37] stored in the log that can prove a certificate was presented after being revoked. Vice versa, if revocation works, it also helps CT: if a monitor detects a rogue certificate, the most immediate reaction should be to revoke it. Accountability is as necessary for revocation as it is for certificate authenticity: Morio and Künnemann [20] showed that OCSP-Stapling (RFC 6066) can provide accountability, but only if the OCSP responder is trusted. In Section N we show that CT benefits from certificate revocation, albeit in a simple model: should a certificate be revoked at the moment its subject recognizes it as rogue (via the monitor) and all monitors be immediately notified of this fact, then the monitor, log and server can be held accountable for the authenticity of the certificate, including revocation status—provided, of course, that either the relevant log is honest or SCT Auditing is used to ensure transparency.

## References

- [1] mozilla. “Revoking Trust in Two TurkTrust Certificates,” Mozilla Security Blog, Accessed: Oct. 9, 2025. [Online]. Available: <https://blog.mozilla.org/security/2013/01/03/revoking-trust-in-two-turktrust-certificates>.
- [2] B. Laurie, A. Langley, and E. Kasper, “Certificate Transparency,” RFC Editor, 6962, Jun. 2013, RFC 6962. DOI: 10.17487/RFC6962. [Online]. Available: <https://www.rfc-editor.org/info/rfc6962>.
- [3] B. Laurie, E. Messeri, and R. Stradling, “Certificate Transparency Version 2.0,” RFC Editor, 9162, Dec. 2021, RFC9162. DOI: 10.17487/RFC9162. [Online]. Available: <https://www.rfc-editor.org/info/rfc9162>.
- [4] S. Boeyen, S. Santesson, T. Polk, R. Housley, S. Farrell, and D. Cooper, “Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile,” Internet Engineering Task Force, Request for Comments RFC 5280, May 2008, 151 pp. DOI: 10.17487/RFC5280. Accessed: Jul. 17, 2025. [Online]. Available: <https://datatracker.ietf.org/doc/rfc5280>.
- [5] N. Blagov, “State of the Certificate Transparency Ecosystem,” 2020. DOI: 10.2313/NET-2020-11-1\_09. Accessed: Jun. 30, 2025. [Online]. Available: [https://www.net.in.tum.de/fileadmin/TUM/NET/NET-2020-11-1/NET-2020-11-1\\_09.pdf](https://www.net.in.tum.de/fileadmin/TUM/NET/NET-2020-11-1/NET-2020-11-1_09.pdf).
- [6] “Merkle Town,” Accessed: Jul. 20, 2025. [Online]. Available: <https://ct.cloudflare.com/>.
- [7] O. Gasser, B. Hof, M. Helm, M. Korczynski, R. Holz, and G. Carle, “In Log We Trust: Revealing Poor Security Practices with Certificate Transparency Logs and Internet Measurements,” in *Passive and Active Measurement*, R. Beverly, G. Smaragdakis, and A. Feldmann, Eds., vol. 10771, Cham: Springer International Publishing, 2018, pp. 173–185, ISBN: 978-3-319-76480-1 978-3-319-76481-8. DOI: 10.1007/978-3-319-76481-8\_13. Accessed: Jun. 30, 2025. [Online]. Available: [http://link.springer.com/10.1007/978-3-319-76481-8\\_13](http://link.springer.com/10.1007/978-3-319-76481-8_13).
- [8] L. Nordberg, D. K. Gillmor, and T. Ritter, “Gossiping in CT,” Internet Engineering Task Force, Internet Draft (Expired) draft-ietf-trans-gossip-05, Jan. 14, 2018, 57 pp. Accessed: Apr. 2, 2025. [Online]. Available: <https://datatracker.ietf.org/doc/draft-ietf-trans-gossip>.
- [9] “Chrome Certificate Transparency Policy,” Certificate Transparency, Accessed: May 27, 2025. [Online]. Available: [https://googlechrome.github.io/CertificateTransparency/ct\\_policy.html](https://googlechrome.github.io/CertificateTransparency/ct_policy.html).
- [10] C. Thompson and E. Stark, “Opt-in SCT Auditing (public),” Sep. 30, 2020. [Online]. Available: <https://docs.google.com/document/d/1G1Jy8LJgSqJ-B673GnTYIG4b7XRw2ZLtvvSlrqFcl4A/>.
- [11] “SecurityEngineering/Certificate Transparency - MozillaWiki,” Accessed: Apr. 2, 2025. [Online]. Available: [https://wiki.mozilla.org/SecurityEngineering/Certificate\\_Transparency](https://wiki.mozilla.org/SecurityEngineering/Certificate_Transparency).
- [12] “Apple’s Certificate Transparency policy,” Apple Support, Accessed: May 27, 2025. [Online]. Available: <https://support.apple.com/en-us/103214>.
- [13] “TLS Policy,” GitHub, Accessed: May 27, 2025. [Online]. Available: <https://github.com/brave/brave-browser/wiki/TLS-Policy>.
- [14] *SSLMate/certspotter*, SSLMate, Jul. 8, 2025. Accessed: Jul. 9, 2025. [Online]. Available: <https://github.com/SSLMate/certspotter>.
- [15] C. Project. “Monitors : Certificate Transparency,” Accessed: Jul. 9, 2025. [Online]. Available: <https://certificate.transparency.dev/monitors/>.
- [16] R. Sleevi. “Sustaining Digital Certificate Security,” Google Online Security Blog, Accessed: Jul. 15, 2025. [Online]. Available: <https://security.googleblog.com/2015/10/sustaining-digital-certificate-security.html>.
- [17] D. O’Brien, R. Sleevi, and A. Whalley. “Chrome’s Plan to Distrust Symantec Certificates,” Google Online Security Blog, Accessed: Jul. 15, 2025. [Online]. Available: <https://security.googleblog.com/2017/09/chromes-plan-to-distrust-symantec.html>.
- [18] “Certificate Transparency - Security on the web | MDN,” Accessed: Jun. 2, 2025. [Online]. Available: [https://developer.mozilla.org/en-US/docs/Web/Security/Certificate\\_Transparency](https://developer.mozilla.org/en-US/docs/Web/Security/Certificate_Transparency).
- [19] S. Meier, B. Schmidt, C. Cremers, and D. Basin, “The TAMARIN Prover for the Symbolic Analysis of Security Protocols,” in *Computer Aided Verification*, N. Sharygina and H. Veith, Eds., red. by D. Hutchison et al., vol. 8044, Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 696–701, ISBN: 978-3-642-39798-1 978-3-642-39799-8. DOI: 10.1007/978-3-642-39799-8\_48. Accessed: Jul. 2, 2025. [Online]. Available: [http://link.springer.com/10.1007/978-3-642-39799-8\\_48](http://link.springer.com/10.1007/978-3-642-39799-8_48).
- [20] K. Morio and R. Künnemann, “Verifying Accountability for Unbounded Sets of Participants,” in *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*, Dubrovnik, Croatia: IEEE, Jun. 2021,

- pp. 1–16, ISBN: 978-1-7281-7607-9. DOI: 10.1109/CSF51468.2021.00032. Accessed: Mar. 18, 2025. [Online]. Available: <https://ieeexplore.ieee.org/document/9505190/>.
- [21] J. Y. Halpern. “A Modification of the Halpern-Pearl Definition of Causality.” arXiv: 1505.00162 [cs], Accessed: Jul. 10, 2025. [Online]. Available: <http://arxiv.org/abs/1505.00162>, pre-published.
- [22] “Common CA Database by the Linux Foundation,” Accessed: Oct. 14, 2025. [Online]. Available: <https://www.ccadb.org/policy#5-audit-disclosures>.
- [23] K. Morio, I. Esiyok, D. Jackson, and R. Künnemann, “Automated Security Analysis of Exposure Notification Systems,” in *32nd USENIX Security Symposium (USENIX Security 23)*, USENIX Association, Aug. 2023, pp. 6593–6610. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity23/presentation/morio>.
- [24] A. Bruni, R. Giustolisi, and C. Schuermann, “Automated Analysis of Accountability,” in *Information Security*, P. Q. Nguyen and J. Zhou, Eds., vol. 10599, Cham: Springer International Publishing, 2017, pp. 417–434, ISBN: 978-3-319-69658-4 978-3-319-69659-1. DOI: 10.1007/978-3-319-69659-1\_23. Accessed: Apr. 2, 2025. [Online]. Available: [https://link.springer.com/10.1007/978-3-319-69659-1\\_23](https://link.springer.com/10.1007/978-3-319-69659-1_23).
- [25] R. Künnemann, I. Esiyok, and M. Backes. “Automated Verification of Accountability in Security Protocols.” arXiv: 1805.10891 [cs], Accessed: Mar. 18, 2025. [Online]. Available: <http://arxiv.org/abs/1805.10891>, pre-published.
- [26] V. Cheval, J. Moreira, and M. Ryan. “Automatic verification of transparency protocols (extended version).” arXiv: 2303.04500 [cs], Accessed: Mar. 17, 2025. [Online]. Available: <http://arxiv.org/abs/2303.04500>, pre-published.
- [27] S. A. Crosby and D. S. Wallach, “Efficient Data Structures for Tamper-Evident Logging,”
- [28] O. Gasser, B. Hof, M. Helm, M. Korczynski, R. Holz, and G. Carle, “In log we trust: Revealing poor security practices with certificate transparency logs and internet measurements,” in *Passive and Active Measurement*, R. Beverly, G. Smaragdakis, and A. Feldmann, Eds., Cham: Springer International Publishing, 2018, pp. 173–185, ISBN: 978-3-319-76481-8.
- [29] “SCT Auditing Google Source,” Google. [Online]. Available: [https://chromium.googlesource.com/chromium/src/+refs/heads/main/services/network/sct\\_auditing/](https://chromium.googlesource.com/chromium/src/+refs/heads/main/services/network/sct_auditing/).
- [30] “Expect-CT header - HTTP | MDN,” Accessed: Jun. 30, 2025. [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers/Expect-CT>.
- [31] S. Wrótniak, H. Leibowitz, E. Syta, and A. Herzberg, “Provable Security for PKI Schemes,” in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, Salt Lake City UT USA: ACM, Dec. 2, 2024, pp. 1552–1566, ISBN: 979-8-4007-0636-3. DOI: 10.1145/3658644.3670374. Accessed: Mar. 19, 2025. [Online]. Available: <https://dl.acm.org/doi/10.1145/3658644.3670374>.
- [32] J. Feigenbaum, A. D. Jaggard, and R. N. Wright, “Open vs. closed systems for accountability,” in *Proceedings of the 2014 Symposium and Bootcamp on the Science of Security*, Raleigh North Carolina USA: ACM, Apr. 8, 2014, pp. 1–11, ISBN: 978-1-4503-2907-1. DOI: 10.1145/2600176.2600179. Accessed: Apr. 4, 2025. [Online]. Available: <https://dl.acm.org/doi/10.1145/2600176.2600179>.
- [33] H. Chockler and J. Y. Halpern. “Responsibility and blame: A structural-model approach.” arXiv: cs/0312038, Accessed: Apr. 8, 2025. [Online]. Available: <http://arxiv.org/abs/cs/0312038>, pre-published.
- [34] R. Küsters, T. Truderung, and A. Vogt, “Accountability: Definition and relationship to verifiability,” in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, Chicago Illinois USA: ACM, Oct. 4, 2010, pp. 526–535, ISBN: 978-1-4503-0245-6. DOI: 10.1145/1866307.1866366. Accessed: Apr. 8, 2025. [Online]. Available: <https://dl.acm.org/doi/10.1145/1866307.1866366>.
- [35] V. Cortier, J. Lallemand, and B. Warinschi, “Fifty Shades of Ballot Privacy: Privacy against a Malicious Board,” in *2020 IEEE 33rd Computer Security Foundations Symposium (CSF)*, Jun. 2020, pp. 17–32. DOI: 10.1109/CSF49147.2020.00010. Accessed: Jul. 10, 2025. [Online]. Available: <https://ieeexplore.ieee.org/document/9155128/>.
- [36] N. Korzhitskii, M. Nemec, and N. Carlsson. “Postcertificates for Revocation Transparency.” arXiv: 2203.02280 [cs], Accessed: Oct. 13, 2025. [Online]. Available: <http://arxiv.org/abs/2203.02280>, pre-published.
- [37] J. Kong, D. James, H. Leibowitz, E. Syta, and A. Herzberg. “CTng: Secure Certificate and Revocation Transparency,” Accessed: Oct. 13, 2025. [Online]. Available: <https://eprint.iacr.org/2021/818>, pre-published.
- [38] “Baseline Requirements for the Issuance and Management of Publicly-Trusted TLS Server Certificates,” May 16, 2025.

