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, loggers, monitor but also the end user's client—it is very hard to provide a precise account of how CT eliminates trust assumptions in exchange for complex infrastructure. Analyses both in Dolev-Yao paradigm and the computational paradigm 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 analyzing proposed extensions for SCT Auditing and Gossiping. We show that plain CT relies on an honest log, but provides accountability under this assumption. Furthermore, 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 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 aim to identify parties violating the security property. This mechanism is part of the protocol although often underspecified. It is implemented through the parties in the protocol, 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 able to describe how to keep logs or monitors accountable. Our results are the first to consider corruption scenarios involving all three roles: CA, log and monitor. Compared to 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 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.

Besides 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.

Organisation. In **??**, we start with necessary background, including the CT standard and its proposed extensions. 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 recapitulates our findings and presents details on the proofs for every considered

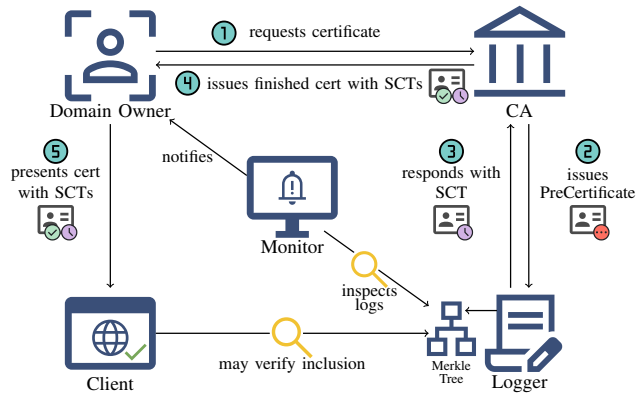

Figure 1. Different roles and usual information flow in the CT protocol with pre-certificates.

property and model. Section 9 summarizes other works that consider CT and accountability.

2. Certificate Transparency

To ensure the authenticity and integrity of the TLS handshake used in HTTPS, hosts must present a certificate that maps their identity (which includes the domain that the user typed in the URL bar) to their public key. These certificates come from third parties, known as Certificate Authorities (CAs). Clients trust the CAs that are (transitively) trusted by a root CA. Hundreds of root CAs come with the browser, each equally trusted and thus a single point of failure. Root CAs can—and have in the past [1], [2]—issued rogue certificates.

By virtue of being signed, rogue certificates identify their issuer, hence Certificate Transparency (CT, [3], [4]) makes them public so browser vendors can remove the responsible CAs from 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 [5] and contain the certificate's issuer, the object's (e.g., website's) identity and associated public key, a validity period and they have a unique serial number. X.509 allows extensions fields; CT uses those to attach information about where a certificate was logged.

Loggers. Loggers maintain a publicly accessible list of certificates issued by CAs. A valid entry to the log must consist of the target certificate and the complete chain of certificates required to validate it. Anyone can submit an entry; the logger must accept it 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 'pre-certificate' extension that ensures clients will never accept it. Loggers accept pre-certificates and thus issue an SCT. The SCT is computed over the pre-certificate without the pre-certificate extension, while the later log entry contains the full 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 [6], [7].

Signed Certificate Timestamps. Signed Certificate Timestamps (SCTs) contain the identity of the logger that issued them, a version field and the time 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, the *maximum-merge delay* (MMD). It is one of the configuration parameters of a log and 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 the CT specification [3] for details.

Important for CT is that Merkle Trees provide a function AddLog to add an entry to a log and that a tree can be signed by 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 a snapshot of the log. 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 certificate by maintaining two views, e.g., Merkle tree T_1 contains a rogue certificate and can produce an inclusion proof for this certificate, while

T_2 is T_1 minus that 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 [8], [9].

Monitors. Owners of certificates (called *subjects*) can inspect logs via monitors. They can use tools like cert spotter [15] to monitor for their own certificates, or appoint third parties to do it for them [16]. 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 which keys may appear for it needs to be communicated out-of-band. If it finds a rogue certificate, this indicates potential misbehavior of a CA. The monitor reports to the subject, which can then take further action to get the certificate revoked; although the CT standard does not specify how this is done. 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 [17]. Symantec's CA was subsequently forced to issue only certificates that confirm to CT. Due to later compliance failures, it is now entirely distrusted by Chrome, and thus most Internet users [1].