## Appendix

### 1. PKI model

We start with the PKI roles: domain owners (servers), CAs, and clients.

A DomainOwner or the server represents the honest case: a web server with an identity and generated key pair.

```
rule Domain_Owner_Init:
  let pubK = pk(~sk) in
  [ Fr(~sk) ] -[ Time($now) ]->[ !DomainOwner(
    ↳ $IdServer, pubK), Out(pubK) ]
```

The public key `pubK` is additionally shared on the network (with the adversary) to reflect its nature. The secret key will never be used as we only focus on certificates and do not perform a handshake during the protocol that uses the transparent certificate. The `!DomainOwner` fact will, in the next step, be handed to the Certificate Authority to request a certificate.

Registration of a CA also includes generating a key pair, where the public key is sent to the network. Additionally, for (root) CAs, they create self-certificates that include their identity, key, and serial number. This certificate will become a part of the chain of trust for every certificate it issues in the future.

```
rule CA_Init:
  let
    tbs = <$IdCA, $IdCA, pk(~skCA), 'SelfCert'
  < , true(), ~serial, $i, $j>
    self_sig = sign(tbs, ~skCA)
  in
    [ Fr(~skCA), Fr(~serial) ]
    -[ Time($i) ]→
    [ Out(tbs), Out(self_sig), Out(<$IdCA, pk(~
  < skCA)>),
      /* publicly announce the public key and
  < certificate on the network */
      !CA_s0($IdCA, ~skCA, pk(~skCA), tbs,
  < self_sig),
      !CA($IdCA, pk(~skCA), tbs, self_sig) ]
```

After receiving the request, the CA responds by issuing a certificate for the identity and key pair provided by the `!DomainOwner` fact. Its usage represents an ownership check performed by the CA, which reflects honest CA behavior. This rule represents the CA's behavior, yet it contains `!DomainOwner` which must come from `Domain_Owner_Init`. This is used to represent the mandatory [38] identity checks the CA has to perform, i.e., we assume every honest user to accurately present the identity and each honest CA to verify it. We will present corruption rules for both later.

```
rule Certify:
  let
    cert_tbs = <$IdCA, $Id, pkID, 'PubKey',
  < false(), ~serial, $i, $j>
    cert_sig = sign(cert_tbs, skCA)
    cert = <cert_tbs, cert_sig>
  in
    [ In(<$Id, pkID >), !DomainOwner($Id, pkID),
      /* public key to sign. Using this rule
  < includes an "identity check"
  < which is simplified: There is a user fact with
  < this public key. */
      !CA_s0($IdCA, skCA, pkCA, CA_tbs, CA_sig), Fr
  < (~serial) ]
    -[ Certified($IdCA, cert),
      Asserts_PKI($IdCA, <$IdCA, $Id, pkID, ~
  < serial>),
      ServersRealCert(cert),
      Time($i) ]→
    [ Out(cert_tbs), Out(cert_sig) /* outputs cert
  < with signature */ ]
```

It will announce both the issued certificate along with its self-signed certificate. Together, they represent the certificate

chain that is necessary to verify the signature on each certificate. We consider three levels: root CAs (whose public key the client knows), intermediate CAs and end-entities like web servers.

Clients then validate the incoming certificate they encounter. Clients represent users of a browser that visit the website of `subject` given in the certificate; their identity is represented as a public variable. Throughout, we assume the Dolev-Yao adversary. We add further capabilities by adding key leakage for CAs. This is modeled by a rule that takes the persistent role fact and outputs the included secret key to the network.

```
rule compromiseCA:
  [ !CA_s0($IdCA, skCA, pkCA, CA_tbs, CA_sig) ]
  -[ Corrupted($IdCA),
      Time($i) ]
  ]→
  [ Out(skCA) ]
```

## 2. Additional verification results times

## 3. Replacement Property

For every accountability lemma the Replacement Property (RP) needs to be manually verified if we use public constants or restrictions in our model. We heavily use both, thus we provide a brief argument that RP is not violated in our model. The property states that for every single-matched trace of a case test, the instantiation of free parties can be replaced by any other party which is allowed according to our model.

Our model uses public constants as type identifier for different certificates. By design, these public constants are not used as identifier for different roles and for each role initialization, we restrict, that the used public variable is unique among different roles.

We use in total 31 restrictions in our model. 12 of these restrictions are inline and concerned comparing public keys or SCTs, they do not affect role identifiers and thus do not violate the RP condition. Two more inline restrictions state that the two loggers considered in SCTs must be distinct. They remain distinct under bijective renaming. Six other restrictions axiomatically assert the Merkle proofs, not restricting the identities of involved parties. We restrict that CAs, loggers and domain owners only register once to ensure that they only have one public key. Additionally, the public variable cannot be reused for different roles, including monitors. These unique registrations remain distinct under bijective renaming. Three more restrictions concern the uniqueness of log identifiers. Recall that these log identifiers are distinct from the identifiers of loggers. The remaining four restrictions are for equality, time axioms, and consistent assertions used in the external check. All of them do not influence role identifiers. Thus our restrictions and use of public constants do not violate RP and our analysis results are valid.

type	model	assumption details	proof details			
			interaction	verif.	time (s)	# steps
san.	PKI		—	✓	1.28	25
	CT		—	✓	151.84	413
	Audit		—	✓	18.01	50
	Gossip		—	✓	35.96	123
T	CT	w/o MMD	yes	✗	—	—
		with MMD	yes	✗	—	—
		honest logger	—	✓	8.60	148
	Gossip		—	✓	3.42	8
		w/o client proof	yes	✗	—	—
		with client proof	yes	✗	—	—
AT	CT	clientside	SUFF	✓	35.41	413
		monitor with client	SUFF	✓	23.78	213
		monitor alone	—	✗	26.09	97
	Gossip		SUFF	✓	20.06	167
			SUFF*,+	✓	12.86	26
			—	✓	23.82	50
AA	PKI Ext.	external validator	—	✓	27.92	58
		monitor validator	—	✓	27.92	58
	CT Ext.	client blames log	SUFF	✓	167.72	1736
		monitor blames log	SUFF*	✓	205.88	1279
		monitor fetches log	—	✗	64.50	416
		m. fetches benign log	—	✓	204.12	1589
	Audit		—	✓	32.18	120
		added logger blame	SUFF*	✓	204.12	1589
	Gossip		SUFF,EMP	✗	120.88	997
			—	—	—	—

TABLE 3. WE REPORT PROOF DETAILS OF ALL SANITY PROPERTIES (SAN.), TRANSPARENCY (T), ACCOUNTABLE TRANSPARENCY (AT), AND ACCOUNTABLE AUTHENTICITY (AA) EACH GROUPED BY MODEL AND FURTHER ASSUMPTIONS. THE INTERACTION COLUMN DENOTES IF USER INTERACTION IS REQUIRED FOR THE FULL PROOF (YES). FOR ACCOUNTABILITY PROPERTIES, WE HIGHLIGHTED WHEN SUFFICIENCY (SUFF), INCLUDING SINGLE-MATCHED OR EMPTY (EMP) IS SHOWN WITH USER INTERACTION. (\*) DENOTES THAT INSTEAD A STRONGER CLAIM IS SHOWN. IF HELPER LEMMAS ARE REQUIRED, WE DENOTE THIS WITH A + AND PRESENT CORRESPONDING PROOF DETAILS IN ??. ✓ DENOTES THAT THE PROPERTY IS VERIFIED, WHILE ✗ INDICATES THAT WE HAVE FOUND AN ATTACK. PROOF TIMES AND NUMBER OF REQUIRED STEPS ARE ACCUMULATED FOR SANITY LEMMAS AND ACCOUNTABILITY PROPERTIES, WHERE MANUAL INTERACTION STEPS ARE NOT REPORTED.

Lemma	proof details	
	time (s)	# steps
CA_auth	8.0	14
Loggers_auth	8.3	12
validity_guarantees_logging	7.9	11
InclusionProofViolation_Guarantee	12.3	96

TABLE 4. WE PRESENT THE REQUIRED TIME AND NUMBER OF STEPS TO AUTOMATICALLY VERIFY EACH LEMMA LISTED HERE.