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 [3], [4], [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 regard to authenticity, this has not security implications; our models for CT 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 they require. 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 they accept.

We compare these design decisions with the guarantees that we can prove and the assumptions these proofs require Section 8.

3. Background

3.1. Tamarin

Tamarin performs automated analysis in the Dolev-Yao model [19], wherein messages are described as abstract terms. Terms are composed from uninterpreted function symbols that represent cryptographic primitives and are applied to *variables* representing yet unknown values, as well as *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 behavior 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 $\mathsf{h}/1$ to indicate that h takes 1 parameter. Symmetric encryption and decryption are typically written as the function symbols sign, verify, true and the equation $\mathsf{verify}(\mathsf{sign}(m, \mathsf{sk}), m, \mathsf{pk}(\mathsf{sk})) = \mathsf{true}$, 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 describe the dynamics of the protocol. They have the form

$$l \; -[\; a \;]\!\rightarrow \; r$$

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 model the adversary knowledge, which increases over time. The built-in 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. A trace is the sequence of labels in an execution, but skipping empty labels $\emptyset$. Security properties are expressed in a first-order logic with quantification over terms and time points (indexes in the trace). The atom $f(\ldots)@i$ expresses a fact $f(\ldots)$ occurs at time point i. The other atoms are term equality and comparison of time points.

	Chrome [10], [11]	Edge[2]	Firefox (Desktop)[12]	Firefox (Mobile)[12]	Safari [13]	Brave[3] [14]
CT supported / enforced	✓	✓	✓	✗	✓	✓
client-side inclusion check	✗	—	✗	✗	✗	
SCT Audit [11]	●[1]	—	✗	✗	✗	
# of required SCTs	2 [4]	—	2 [4]	—	2 [4]	2 [3,4]
# of trusted loggers	7	—	7	—	8	7–8 [3]

[1] for random set of certificates [2] no public policy, hence tested or unknown (?) [3] follows policy of Apple and Chromium, depending on platform
[4] 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 [20]. Their definition is protocol agnostic and based on causality; a protocol provides accountability for property φ if it can always correctly identify the parties whose deviation from the protocol (jointly) caused [21] a violation $\neg\varphi$. Consequently, iff the protocol identifies no such set of parties, then φ must hold, hence accountability for φ implies verifiability for φ.

In this context, CT's task is to find out whether φ 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 under-specified. Our task is to figure out these conditions (we call them *tests*) and the precise property φ.

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. We describe, in set of rules that anyone can execute, a judgment procedure that emits the event `DocumentedIntermediateCA` if a root CA can provide the documents attesting its identity and `UndocumentedIntermediateCA` if not. Root CAs have to publish which intermediate CAs they signed [22], [23], [24] into the publicly accessible CCADB database [25]. Entries contains the intermediate CA's identity and public key and proofs for their authenticity.

Depending on whether the intermediate CA is legitimate either the root CA or the intermediate CA is blamed.

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

test CAfalseAsserts_rootBlamed:
"∃ idCA [..].
  AssertsNot(idCA, stmt, ca_fields)@t0
  // ...and was signed by illegitimate CA.
   ∧ UndocumentedIntermediateCA(rootCA, idCA,
↳ ca_fields)@t1"
```

Observe that the underlined variables `idCA` and `rootCA` are unbound. 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 φ) and did so by deviating from their normal behavior (i.e., the protocol). For instance, if t is such that three independent attacks happen, two matching the first test and one matching the second, then the verdict identifies three singleton causes, two of them 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, φ 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 combined 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, φ 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 in this model and less intuitive, hence we omit their explanation and instead refer the reader to [20]. Tamarin checks them via syntactic conditions (BR and RP in [20]). 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 Section C.

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. This validator must always be online and each client has to trust it. An unrealistic assumption, however, it helps transition to 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 the new infrastructure. Each time, efficiency improves or parties have better incentives to remain honest.

We use standard function symbols $\mathsf{pk}/1$, $\mathsf{sign}/2$ and $\mathsf{verify}/3$ and the equation $\mathsf{verify}(\mathsf{sign}(x,y),x,\mathsf{pk}(y)) = \mathsf{true}()$ 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 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 that the tuple contains. When set to 'pubKey-CA' or 'pubKey', it is a regular public key certificate 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., the subject is an intermediate CA. For 'pubKey', the certificate is a leaf, i.e., the subject it an end entity, e.g., a web server. 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 the following:

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($rootCA,pkRootCA,root_fields,
                    root_self_sig), // check 1
  // incoming certificate chain
  In(CA_fields), In(ca_sig),
  In(pub_fields), In(pub_sig),
] –[
 ClientAcceptsChain($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, $j),
 ↳ CheckValidUntil($now, $j0), // check 4
  ]↦ [ ]
```

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 φ.

Modeling Timestamps. For check 3), we need to model the two timestamps 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. [26] and model timestamps as public values that obtain a meaning through an axiomatic model. Using public variables ensures the adversary can access all timestamps, thereby also simplifying Tamarin's message deduction.

Every rule that needs to measure time obtains an event `Time($now)`. We relate timestamps with the following restriction. Restrictions are properties that Tamarin assumes to hold.

```
restriction time_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 an 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 pubic key, or, put more generally, that the statement of a certificate, i.e., fields (1—3) and (5), is not forged.

```
lemma authenticity:
"∀ 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 all CAs are honest (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 fingerprints. 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:

Listing 1. PKI provides acc. auth. assuming a trusted external validator.

```
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
 ∧ MonitorChecksCCADB(rootCA, ca_fields)@t1
 ⇒  // then the statement is correct
 (∃ idCA #t. Asserts_PKI(idCA, stmt)@t)"
```

Recall that root CAs document the intermediate CAs they sign in the CCADB (see Section 3.2).

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 time of 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 embedded 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 Section 2) to check new certificates that have the domain owner as subject against a *ground truth*, i.e., a set of tuples $(issuer, subject, public\ key, serial\ number)$. Recall that we call this tuple the certificate's *statement*: the *issuer* asserts the *subject* has a specific *public key* and this statement carries that *serial number* (for uniqueness). Following the certification guidelines in CT, honest CAs assert the statement on pre-certification in an event GroundTruth we we can use to define rogue certificates (and, analogously, benign certificates) [3].

```
restriction rogueCert: "∀ idCA stmt #t1.
RogueCert(idCA, stmt)@t1 ⇒
 ¬(∃ #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 Section 2, 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$(id_{log}, \ldots)$ represents additions to the log maintained by id_{log} and obtains a semantics *axiomatically*, through a restriction on traces. For the single-ledger case, earlier work [27], [28] 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 [29] 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 needs 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 [30]. 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, #now, commit_ts) <=>
(∃ sct_ts #t.
/* l was added to L's log at timepoint in SCT,
```

```
   which is the start of MMD period
   LoggerComputesAddLogTime(L, l, f, sct_ts)@t
        ∧ #t < #now
/* ...MMD period has passed */
        ¬(commit_ts = sct_ts)
        )
```

This rule takes account of the MMD by considering the time of the addition to the log and the time of the snapshot. As a simplification, we consider every action with the same `Time($i)` fact to belong to the same MMD period and every later time stamp to be after it. 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 D.7 and Section D.8; here, we only quote the failing case for inclusion proofs.

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

Accountable Authenticity. As discussed before, we assume the monitor knows the ground truth. We thus add a rule permitting it to raise an alarm. Nevertheless, the monitor needs to be active to perform this check. We add this 'liveness' requirement as a premise in the target property of the accountability lemma. Whenever a client accepts a certificate's statement (as before), but the monitor also performed the checks, then the client can be sure they know the ground truth (or someone can be held accountable failure). We make the checks the monitor needs to perform explicit: the monitor needs to fetch and inspect the log and check for information regarding the intermediate CA in the CCADB database. This asserts the checks are performed, but not that they pass: for instance, the monitor might find no or incorrect information in the CCADB databse, which then informs the judgement in the tests.

```
lemma A_CT_1 [heuristic=O "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
        ∧ MonitorFetchesLogSnapshot(idM, idL,
 ↳ session)@t1
        ∧ MonitorInspectsLog(idM, idL, stmt,
 ↳ ca_fields, commit, session)@t2
        ∧ MonitorChecksCCADB(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
        ∧ MonitorFetchesLogSnapshot(idM, idL,
 ↳ session)@t1
        ∧ MonitorBlamesCA_CT(idCA, idL, stmt,
 ↳ ca_fields, commit, session)@t2 /* idCA blamed
 ↳ */
        ∧ DocumentedIntermediateCA(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 [31]. Given CT's name, one could even argue that user's might expect this property.

In Section D.9, we formulate transparency as a standalone 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 [9] afterwards.

6. SCT auditing

SCT auditing was adapted as an opt-in mechanism in Google Chrome [11], [32]. 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 they visited 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 E.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
  ∧DocumentedIntermediateCA(rootCA, ca_fields,
 ↳ idCA)@t1"

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

Listing 2. SCT auditing provides acc. auth. against untrusted loggers and CAs.