## 4. Modelling CT (Details)

**4.1. Adding Loggers and Monitors.** We introduce loggers and monitors similarly to the other roles. Both roles have an initialization rule. As the monitor does not have its own key pair, it only consists of a public variable representing its identity. Loggers register a public key along with a self-signed certificate. Their key is later used to issue and sign SCTs. The main purpose of logs is to maintain the log structure and hand out commitments and proofs for inspecting parties. The details describing the log model in detail are given in Section E.

```
rule Monitor_Init:
  [ ] ⊢ Time($i) ⊢ [ !St_Monitor($IdM) ]
```

```
rule Logger_Init:
  let
```

```
tbs = <$IdL, $IdL, pk(~skL), 'SelfCert',
  false(), ~serial, $i, $j>
sig = sign(tbs, ~skL)
in
  [ Fr(~skL), Fr(~serial) ]
  ⊢ [ Time($i) ] ⊢
  [ Out(tbs), Out(sig), !Logger_s($IdL, ~skL, pk
    (~skL)),
    !Logger($IdL, pk(~skL)) ]
```

We model the monitor check by adding a rule that represents a monitor starting to observe a logger and a second rule that represents the details in ?? of tracking a specific subject.

```
rule StartMonitoring:
```

```
let
  logcert_tbs = <$IdL, $IdL, pkL, 'SelfCert'
  false(), serial, $i, $j>
in
  [ In(logcert_tbs), In(logcert_sig), /*
    incoming logger self-cert */
    !St_Monitor($IdM), !Logger($IdL, pkL) ]
  ⊢ [
    Eq(verify(logcert_sig, logcert_tbs, pkL),
      true()),
    Time($now)
  ] ⊢
  [ !LoggerDict($IdM, $IdL, <logcert_tbs,
    logcert_sig>) ]
```

```
rule monitorTrackSubject:
```

```

    [ !St_Monitor($IdM), !DomainOwner_Truth(su,
    pubK) ]
    [ !TrackSubject($IdM, su, pubK) ]

rule PreCertify: /* CA issues a pre-certificate */
let
    preC_tbs = <$IdCA, $Id, pkID, 'pre-cert',
    false(), ~serial, $i, $j>
    preC_sig = sign(preC_tbs, skCA)
    preC = <preC_tbs, preC_sig>
in
    [ In(<$Id, pkID>), !DomainOwner($Id, pkID), /*
    represents identity check */
    !CA_s0($IdCA, skCA, pkCA, CA_tbs, CA_sig), Fr
    (~serial) ]
    [ PreCertified($IdCA, preC),
    Asserts_PKI($IdCA, <$IdCA, $Id, pkID, ~
    serial>), /* external check */
    ServersRealCert(<$IdCA, $Id, pkID, ~serial
    >),
    ServersRealKey(<$Id, pkID>),
    Time($i) ]→
    [
    Out(preC_tbs), Out(preC_sig),
    Certificate_Store($IdCA, preC),
    /* outgoing pre-certificate */
    !HonestlyIssued(preC), !DomainOwner_Truth(
    $Id, pkID)
    /* ground truth facts */
    ]

/* CA embeds both SCTs in the pre-cert */
rule PubKeyCertify:
let
    tbs_preC = <$IdCA, $Id, pkID, 'pre-cert',
    false(), ser, $i, $j>
    tbs_sct1 = <$IdL1, 'SCT', $sct1, <$IdCA, $Id,
    pkID, false(), ser, $i, $j>>
    tbs_sct2 = <$IdL2, 'SCT', $sct2, <$IdCA, $Id,
    pkID, false(), ser, $i, $j>>
    sct1 = <$IdL1, 'SCT', $sct1, sct1_sig>
    sct2 = <$IdL2, 'SCT', $sct2, sct2_sig>
    tbs_pub = <$IdCA, $Id, pkID, 'PubKey', false()
    , ser, $i, $j, sct1, sct2>
    pub_sig = sign(tbs_pub, skCA)
    pubcert = <tbs_pub, pub_sig>
in
    [
    !CA_s0($IdCA, skCA, pkCA, CA_tbs, CA_sig),
    In(sct1), In(sct2), /* 2 SCTs */
    !Logger($IdL1, pkL1), !Logger($IdL2, pkL2)
    ,
    Certificate_Store($IdCA, <tbs_preC,
    sig_preC>)
    ]
    [ Eq(verify(sct1_sig, tbs_sct1, pkL1), true())
    ,
    Eq(verify(sct2_sig, tbs_sct2, pkL2), true
    ()) ,
    Eq(verify(sig_preC, tbs_preC, pkCA), true
    ()) ,
    _restrict( ~( $IdL1 = $IdL2 ) ), /* 2 SCTs
    were from distinct loggers */
    Certified($IdCA, $IdL1, $IdL2),

```

```

    CAAsserts($IdCA, pubcert, sct1, sct2),
    Time($now),
    CheckValidUntil($now, $j)
    ]→
    [
    Out(tbs_pub), Out(pub_sig)
    /* outgoing PubKey cert */
    ]

```

## 5. Log Data Structure

**5.1. Using the trace as ledger.** We model logs maintained by every logger exclusively as property on the trace. We add actions to rules to make additions to the log, as well as generation, and verification of proofs visible. Then we use trace restrictions to enforce relations between these. This is similar to the model of time and space in [23]. For example, the action `LoggerComputesAddLog($LogIdentity` `l` , `cert`) represents additions to the log maintained by `$LogIdentity`.

A similar approach has been used before by Bruni et al. [24] and extended by Künnemann et al. [25] to model CT's public ledgers. Both capture the partition attack by allowing the logger to present different snapshots to different auditors. Inclusion or append-only proofs were not part of both models, and we show that the approach can be further extended to abstractly capture properties of the Merkle tree structure, including append-only proofs and inclusion proofs.

An honest logger accepts an incoming pre-certificate only if it has a full chain that ends with a root certificate. As we only consider root CAs, this chain is of length 1. Then, it issues the corresponding SCT and adds the chain and pre-certificate to its publicly visible log.

```

rule add_preC_to_log:
let
    preC_tbs = <$IdCA, $Id, pkID, 'pre-cert',
    false(), ser, $i, $j>
    /* stripped signature */
    preC = <preC_tbs, sig_preC>
    preC_fields = <$IdCA, $Id, pkID, ser>
    sct_tbs = <$IdL, 'SCT', $now, <$IdCA, $Id,
    pkID, false(), ser, $i, $j>>
    sct_sig = sign(sct_tbs, skL)
    sct = <$IdL, 'SCT', $now, sct_sig>
in
    [
    In(preC_tbs), In(sig_preC), /* incoming
    pre-cert */
    !Logger_s($IdL, skL, pk(skL)),
    !CA($IdCA, pk_CA, tbs_self, sig_self),
    !Log($IdL, $log_id)
    ]
    [
    Eq(verify(sig_preC, preC_tbs, pk_CA), true
    ) ,
    Logged($IdL, <preC, <tbs_self, sig_self>>)
    ,
    /* Logger stores the full chain in the
    log */

```

```

    LoggerComputesAddLog($IdL, $log_id,
  ↳ preC_fields),
    LoggerComputesAddLogTime($IdL, $log_id,
  ↳ preC_fields, $now),

    Time($now),
    CheckValidUntil($now, $j)
  ↳
  ↳ [ Out(sct), KeepLogged(<$IdL, skL, pk(skL)>,
  ↳ $log_id, preC) ]

```

**5.2. Corrupted Loggers.** Like for CAs, we add a key leakage rule for the logger. This allows the network attacker to construct forged SCTs. To allow manipulation of the log entries, we add rules that can only be used by corrupted loggers.

A maintained log provides two properties that can be checked by every participant:

- 1) Append-only: No entries disappear from the log over time.
- 2) SCT Inclusion: Issued SCTs are backed by an entry in the log.

We add the ability to break precisely these two properties in any possible way. Every logger can maintain arbitrarily many logs; each log can be described via the operations addition, and proof construction.

This snapshot is obtained by adding to every operation on the log ( addition, proof construction) a \$log\_id, representing on which log the action is computed. Whenever some party inspects the entries, it will be in the logger's control which log it presents to them. Honest loggers maintain just one log, i.e., there is an injection from \$IdL to \$log\_id.

```

rule Start_Log_malicious:
  [ !Corrupted_Logger($IdL, ~skL, pk(~skL)) ]
  ↳ StartedLogMalicious($IdL, $log_id,
    Time($now)
  ↳ [ ]

```

```

rule remove_from_log_malicious:
  [ KeepLogged(<$IdL, skL, pk(skL)>, $log_id, <<
  ↳ $IdCA, $Id, pkID, 'pre-cert', false(), ser, $i,
  ↳ $j>, sig_preC>)
    /* must have been logged before */,
    !Corrupted_Logger($IdL, skL, pk(skL)) ]
  ↳ [ LoggerComputesRemoveLog($IdL, $log_id, <
  ↳ $IdCA, $Id, pkID, ser>),
    Time($i)
  ↳ [ ]

```

```

rule forge_log_snapshot:
  let
    preC_fields = <$IdCA, $Id, pkID, ser>
    preC_tbs = <$IdCA, $Id, pkID, 'pre-cert',
  ↳ false(), ser, $i, $j>
    preC = <preC_tbs, preC_sig>
  in
  [ In(preC_tbs), In(preC_sig),
    /* Incoming certificate that is added */
    !Corrupted_Logger($IdL, skL, pkL), !CA(
  ↳ $IdCA, pkCA, CA_tbs, CA_sig)
  ↳
  ↳ [ Eq(verify(preC_sig, preC_tbs, pkCA), true())
  ↳ ,

```

```

    LoggerComputesAddLog($IdL, $log_id,
  ↳ preC_fields),
    LoggerComputesAddLogTime($IdL, $log_id,
  ↳ preC_fields, $now),
    Logged($IdL, <preC, <CA_tbs, CA_sig>>), /*
  ↳ Logger stores full chain */
    Time($now)
  ↳
  ↳ [ KeepLogged(<$IdL, skL, pkL>, $log_id, preC)
  ↳ ]

```

Now a log on the trace can be constructed incrementally by adding or removing entries from it.

### 5.3. Visibility Predicates and Maximum Merge Delay.

To formulate that an entry is visible at a given time point in the trace, we introduce visibility predicates that can be reused for this purpose. Intuitively we say that an entry is visible in the considered log with \$log\_id if at the given time point on the trace, the entry has been added (LoggerComputesAddLog) before and not removed after again. Likewise, an entry is not visible if it has never been added or has been added but removed again after. We formalize visibility as predicates based on the #time of inspection.

```

predicate: EntryVisible(Logger, cert, log_id, #
  ↳ time) <=>
  (∃ #t.
    LoggerComputesAddLog(Logger, log_id, cert)
  ↳ @t
    /* there has been an addition to log_id
  ↳ prior to #time */
    ∧ ¬(∃ #t1. LoggerComputesRemoveLog(
  ↳ Logger, log_id, cert)@t1
    ∧ #t < #t1 ∧ #t1 < #time)
    /* after the visible addition, there
  ↳ has been
    no removal of the same entry */
    ∧ #t < #time
  )

```

```

predicate: EntryNotVisible(Logger, cert, log_id, #
  ↳ time) <=>
  ((∃ #t.
    LoggerComputesAddLog(Logger, log_id, cert)
  ↳ @t
    /* there has been an addition to log_id
  ↳ prior to #time */
    ∧ (∃ #t1. LoggerComputesRemoveLog(Logger,
  ↳ log_id, cert)@t1
    /* but removed (and not added again)
  ↳ after */
    ∧ #t < #t1 ∧ #t1 < #time
    ∧ ¬(∃ #t2. LoggerComputesAddLog(
  ↳ Logger, log_id, cert)@t2
    ∧ #t1 < #t2
    ∧ #t2 < #t3)
    )
    ∧ #t < #time
  )
  ∨
  ¬(∃ #t.
    LoggerComputesAddLog(Logger, log_id,
  ↳ cert)@t ∧ #t < #time
    /* or there has never been an addition
  ↳ to the log with log_id */
  )

```

Maximum Merge Delay (MMD). In CT, loggers are allowed to have a time period between SCT issuance and the corresponding entry becoming visible in the log, called maximum-merge delay (MMD). We include this notion in our model by modifying additions to the log and changing the visibility predicates. Additions to the log now become scheduled at the time they are performed. The fact is annotated with the current model time according to the Time  $\hookrightarrow$  action fact. Scheduled additions are pending and, in the sense of MMD, become visible to inspecting parties after the delay has passed.

```

predicate: EntryVisible_MMD(Logger, cert, log_id,
   $\hookrightarrow$  #time, commit_time) <=>
  (  $\exists$  different_time #t.
    LoggerComputesAddLogTime(Logger, log_id,
       $\hookrightarrow$  cert, different_time)@t
    /* there has been an addition to log_id
       $\hookrightarrow$  prior to #time */
     $\wedge \neg(\text{commit\_time} = \text{different\_time})$ 
    /* ...that addition was visible w.r.t. MMD
       $\hookrightarrow$  */
     $\wedge \neg( \exists \#t1. \text{LoggerComputesRemoveLog}(\text{Logger}, \text{log\_id}, \text{cert})@t1$ 
       $\wedge \#t < \#t1 \wedge \#t1 < \#time)$ 
    /* after the visible addition, there has
       $\hookrightarrow$  been no removal of the same
      entry prior to #time */
     $\wedge \#t < \#time$ 
  )

```

We include the time of the proof handout (proof\_time) in the predicate. Additions to the log are visible at proof\_time, if they have been scheduled to a strictly different time (different\_time), which represents a different MMD interval. Removals from the log are assumed to become immediately visible when they occur in the trace.

EntryNotVisible\_MMD is adapted similarly: either the entry has not been visibly added before, or it has been visibly added but removed after and not visibly added after the removal at the time of the inspection.

```

predicate: EntryNotVisible_MMD(Logger, cert,
   $\hookrightarrow$  log_id, #time, commit_time) <=>
  ((  $\exists$  different_time #t.
    LoggerComputesAddLogTime(Logger, log_id,
       $\hookrightarrow$  cert, different_time)@t
    /* there has been an addition to log_id
       $\hookrightarrow$  prior to #time */
     $\wedge \neg(\text{commit\_time} = \text{different\_time})$  /* which
       $\hookrightarrow$  was visible (MMD passed) */
     $\wedge \#t < \#time$ 
     $\wedge ( \exists \#t1. \text{LoggerComputesRemoveLog}(\text{Logger},$ 
       $\hookrightarrow$  log_id, cert)@t1
      /* but removed after */
       $\wedge \#t < \#t1 \wedge \#t1 < \#time$ 
      /* and removal still prior to
         $\hookrightarrow$  considered #time */
       $\wedge \neg( \exists \#t2 \text{ different\_time.}$ 
        LoggerComputesAddLogTime(Logger,
           $\hookrightarrow$  log_id, cert, different_time)@t2
           $\wedge \neg(\text{commit\_time} = \text{different\_time})$ 
           $\wedge \#t1 < \#t2 \wedge \#t2 < \#time$ 
          /* and not visibly added
             $\hookrightarrow$  afterwards again, while still prior to
            the inspection time point */
        )
      )
  )

```

```

)
 $\vee$ 
 $\neg( \exists \text{ different\_time } \#t.
  \text{LoggerComputesAddLogTime}(\text{Logger},
    \hookrightarrow \text{log\_id}, \text{cert}, \text{different\_time})@t
    \wedge \neg(\text{commit\_time} = \text{different\_time})
    /* or there has never been a visible
       $\hookrightarrow$  addition to the log at the time
      of the inspection */
     $\wedge \#t < \#time$ 
  )
)$ 
```

In the following, we will assume these modified visibility predicates and the setting with MMD.

**5.4. Fetching proofs and snapshots.** For both inclusion and append-only proofs, we keep them fully abstract on the trace. This means we do not compute proofs that are then verified when other parties receive them, but use fresh variables as symbolic proofs. In order to be able to make use of proofs in the model, we use different trace properties that capture a snapshot of a log at the time of the proof. We say that the proof succeeds in the model if an actual proof would have worked for the situation of the log at the time of creating the proof. We axiomatically assert this relation with restrictions on the trace.

For some inclusion proof of entry  $x$ , this means that the proof succeeds in the model if, at the time of handing out the proof for some log,  $x$  is visible. Append-only proofs verify that between different snapshots of a log, no entries disappear. In Section E.5 and Section E.6, we present the details of necessary restrictions to model this approach specifically for inclusion and append-only proofs. This approach assumes, as is common for the symbolic model, perfect cryptography of all Merkle tree proofs.

Fetching new snapshots consists of three steps. First, the party that fetches the snapshot starts by creating a ProofRequest fact. Loggers can receive this fact and then hand out the current snapshot for their log, if they are honest, or, in the corruption case, for one of the logs that they maintain, including forged ones.

```

rule ProofFetching:
  [ Fr( $\sim$ session), In($IdFetcher), !Logger($IdL,
     $\hookrightarrow$  pkL) ]
   $\neg$ [
    SomeoneFetchesLogSnapshot($IdFetcher, $IdL
       $\hookrightarrow$  ,  $\sim$ session),
    Time($now)
  ]  $\rightarrow$ 
  [ ProofRequest( $\sim$ session, $IdL, $IdFetcher),
     $\hookrightarrow$  RequestFetched( $\sim$ session)
    /* in a secure channel, network is used for
       $\hookrightarrow$  gossiping */ ]

```

We add a  $\sim$ session value to every proof request in order to assign every start of a proof to the proof itself. In doing so, we are able to formulate time restrictions on a proof run, e.g., by stating that a client starts to run the proof after it has received a corresponding SCT.

In the second step, a logger hands out the snapshot for a \$log\_id of their choice. This is denoted by the

LoggerHandsOutSTHCommit action where  $\$log\_id$  is free. The fact contains the session generated before to attribute the start of the proof fetch to the actual produced proof, involved parties, and a freshly generated  $\sim$  commitment as well as the current time.

```

rule produceSTH_Proof:
  [ !Logger_s($IdL, skL, pkL), Fr( $\sim$ commit) /*
    abstract representation
    of the STH/commitment */ , ProofRequest(session
    , $IdL, $IdFetcher) ]
  -[
    LoggerPresentsSnapshot($IdL, $IdFetcher,
    $log_id, $i,  $\sim$ commit, session),
    /* $log_id of the log the proof is for
    , chosen by the logger */

    Time($i)
  ] $\rightarrow$ 
  [ GivenOutSnapshot($log_entries,  $\sim$ commit,
    session), Out(< $\sim$ commit, session>) ]

```

This rule is reused for both kinds of proofs and presenting all entries of a log, which is relevant for monitors looking for specific certificates. We are allowed to combine this into a single rule since we do not implement proof calculations and thus do not distinguish between these steps. We implicitly assume that loggers comply when presenting a commitment and every underlying entry. This assumption is relaxed when we turn to gossiping, where only the STH commitment is shared (Section 7).

Incoming proof requests and outgoing proofs are not implemented on the network to exclude the Dolev-Yao adversary. Assume the contrary and an adversary that can intercept old proofs and replay them later, while the logger behaves honestly. In these cases, an old snapshot might be used at a later point and disprove inclusion. A natural step would be to blame the logger in this case, as the presented proof is not valid, but the logger was honest the whole time. This is problematic when we aim at achieving accountability.

To highlight this limitation, we explicitly use a secure channel for these steps. Following the CT standard, the communication with a logger is implemented using HTTPS GET and POST requests, excluding such attacks too.

In the third step, the handed-out snapshot is received and checked for the property of interest. We use different rules and restrictions representing every possible outcome of that check. We outline these in the next sections.

**5.5. Inclusion Proof.** We formulate inclusion proofs using the visibility predicates presented in Section E.3. The time-point of the snapshot handout by the log, which is used in the proof, is then applied to the visibility predicate.

```

restriction inclusionProofFails:
  " $\forall$  IdL fields commit session #t5.
  InclusionProofFails(IdL, fields, commit,
  session)#t5
   $\Rightarrow$  ( $\exists$  IdFetcher log_id time #t3.
    LoggerPresentsSnapshot(IdL, IdFetcher,
    log_id, time, commit, session)#t3
    /* Logger handed out a snapshot of its
    log with log_id to the recipient... */
     $\wedge$  #t3 < #t5
    /* ...prior to the check */
     $\wedge$  EntryNotVisible_MMD(IdL, fields, log_id,
    #t3, time))"
  /* ...and the entry is not visible in
  that snapshot */

```

The commitment can be seen as the current STH of the log with  $\$log\_id$ . All underlying  $\$log\_entries$  required for verifying the STH are delivered separately.

Likewise, the second variant uses the `EntryVisible` predicate.

```

restriction inclusionProofWorks:
  "∀ IdL fields commit session #t5.
  InclusionProofSucceeds(IdL, fields, commit,
  ↳ session)#t5
  ⇒ (∃ id log_id time #t2.
    LoggerPresentsSnapshot(IdL, id, log_id,
  ↳ time, commit, session)#t2
    /* Logger handed out a snapshot of its
  ↳ log with log_id to the recipient... */
    ∧ #t2 < #t5
    /* ...prior to the check */
    ∧ EntryVisible_MMD(IdL, fields, log_id, #
  ↳ t2, time))"
  /* and the entry is visible in that
  ↳ snapshot */

```

Depending on the action fact in the trace, we learn whether the proof would have succeeded.

**5.6. Append-Only Proof.** When we talk about the append-only property, we always refer to a pair of snapshots of a log. The proof shows that all entries in the older snapshot are also included in the more recent variant. This proof is not possible in two cases: either entries were deleted between snapshots that were present in at least one existing snapshot a party has, or the log presented a different `$log_id` to the same party, representing a completely different list of entries.

We capture this behavior by introducing a rule that simply collects a snapshot of the log and a second kind of rule that either proves or disproves the property based on an additional fetched snapshot. Similar to the inclusion proof, we use restrictions for that.

```

rule PartyRunsConsistencyCheck:
  [ !Logger($IdL, pkL), In($IdUser),
  ↳ RequestFetched(session),
  ↳ GivenOutSnapshot($log_entries, commit, session
  ↳ ) ]
  ¬[
    ConsistencyCheck($IdUser, $IdL, session,
  ↳ commit),
    Time($i)
  ]⇒
  [ ]

```

We represent the two violation variants as disjunction. The first variant captures the removal from the log using the predicates defined in Section E.3. Sorted by timepoints, the actions represent two proofs that are handed out to the party. For the first proof, an entry is visible, which is not visible anymore in the second proof at `#t6`.

The second variant represents the case where a logger maintains multiple (forged) logs with distinct `$log_id`'s, showing the diverging snapshots to the party.

```

restriction AppendOnlyFails:
  "∀ commit session id IdL #t7.
  AppendOnlyViolation(id, IdL, commit, session)
  ↳ @t7
  /* Append-only violation found by id with
  ↳ affected logger IdL is only
  ↳ possible if */

```

```

  ⇒ (∃ fields time lid1 time2 commit2
  ↳ session2 #t1 #t2 #t3 #t4 #t5 #t6.
    EntryVisible_MMD(IdL, fields, lid1
  ↳ , #t2, time)
    /* there is an entry in the log
  ↳ that is visible at #t2 and MMD
    period time in lid1 */
    ∧ SomeoneFetchesLogSnapshot(id,
  ↳ IdL, session2)#t1
    /* id starts fetching a snapshot
  ↳ of the log maintained by IdL */
    ∧ LoggerPresentsSnapshot(IdL, id,
  ↳ lid1, time, commit2, session2)#t2
    /* The Logger presents the
  ↳ snapshot for lid1 */
    ∧ ConsistencyCheck(id, IdL,
  ↳ session2, commit2)#t3
    /* this snapshot is received and
  ↳ stored by id */
    ∧ LoggerComputesRemoveLog(IdL,
  ↳ lid1, fields)#t4
    /* removal occurred after #t2 */
    ∧ ¬(∃ diff_time #t.
      LoggerComputesAddLogTime(IdL,
  ↳ lid1, fields, diff_time)#t
      ∧ #t4 < #t ∧ #t < #t6 ∧ ¬(
  ↳ time2 = diff_time))
    /* and there is not a visible
  ↳ addition prior to handing out
    the second snapshot, i.e., the
  ↳ cert has actually been removed
    and not added again after */
    ∧ SomeoneFetchesLogSnapshot(id,
  ↳ IdL, session)#t5
    ∧ LoggerPresentsSnapshot(IdL, id,
  ↳ lid1, time2, commit, session)#t6
    /* The logger shows id another
  ↳ snapshot for lid1 */
    ∧ #t1 < #t2 ∧ #t2 < #t3 ∧ #t2 < #
  ↳ t4 ∧ #t4 < #t6 ∧ #t3 < #t5
    ∧ #t5 < #t6 ∧ #t6 < #t7)
    ∨ /* ...or alternatively... */
    (∃ lid1 lid2 time1 time2 commit2
  ↳ session2 #t1 #t2 #t3.
      LoggerPresentsSnapshot(IdL, id,
  ↳ lid1, time1, commit2, session2)#t1
      /* IdL hands out a snapshot of
  ↳ lid1 to id */
      ∧ ConsistencyCheck(id, IdL,
  ↳ session2, commit2)#t2
      ∧ LoggerPresentsSnapshot(IdL, id,
  ↳ lid2, time2, commit, session)#t3
      /* and a second snapshot of lid2
  ↳ which is a different log_id to the
      same recipient (id) */
      ∧ #t1 < #t2 ∧ #t2 < #t3 ∧ #t3 < #
  ↳ t7
      ∧ ¬(lid1 = lid2))"

```

For verifying the append-only property given an up-to-date snapshot of the log, we take all existing snapshots a party has retrieved in the past into account. By this, we assume that a party stores every snapshot it receives at one point and considers them when checking for consistency. Every entry that was visible in any older snapshot must be visible in the newest snapshot too.

```

restriction appendOnlyWorks:
  "(∀ id IdL commit session time1 log_id log_id2
  ↳ fields commit2 session2 time2 #t0 #t1 #t2 #t4 #
  ↳ t5.

```

```

    AppendOnlyVerified(id, IdL, commit,
  ↵ session)@t5
    /* If an append-only proof is verified, we
  ↵ know that ... */
    LoggerComputesAddLog(IdL, log_id, fields)
  ↵ @t0
    ∧ EntryVisible_MMD(IdL, fields, log_id, #
  ↵ t1, time1)
    /* for all entries that have been visible
  ↵ at some point, */
    ∧ LoggerPresentsSnapshot(IdL, id, log_id,
  ↵ time1, commit2, session2)@t1
    ∧ ConsistencyCheck(id, IdL, session2,
  ↵ commit2)@t2
    /* every consistency check processed in
  ↵ the past by that client */
    ∧ LoggerPresentsSnapshot(IdL, id, log_id2,
  ↵ time2, commit, session)@t4
    /* and every snapshot handed out after
  ↵ these */
    ∧ #t0 < #t1 ∧ #t1 < #t2 ∧ #t2 < #t4 ∧ #t4
  ↵ < #t5
    ⇒ EntryVisible_MMD(IdL, fields,
  ↵ log_id2, #t4, time2))
    /* then this entry is also visible in
  ↵ the newest handout */
    ∧ /* and simultaneously... */
    (∀ id IdL commit session #t5.
    AppendOnlyVerified(id, IdL, commit,
  ↵ session)@t5
    /* An append-only proof is only valid if
  ↵ these pre-conditions hold... */)
    ⇒
    (∃ time1 lid fields commit2 session2
  ↵ time2 #t0 #t1 #t2 #t4.
    LoggerComputesAddLog(IdL, lid,
  ↵ fields)@t0
    ∧ EntryVisible_MMD(IdL, fields,
  ↵ lid, #t1, time1)
    ∧ LoggerPresentsSnapshot(IdL, id,
  ↵ lid, time1, commit2, session2)@t1
    ∧ ConsistencyCheck(id, IdL,
  ↵ session2, commit2)@t2
    ∧ LoggerPresentsSnapshot(IdL, id,
  ↵ lid, time2, commit, session)@t4
    ∧ #t0 < #t1 ∧ #t1 < #t2 ∧ #t2 < #
  ↵ t4 ∧ #t4 < #t5))"

```

## 6. Transparency

We show that transparency without further assumptions, i.e., allowing dishonest loggers does not hold. We then consider transparency, assuming honest loggers, and find a counterexample where the MMD is ignored. Assuming honest loggers and inclusion checks that respect the maximum-merge delay then suffice to prove transparency.

We define transparency as visibility of an entry after it has been actively used and first validated by a client. Clients as well as monitors should be able to successfully prove the inclusion after that.

**Lemma** transparency\_doesnt\_hold:

```

  "∀ id IdL IdL1 IdL2 fields commit session #t1
  ↵ #t2 #t3.
    ClientAcceptsCert(IdL1, IdL2, fields)@t1
    ∧ SomeoneFetchesLogSnapshot(id, IdL,
  ↵ session)@t2
    ∧ SomeoneLooksForCert(IdL, fields, commit,
  ↵ session)@t3

```

```

    /* inclusion proof is done by some
  ↵ party (either monitor or client) */
    ∧ ((IdL = IdL1) ∨ (IdL = IdL2))
    ∧ #t1 < #t2 ∧ #t2 < #t3
    ⇒
    (∃ #t. InclusionProofSucceeds(IdL,
  ↵ fields, commit, session)@t)"

```

We are able to find a counterexample for this: the logger is corrupt and hides the entry from its log. We need at least one of the two involved loggers to behave honestly.

**Lemma** transparency\_logger\_client\_without\_mmd:

```

  "∀ id IdL IdL1 IdL2 fields commit session #t1
  ↵ #t2 #t3.
    ClientAcceptsCert(IdL1, IdL2, fields)@t1
    ∧ SomeoneFetchesLogSnapshot(id, IdL,
  ↵ session)@t2
    ∧ SomeoneLooksForCert(IdL, fields, commit,
  ↵ session)@t3
    ∧ ((IdL = IdL1) ∨ (IdL = IdL2))
    ∧ #t1 < #t2 ∧ #t2 < #t3
    ∧ ¬(∃ #t. Corrupted(IdL)@t) /* assume
  ↵ that IdL is honest */
    ⇒
    (∃ #t. InclusionProofSucceeds(IdL,
  ↵ fields, commit, session)@t)"

```

There is still a counterexample for the case of an honest logger: running the proof in the same MMD interval. During this, there is no guarantee of the entry being included. Its absence is not a violation and does not require the corruption of a logger.

Further requiring that the proof is run after the MMD period is sufficient to prove the lemma.

**Lemma** transparency\_logger\_any\_party:

```

  "∀ id IdL IdL1 IdL2 fields commit session #t1
  ↵ #t2 #t3.
    ClientAcceptsCert(IdL1, IdL2, fields)@t1
    ∧ SomeoneFetchesLogSnapshot(id, IdL,
  ↵ session)@t2
    ∧ SomeoneLooksForCert(IdL, fields, commit,
  ↵ session)@t3
    ∧ ((IdL = IdL1) ∨ (IdL = IdL2))
    ∧ #t1 < #t2 ∧ #t2 < #t3
    ∧ ProofRunsAfterMMD(IdL, commit, session)
    ∧ ¬(∃ #t. Corrupted(IdL)@t)
    ⇒
    (∃ #t. InclusionProofSucceeds(IdL,
  ↵ fields, commit, session)@t)"

```

This means that for any certificate used in the CT environment and thus expected to be included in a log (ClientAcceptsCert), visibility is not guaranteed by default. The entry is only guaranteed to be visible if the log is maintained honestly and all considered proofs occur after the MMD.

## 7. Accountable transparency

We have seen that transparency in CT does not hold. We can, however, prove client-side accountability with respect to transparency. For the monitor perspective, we cannot do the same without further assumptions.

For the client-side variant, the client only needs to run an inclusion proof for the certificate it has received. If that

proof is additionally run after the maximum-merge delay, we can use the negative outcome (inclusion violation) as a case test and prove accountability for transparency with that.

```
test LoggerViolatesInclusion_client:
  "∃ IdL1 IdL2 fields commit session log_id
  ↳ time1 id #t0 #t1 #t2.
    ClientAcceptsCert(IdL1, IdL2, fields)@t0
    ∧ ((IdL = IdL1) ∨ (IdL = IdL2))
    ∧ LoggerPresentsSnapshot(IdL, id, log_id,
  ↳ time1, commit, session)@t1
    ∧ ClientFindsInclusionViol(IdL, fields,
  ↳ commit, session)@t2
    ∧ ProofRunsAfterMMD(IdL, commit, session)
    ∧ #t0 < #t1 ∧ #t1 < #t2
  "
```

```
lemma accountable_transparency_CT_clientside:
  LoggerViolatesInclusion_client accounts for
  "∀ IdL1 IdL2 IdL id log_id time1 fields commit
  ↳ session #t0 #t1 #t2.
    ClientAcceptsCert(IdL1, IdL2, fields)@t0
    ∧ ((IdL = IdL1) ∨ (IdL = IdL2))
    ∧ LoggerPresentsSnapshot(IdL, id, log_id,
  ↳ time1, commit, session)@t1
    /* Logger handed out a snapshot of its
  ↳ log with log_id to the recipient... */
    ∧ ClientChecksInclusion(IdL, fields,
  ↳ commit)@t2
    /* client-side inclusion check of
  ↳ fields in the given snapshot */
    ∧ ProofRunsAfterMMD(IdL, commit, session)
    ∧ #t0 < #t1 ∧ #t1 < #t2
    ⇒
    EntryVisible_MMD(IdL, fields, log_id,
  ↳ #t1, time1)
  "
```

For the monitor, essentially the same lemma but with the inclusion proof requested by the monitor (instead of the client) verifies. In this scenario, however, the monitor is assumed to check inclusion after the client accepted the certificate. This requires communication between both parties, which generally does not apply to CT.

Running the proof after a client accepted the certificate with embedded SCTs received from loggers IdL1 and IdL2 encodes that the proof is run after the considered logger (IdL1 or IdL2) made the promise to include the certificate. By receiving SCTs along with certificates, a client learns these promises from loggers and then can check using inclusion proofs whether this promise is fulfilled. If not, the logger can be blamed for this. A client learns this for every certificate it encounters, but a monitor does not. Monitors in CT do not receive the certificate with embedded SCTs prior to fetching a log. They are missing knowledge of when an inclusion proof for a specific entry must succeed. A realistic perspective for the monitor excludes that the proof is run after a client accepted the certificate to capture the knowledge of the monitor more precisely.

If we exclude this, the lemma does not hold anymore, and uniqueness fails. In the counterexample, the case test leads to blaming a logger that is not corrupted and behaves honestly. Transparency does not hold due to equivocation, as shown earlier. Additionally, the monitor is unaware of any

violation by the loggers and cannot assign blame without further knowledge, for example, by communicating with a client.

## 8. Accountable authenticity

**8.1. Monitor as external validator.** In a first step, we move the external check to the monitor. We add two rules that each receive a certificate to check using the TrackSubject fact, which contains the ground truth. Depending on whether the incoming certificate contradicts the known key and identity, the monitor adds a CAFakedCert action fact to the trace.

```
rule monitor_externalCheck_Fake:
  let
    fields = <$IdCA, $Id, pkID, ser>
    tbs = <$IdCA, $Id, pkID, 'PubKey', false()
  ↳ , ser, $i, $j>
  in
    [ !TrackSubject($IdM, $Id, pubK), !CA($IdCA,
  ↳ pkCA, CA_tbs, CA_sig),
    In(<$IdCA, $Id, pkID, 'PubKey', false(), ser
  ↳ , $i, $j, sct1, sct2>),
    In(sig) ]
    ¬[ Eq(verify(sig, tbs, pkCA), true()), /*
  ↳ signed by $IdCA */
      _restrict( ¬(pkID = pubK)),
      /* received cert contradicts the known
  ↳ truth to the monitor */
      CAFakesCert($IdCA, fields), /* CA blamed
  ↳ */
      MonExtFakeCheck(fields),
      Time($now)
    ]
  ↳
  [ ]
```

```
rule monitor_externalCheck_Verified:
  let
    fields = <$IdCA, $Id, pkID, serial>
    tbs = <$IdCA, $Id, pkID, 'PubKey', false()
  ↳ , ser, $i, $j>
  in
    [ !TrackSubject($IdM, $Id, pubK), !CA($IdCA,
  ↳ pkCA, CA_tbs, CA_sig),
    !HonestlyIssued(
      <<$IdCA, $Id, pkID, 'pre-cert', false
  ↳ (), serial, $i, $j>, pre_sig>),
    In(<$IdCA, $Id, pkID, 'PubKey', false(),
  ↳ ser, $i, $j, sct1, sct2>),
    In(sig)
  ]
    ¬[ MonExtFakeCheck(fields),
      Eq(verify(sig, tbs, pkCA), true()), /*
  ↳ signed by $IdCA */
      Eq(pkID, pubK), /* public key is known to
  ↳ the monitor */
      Time($now)
    ]
  ↳
  [ ]
```

With CAFakesCert used as the test, we can prove accountability for authenticity.

```
test dishonest_CA:
  "∃ fields idC #i #j.
    CAFakesCert(IdCA, fields)@i
```

```

    /* External check found that the
    ↪ certificate with fields is fake */
    ∧ ClientAcceptsCert_Fields(idC, fields)@j
    "

lemma acc_faked_cert_CA:
  dishonest_CA accounts for
  "∀ idC fields #t1 #t2.
    ClientAcceptsCert_Fields(idC, fields)@t1
    /* If a client receives a certificate
    ↪ with the claim in fields... */
    ∧ MonExtFakeCheck(fields)@t2
    /* ... and there has been an external
    ↪ check done by the monitor */
    ⇒ (∃ #t0.
      ServersRealCert(fields)@t0
      /* ...then this claim is true */
      ∧ #t0 < #t1)"

```

This allows us to reach the same result as with the external check, but we again rely on unrealistic assumptions. We assume trust in the monitor, that it knows the ground truth, and leave out how the certificate to check reaches the monitor.

In the next step, we use the log entries as the source for the certificate that is being checked.

**8.2. Checking certificates in the log.** We use inclusion proofs run by the monitor to formally state whether a monitor looked for a specific entry. If an entry is visible, the monitor learns about the entry and then runs a similar check as before by comparing the statement (identity-key-pair) with the known ground truth to the monitor. This is formalized in the next rule.

```

rule monitorFindsRogueCertificate:
  let
    fields = <$iss, su, ke, ser>
  in
    [
      !LoggerDict($IdM, $IdL, logcert), !
      ↪ TrackSubject($IdM, su, pubK),
      !St_Monitor($IdM), !Logger($IdL, pkL), In(
      ↪ fields),
      GivenOutSnapshot($log_entries, commit,
      ↪ session)
    ]
    [
      _restrict(-(pubK = ke)),
      /* found cert actually has a different
      ↪ statement than the original one
      (different identity) */
      InclusionProofSucceeds($IdL, fields,
      ↪ commit, session),
      MonitorTryAuthCheckLog($IdL, <su, ke>),
      Monitor_CAFakesCertInLogEntry($iss, $IdL,
      ↪ <su, ke>),
      Time($now)
    ]
    ]

```

The statement in the accountability lemma slightly changes now. We state that the test provides us with accountability under the assumption that a monitor finds a forged certificate in a log (Monitor\_CAFakesCertInLogEntry) where the log is one of the two that issued an SCT for it (ClientAcceptsCert).

```

test ca_rogue_replaced_with_monitor_fetch:
  "∃ fields IdL1 IdL2 IdL #i #j.
    Monitor_CAFakesCertInLogEntry(IdCA, IdL,
    ↪ fields)@i
    /* Monitor found that the cert with
    ↪ fields has been faked by IdCA */
    ∧ ((IdL = IdL1) ∨ (IdL = IdL2))
    ∧ ClientAcceptsCert_Key(IdL1, IdL2, fields
    ↪ )@j
    "

lemma acc_CT_monitor_fetches_log:
  ca_rogue_replaced_with_monitor_fetch accounts
  ↪ for
  "∀ IdL IdL1 IdL2 fields #i #j.
    MonitorTryAuthCheckLog(IdL, fields)@i
    /* A monitor checks authenticity of a
    ↪ cert with fields */
    ∧ ((IdL = IdL1) ∨ (IdL = IdL2))
    ∧ ClientAcceptsCert_Key(IdL1, IdL2, fields
    ↪ )@j
    ∧ #j < #i
    ⇒ (∃ #k. ServersRealKey(fields)@k)"

```

We obtain a negative result for this lemma: the counterexample shows a trace where the log is corrupted and hides the entry that the monitor is looking for. Our test does not match in that case, while authenticity is still violated. This result is no surprise, given that transparency does not hold in this model, and the same attack against transparency also violates this approach. If transparency holds, we are able to prove accountable authenticity with the next variant.

```

test ca_rogue_replaced_with_honest_monitor_fetch:
  "∃ fields stmt IdM IdL1 IdL2 IdL commit
  ↪ session #t0 #t1 #t2.
    ClientAcceptsCert_Key(IdL1, IdL2, stmt)@t0
    ∧ ClientAcceptsCert(IdL1, IdL2, fields)@t0
    ∧ SomeoneFetchesLogSnapshot(IdM, IdL,
    ↪ session)@t1
    /* Monitor fetches from the log after a
    ↪ client accepted
    the certificate */
    ∧ MonitorChecksInclusion(IdM, IdL, fields,
    ↪ commit, session)@t2
    /* Monitor runs an inclusion proof for the
    ↪ same certificate */
    ∧ Monitor_CAFakesCertInLogEntry(IdCA, IdL,
    ↪ stmt)@t2
    /* And finds that the contained key
    ↪ contradicts its knowledge
    -> CA is blamed */
    ∧ ProofRunsAfterMMD(IdL, commit, session)
    ∧ ((IdL = IdL1) ∨ (IdL = IdL2))
    ∧ #t0 < #t1
    ∧ ¬(∃ #t. Corrupted(IdL)@t)"
    /* considered logger is honest */

```

/\* Accountable authenticity under honest logger (
 ↪ transparency) holds \*/

```

lemma acc_CT_monitor_fetches_honest_log:
  ca_rogue_replaced_with_honest_monitor_fetch
  ↪ accounts for
  "∀ IdM IdL IdL1 IdL2 stmt fields commit
  ↪ session #t0 #t1 #t2.
    ClientAcceptsCert_Key(IdL1, IdL2, stmt)@t0
    ∧ ClientAcceptsCert(IdL1, IdL2, fields)@t0
    ∧ SomeoneFetchesLogSnapshot(IdM, IdL,
    ↪ session)@t1
    /* Monitor fetches from the log after
    ↪ a client accepted

```

```

    the certificate */
    ∧ MonitorChecksInclusion(IdM, IdL, fields,
    ↳ commit, session)@t2
    /* Monitor runs an inclusion proof for
    ↳ the same certificate */
    ∧ MonitorTryAuthCheckLog(IdL, stmt)@t2
    /* fetched certificate, if found, is
    ↳ checked for authenticity */
    ∧ ProofRunsAfterMMD(IdL, commit, session)
    ∧ ((IdL = IdL1) ∨ (IdL = IdL2))
    ∧ #t0 < #t1
    ∧ ¬(∃ #t. Corrupted(IdL)@t)
    /* considered logger is honest */
    ⇒ (∃ #k. ServersRealKey(stmt)@k)"

```

In the next chapter, we consider the real-world-inspired approach known as SCT auditing. With SCT auditing, we obtain stronger results; in particular, transparency becomes provable. With transparency at hand, we can also show accountability, and monitors are able to additionally blame loggers if they hide an entry in modified case tests.

## 9. Modelling SCT Auditing (Details)

In this chapter, we will introduce this addition to our model and reformulate properties to apply them to SCT auditing.

```

rule monitorAudit_no_viol:
  let
    audit_fields = <$IdCA, $Id, pkID, ser>
    pub_tbs = <$IdCA, $Id, pkID, 'PubKey',
    ↳ false(), ser, $i, $j, sct1, sct2>
    sct_tbs = <$IdL, 'SCT', $sct, <$IdCA, $Id,
    ↳ pkID, false(), ser, $i, $j>>
    sct_to_audit = <$IdL, 'SCT', $sct, sct_sig
    ↳ >
    in
    [ !LoggerDict($IdM, $IdL, logcert), !
    ↳ St_Monitor($IdM), !Logger($IdL, pkL),
    ↳ !TrackSubject($IdM, $Id, pubK), !CA($IdCA,
    ↳ pkCA, CA_tbs, CA_sig),
    ↳ In(sct_to_audit), In(pub_tbs), In(pub_sig)
    ]
    [ Eq(verify(pub_sig, pub_tbs, pkCA), true()),
    ↳ Eq(verify(sct_sig, sct_tbs, pkL), true()),
    ↳ _restrict((sct_to_audit = sct1) ∨ (
    ↳ sct_to_audit = sct2)),
    ↳ _restrict(pubK = pkID),
    ↳ /* does not contradict known public key,
    ↳ no further action */

    ServerLearns(audit_fields),
    Audited_SCT($IdM, audit_fields),
    Audited_SCT_Logger($IdM, $IdL,
    ↳ audit_fields),
    ↳ Audit_CertAuthCheck(<pub_tbs, pub_sig>,
    ↳ sct_to_audit, <$Id, pkID>),

    Time($now)
  ]

```

```

rule monitorAudit_CA_Viol:
  let
    audit_fields = <$IdCA, $Id, pkID, ser>

```

```

    pub_tbs = <$IdCA, $Id, pkID, 'PubKey',
    ↳ false(), ser, $i, $j, sct1, sct2>
    sct_tbs = <$IdL, 'SCT', $sct, <$IdCA, $Id,
    ↳ pkID, false(), ser, $i, $j>>
    pubkey_cert = <pub_tbs, pub_sig>
    sct_to_audit = <$IdL, 'SCT', $sct, sct_sig
    ↳ >
    in
    [ !LoggerDict($IdM, $IdL, logcert), !
    ↳ St_Monitor($IdM), !Logger($IdL, pkL),
    ↳ !CA($IdCA, pkCA, CA_tbs, CA_sig), !
    ↳ TrackSubject($IdM, $Id, pubK),
    ↳ In(sct_to_audit), In(pub_tbs), In(pub_sig) ]
    [ Eq(verify(pub_sig, pub_tbs, pkCA), true()),
    ↳ Eq(verify(sct_sig, sct_tbs, pkL), true()),
    ↳ _restrict((sct_to_audit = sct1) ∨ (
    ↳ sct_to_audit = sct2)),
    ↳ _restrict(¬(pubK = pkID)), /* received
    ↳ cert contradicts the known truth,
    ↳ monitor will take action */
    ↳ ServerLearns(audit_fields),

    Audited_SCT($IdM, audit_fields),
    Audited_SCT_Logger($IdM, $IdL,
    ↳ audit_fields),
    ↳ Audit_CertAuthCheck(pubkey_cert,
    ↳ sct_to_audit, <$Id, pkID>),
    ↳ Audit_CAFakesCert($IdCA, pubkey_cert,
    ↳ sct_to_audit, <$Id, pkID>),
    ↳ /* CA blamed for violation */

    Time($now)
  ]

```

We additionally add a sleep variant, where the monitor finds a rogue certificate but does not react to it, e.g., does not inform the subject and does not blame the responsible CA. This is considered malicious monitor behavior.

## 10. Transparency

In the previous chapter, we showed that transparency to the monitor does not hold without assuming that the MMD has passed and that the considered log is maintained honestly. Assuming that the log is honest is strong and just delegates the trust from the CA in the PKI to the logger in the CT system.

Starting in this section, we relax our transparency notion in order to apply it to SCT auditing. So far, we have stated transparency in terms of the log. When a party fetches the log after an SCT for that log has been issued, transparency holds if the entry is then visible to that client. To include auditing, we instead state that transparency holds if the monitor just learns of an entry by either finding it in a log or receiving it over the network via auditing or gossiping.

Beyond transparency, acquiring knowledge through auditing can be used by monitors to learn what to expect from a logger when they fetch another view and run inclusion proofs. This allows monitors to then hold loggers accountable for misbehavior as we show in ?? and section 6.

Transparency assuming that the certificate is being audited is a straightforward property, since the monitor does

not need the (possibly corrupted) log anymore to learn new entries. The lemma below summarizes that property and verifies.

```
lemma transparency_SCT_Auditing:
  "∀ idM idL stmt #t0.
    AuditedSCT(idM, idL, stmt)@t0
    ⇒
    (∃ #t. MonitorLearns(stmt)@t)
  "
```

We need to assume that the monitor inspects the log and certificates it receives honestly, i.e., that the monitor does not sleep.

## 11. Modelling SCT Gossiping (Details)

In this chapter, we will introduce the additions to our model and reformulate properties to apply them to SCT gossiping.

## 12. Transparency

We first consider a variant where a client accepted the certificate with `fields`, retrieved a commit, and then gossiped that commit to the monitor. We want to prove that, in such a case, the monitor will be able to see the entry with the gossiped commit.

```
lemma no_transparency_gossip_without_client_proof:
  "∀ l1 l2 l id log_id time IdM fields
    ↳ gos_commit gos_session #t0 #t1 #t2.
      ClientAcceptsCert(l1, l2, fields)@t0
      ∧ ((l = l1) ∨ (l = l2))
      ∧ LoggerPresentsSnapshot(l, id, log_id,
    ↳ time, gos_commit, gos_session)@t1
      /* snapshot is after the SCT has been
    ↳ used in a validation
      -> should be visible */
      ∧ #t0 < #t1
      ∧ Gossiped(IdM, l, fields, <gos_commit,
    ↳ gos_session>)@t2
      /* snapshot is shared and monitor
    ↳ looks specifically for fields */
      ⇒
      (∃ #t.
        GosInclusionProofWorks(IdM, l, fields,
    ↳ gos_commit, gos_session)@t)
      /* then this entry should be
    ↳ visible to the monitor */
  "
```

This lemma does not hold. If a logger is malicious and hides an entry in the gossiped snapshot, the monitor will not be able to see it.

To mitigate this, we add an inclusion proof performed by the client. Now, we get the guarantee that the certificate is visible in the underlying STH.

```
lemma no_transparency_gossip_with_client_proof:
  "∀ l1 l2 l IdM fields gos_commit gos_session #
    ↳ t0 #t1 #t2.
      ClientAcceptsCert(l1, l2, fields)@t0
      ∧ ClientSideInclusionProved(l, fields, <
    ↳ gos_commit, gos_session>)@t1
```

```
/* client verified inclusion of fields
↳ in gos_commit */
  ∧ ((l = l1) ∨ (l = l2))
  ∧ Gossiped(IdM, l, fields, <gos_commit,
    ↳ gos_session>)@t2
  /* gos_commit is shared with monitor IdM
↳ which looks
  for fields in the log */
  ⇒
  (∃ #t.
    GosInclusionProofWorks(IdM, l, fields,
    ↳ gos_commit, gos_session)@t)
  /* then the monitor with
↳ gos_commit should see fields too */
  "
```

As discussed earlier, our model allows loggers to hide entries even if an entry is contained in the gossiped STH, at the cost of being exposed for an append-only violation.

The counterexample shows the partition attack again. While the client verified the inclusion, the inclusion proof ran by the monitor, with the gossiped snapshot at hand, fails. The monitor can use the gossiped snapshot to check append-only between the gossiped and fetched snapshot. We will see next, that this can be used to achieve accountable transparency.

## 13. Accountable transparency

To show accountable transparency, we start a successful inclusion proof run by a client. The underlying STH that has been used in the inclusion proof is gossiped to a monitor, and we expect that the same proof works for the monitor side too, i.e., the monitor can see the entry. The append-only check implicitly assumes that the gossiped commit happened prior to the second commit, which we make explicit in this lemma.

```
test LoggerViolatesInclusion_monitor_gossip:
  "∃ IdM logid fields time2 gos_commit
    ↳ gos_session commit session #t1 #t2 #t3 #t4 #t5.
      ClientSideInclusionProved(IdL, fields, <
    ↳ gos_commit, gos_session>)@t1
      /* client verified that fields is visible in
    ↳ gos_commit */
      ↳ SomeoneFetchesLogSnapshot(IdM, IdL, session)
    ↳ @t2
      /* monitor starts own inclusion proof */
      ∧ LoggerPresentsSnapshot(IdL, IdM, logid,
    ↳ time2, commit, session)@t3
      /* receives commit from logger */
      ∧ #t1 < #t2
      ∧ Gos_CannotFindCert(IdM, fields, commit,
    ↳ session, gos_commit, gos_session)@t4
      /* but inclusion in this commit is not
    ↳ falsified */
      ∧ Gos_AppendOnlyViol(IdM, IdL, commit, session
    ↳ , gos_commit, gos_session)@t5"
      /* the monitor finds that append-only between
    ↳ gos_commit and commit is
      violated and blames the logger */
```

This test is based on the findings in Section L, where we found an append-only violation, which we will use with a free logger variable. We add the same conditions

to formulate the assumed relationship between the gossiped and fetched commit as before, but state that the append-only proof and inclusion proof fail.

```
Lemma accountable_transparency_gossiped_monitor:
  LoggerViolatesInclusion_monitor_gossip
  ⌊ accounts for
    "∀ IdM IdL log_id time2 fields commit session
    ⌊ gos_commit gos_session #t1
      #t2 #t3 #t4 #t5.
      ClientSideInclusionProved(IdL, fields, <
    ⌊ gos_commit, gos_session>@t1
      /* client proofs inclusion of fields in
    ⌊ gos_commit */
      ⌊ SomeoneFetchesLogSnapshot(IdM, IdL, session)
    ⌊ @t2
      ⌊ LoggerPresentsSnapshot(IdL, IdM, log_id,
    ⌊ time2, commit, session)@t3
      ⌊ #t1 < #t2
      ⌊ Gos_MonInclCheck(IdM, fields, gos_commit,
    ⌊ gos_session, commit, session)@t4
      /* monitor looks for fields with
    ⌊ gos_commit with new proof commit */
      ⌊ Gos_AppendOnlyCheck(IdM, IdL, commit,
    ⌊ session, gos_commit, gos_session)@t5
      /* append only between gos_commit and
    ⌊ commit is checked */
      ⇒
      InclusionProofSucceeds(IdL, fields, commit
    ⌊ , session)@t4
      /* then the inclusion proof of fields
    ⌊ in commit should also hold */
      "
```

With this case test, accountable transparency with gossiping is verified.

## 14. Accountability with naive certificate revocation

In Section 2 we mentioned that the steps after a subject learns of a rogue certificate are not defined in CT. Usual approaches include blaming the CA and taking steps to get the certificate revoked. In this section we add an idealized certificate revocation mechanism that is used as soon as an authenticity check identifies a rogue certificate. In this section we assume that honest monitors revoke a rogue certificate immediately when found and that clients only accept certificates if they have not been revoked.

**14.1. Simplified revocation model.** We implement a simplified revocation using a restriction. Now clients only accept (validate) a certificate if it has not been revoked already.

```
restriction onlyValidateNonRevoked:
  "∀ idL1 idL2 rootCA CA_fields idCA stmt #i.
  ⌊ ClientAcceptsChain(idL1, idL2, rootCA,
  ⌊ CA_fields, idCA, stmt)@i
  ⇒ not (∃ #j. Revoked(stmt)@j ∧ #j < #i)"
```

The Revoked action fact is added to every authenticity check, including the checks performed in auditing. Violating authenticity now requires two corruptions: a malicious CA, as before, and a monitor that ignores a rogue certificate and thus does not revoke it. The revocation model is ideal in the sense that it becomes immediately visible to clients, and monitors are able to issue the revocation; usually the responsible CA does this on request.

**14.2. Security properties.** We again focus on accountable authenticity. With revocation, however, a monitor can prevent violations of authenticity by scanning for suspicious certificates before a client accepts them. To violate authenticity now, a CA must collude either with the logger or, alternatively, with the monitor.

We show accountability under three different variants: (1) perfect transparency, where no logger is corrupted, (2) transparency for the considered logger holds, and (3) no restriction on logger misbehavior.

To model that a monitor inspects a given log prior to a timestamp point on the trace yet after an SCT exists, we introduce a liveness predicate.

```
predicate: MonitorLiveness(stmt, #validation_time)
  ⌊ <=>
    (∃ IdL IdM time commit session
    ⌊ log_id1 log_id2 #j #k #l.
      LoggerComputesAddLog(IdL,
    ⌊ log_id1, stmt)@j
      /* SCT exists already */
      ⌊ LoggerPresentsSnapshot(IdL,
    ⌊ IdM, log_id2, time, commit, session)@k
      /* Monitor receives a
    ⌊ snapshot from the log after it has been added.
    ⌊ Here the logger may be dishonest and is able to
    ⌊ use a different snapshot */
      ⌊ not (∃ #t0. Time(time)@t0 ∧
    ⌊ #t0 < #k)
      /* only considered after
    ⌊ MMD (predicate is forbidden here) */
      ⌊ MonitorInspectsLog_noTrust(
    ⌊ IdL, stmt, commit, session)@l
      ⌊ #j < #k ∧ #k < #l
      ⌊ #l < #validation_time
    )
```

To achieve perfect transparency, we exclude any logger corruption. If a monitor is live before a client accepts the certificate, but after the certificate already exists, authenticity holds, unless that monitor sleeps and ignores rogue certificates or the logger did not add the entry yet. Our case test captures this misbehavior using the MonitorSleeps action fact where the identities of the CA and monitor are both free.

```
test monitor_sleeps_intm_CA:
  "∃ stmt idL1 idL2 rootCA CA_fields #t #t1 #t2
  ⌊ .
    MonitorSleeps(idM, stmt)@t
    ⌊ ClientAcceptsChain(idL1, idL2, rootCA,
  ⌊ CA_fields, idCA, stmt)@t1
    ⌊ Ext_LegitimateCA(rootCA, CA_fields,
  ⌊ idCA)@t2
    ⌊ not (∃ idL #t. CompromiseLogger(idL)@t)
    /* assumed perfect transparency */
    ⌊ MonitorLiveness(stmt, #t1)
    "
```

Throughout, we consider a second test that is nearly identical, but blames the root CA if the intermediate CA is found illegitimate.

```
Lemma A_Revoke_1:
  monitor_sleeps_intm_CA, monitor_sleeps_root_CA
  ⌊ accounts for
  "∀ idL1 idL2 rootCA CA_fields idCA stmt #t1 #
  ⌊ t2.
```

```

    MonitorLiveness(stmt, #t1)
    ∧ not (∃ idL #t. CompromiseLogger(idL)@t)
    /* assumed perfect transparency */
    ∧ ClientAcceptsChain(idL1, idL2, rootCA,
    ↪ CA_fields, idCA, stmt)@t1
    ∧ LegitimateCheck(rootCA, CA_fields)@t2

    ⇒
    (∃ #t0. GroundTruth(idCA, stmt)
    ↪ @t0 ∧ #t0 < #t1)
    "

```

We require both to be corrupted in a violation, as a monitor corruption is not enough to break authenticity, but in this model also does not prevent it. Not allowing any logger corruption is quite strong, and we can weaken this requirement by only assuming that the considered log, which is used by the monitor to explore rogue certificates, is honest.

```

test monitor_sleeps_cond_intm_CA:
    "∃ idL1 idL2 rootCA CA_fields stmt #t #t1 #t2
    ↪ .
        MonitorSleeps(idM, stmt)@t
        ∧ MonitorLivenessWithHonestLogger(idL1,
    ↪ idL2, stmt, #t1)
        ∧ ClientAcceptsChain(idL1, idL2, rootCA,
    ↪ CA_fields, idCA, stmt)@t1
        ∧ Ext_LegitimateCA(rootCA, CA_fields,
    ↪ idCA)@t2
    "

```

```

lemma A_Revoke_2:
    monitor_sleeps_cond_intm_CA,
    monitor_sleeps_cond_root_CA accounts for
    "∀ idL1 idL2 rootCA CA_fields idCA stmt #t1 #
    ↪ t2.
        MonitorLivenessWithHonestLogger(idL1, idL2
    ↪ , stmt, #t1)
        ∧ ClientAcceptsChain(idL1, idL2, rootCA,
    ↪ CA_fields, idCA, stmt)@t1
        ∧ LegitimateCheck(rootCA, CA_fields)@t2
        ⇒
        (∃ #t0. GroundTruth(idCA, stmt)
    ↪ @t0 ∧ #t0 < #t1)
    "

```

We can prove both variants. Further relaxing this assumption by not restricting any corruption reveals an attack.

```

test monitor_sleeps_unconditional_intm_CA:
    "∃ idL1 idL2 rootCA CA_fields stmt #t #t1 #t2
    ↪ .
        MonitorSleeps(idM, stmt)@t
        ∧ ClientAcceptsChain(idL1, idL2, rootCA,
    ↪ CA_fields, idCA, stmt)@t1
        ∧ Ext_LegitimateCA(rootCA, CA_fields,
    ↪ idCA)@t2
        ∧ MonitorLiveness(stmt, #t1)
    "

```

```

lemma A_Revoke_3:
    monitor_sleeps_unconditional_intm_CA,
    monitor_sleeps_unconditional_root_CA accounts for
    ↪ for
    "∀ idL1 idL2 rootCA CA_fields idCA stmt #t1 #
    ↪ t2.
        MonitorLiveness(stmt, #t1)
        ∧ ClientAcceptsChain(idL1, idL2, rootCA,
    ↪ CA_fields, idCA, stmt)@t1

```

```

    ∧ LegitimateCheck(rootCA, CA_fields)@t2
    ⇒
    (∃ idCA #t0. GroundTruth(idCA,
    ↪ stmt)@t0 ∧ #t0 < #t1)
    "

```

The considered log fetched by the monitor now is manipulated. The attack shows a trace where the certificate has been hidden by the logger, by crafting a proof for a log that does not contain said certificate. The monitor does not see the rogue certificate and thus does not revoke it. This violates authenticity, although the case test does not apply, and verifiability is violated. We have already seen that auditing can be used to improve transparency by not relying solely on the logs anymore while also allowing to obtain accountable authenticity (Section J). Next we will consider the revocation model also for auditing.

```

test monitor_sleeps_unconditional_audit_intm_CA:
    "∃ idL1 idL2 rootCA CA_fields stmt #t1 #t2 #
    ↪ t3.
        MonitorAuditSleeps(idCA, idM, stmt)@t1
        ∧ ClientAcceptsChain(idL1, idL2, rootCA,
    ↪ CA_fields, idCA, stmt)@t2
        ∧ #t1 < #t2
        ∧ Ext_LegitimateCA(rootCA, CA_fields,
    ↪ idCA)@t3
    "

```

```

lemma A_RevokeAudit_1:
    monitor_sleeps_unconditional_audit_intm_CA,
    ↪ monitor_sleeps_unconditional_audit_root_CA
    ↪ accounts for
    "∀ idM idL1 idL2 idL rootCA CA_fields idCA
    ↪ stmt #t1 #t2 #t3.
        AuditedSCT(idM, idL, stmt)@t1
        ∧ #t1 < #t2
        ∧ ClientAcceptsChain(idL1, idL2, rootCA,
    ↪ CA_fields, idCA, stmt)@t2
        ∧ LegitimateCheck(rootCA, CA_fields)@t3
        ⇒
        (∃ #t0. GroundTruth(idCA, stmt)
    ↪ @t0 ∧ #t0 < #t1)
    "

```

In this variant we assume that every certificate is shared with the monitor via auditing prior to accepting it. Again, there is one violation possible if the CA and monitor collude.

Mounting the same approach to gossiping fails here. As in CT, if both, the CA and logger collude the monitor, even if honest, is not able to learn of the rogue certificate, nor initiate a revocation in time. Without further assumptions, we cannot achieve accountable authenticity in this variant, which is in line with the previous findings in Section 7.

Discussion. The tests used in the different properties are, in this variant, not practical in real applications. While we are able to hold monitors accountable here, the tests rely on the action fact that is part of the malicious behavior—it is not clear which distinct party is able to do that in a real-world setting. We explore a possible solution in ?? while presenting limitations and challenges there. While we are over simplifying revocation here, the model reveals new attack patterns that only work when multiple parties (CA and monitor) collude and deviate from the protocol.