```
lemma A_CTAudit_1:
ca_rogue_audit, root_ca_rogue_audit account for
"∀ ca_fields stmt rootCA idL #t0 #t1.
    AuditAuthCheck(idL, ca_fields, stmt)@t0
    // For every certificate that is audited to
 ↳ the monitor
    ∧ MonitorChecksCCADB(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, and uniquely for SCT auditing, the monitor can use the inclusion check to additionally hold loggers accountable (e.g., for third parties), despite (as we have just seen) not itself relying on the logger. Chrome's design documentation hints that this might take place: 'Google will [..] raise an alert if any log misbehavior is detected. The server-side auditing and alerting is covered in a separate Google-internal doc' [11]. 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
        ∧ MonitorFetchesLogSnapshot(idM, idL,
↳ session)@t1
        ∧ MonitorFindsCert(idM, idL, stmt, commit,
↳  session)@t2
        ∧ DocumentedIntermediateCA(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
        ∧ MonitorFetchesLogSnapshot(idM, idL,
↳ session)@t1
        ∧ CannotFindCert(idM, stmt, idL, commit,
↳ session)@t2
        ∧ DocumentedIntermediateCA(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.

Listing 3. SCT auditing provides acc. auth. assuming a trusted monitor.

```
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 */
        ∧ MonitorFetchesLogSnapshot(idM, idL,
↳ session)@t1
        [ remainer literally like A_CT_1 ]
        ∧ MonitorInspectsLog(idM, idL, stmt,
↳ commit, session)@t2
        ∧ MonitorChecksCCADB(rootCA, ca_fields)@t3
        ∧ ProofRunsAfterMMD(idL, commit, session)
        ∧ #t0 < #t1
            ⇒
                (∃ idCA #t. GroundTruth(idCA, stmt
↳ )@t)"
```

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 by definition that transparency for this certificate holds. (See Section E.3 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 rule 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 by 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 accountable. 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 no 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. [9] 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 send to the monitor (comparable to *STH Pollination* [9]) or where the monitor doubles as a trusted auditor (*Trusted Auditor Relationship* [9]). Compared to *STH Pollination* [9] (or *SCT Feedback* [9] 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 [8], [33] 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.

Modelling. We add a new rule that broadcasts the client's STH to the monitor. The monitor performs similar authenticity checks to CT, using the entries on the log to find potentially rogue certificates. In fact, we can reuse the same rules for authenticity checks in gossiping too. What is new, however, is that the monitor receives a gossiped STH from a client. For this STH, inclusion of a specific entry may have already been proven by the client. To make use of this information, we add new rules to check append-only between the gossiped STH and a new fetched snapshot from the log. So far, we only considered append-only checks between snapshots that have been stored by the monitor or client respectively. In this variant, one such STH is received from the network.

In Section G we provide the modified rules for this. They emit `Gos_AppendOnlyViol` if the append-only check between the gossiped STH and the newly fetched snapshot fails.

As we reuse the monitor authenticity checks on the log used in CT here, we distinguish between whether the certificate contradicts the known key for its subject or not and blame the CA in the first case, and, assume the involved monitor to be honest.

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

Listing 4. Refuted lemma: STH gossiping does not provide acc. auth. if the logger is not trusted.

```
lemma A_CTGossip:
    ca_rogue_gossip, root_ca_rogue_gossip
    account for
"∀ idM idL [..] .
   ClientSideInclusionProved(idL, stmt, gos_sth,
 ↳ gos_session)@t0
   ∧ MonitorFetchesLogSnapshot(idM, idL, session)
 ↳ @t1
   ∧ MonitorInspectsLog(idM, idL, stmt, ca_fields,
 ↳  sth, session)@t2
   ∧ MonitorChecksCCADB(rootCA, ca_fields)@t3
   ∧ Gos_AppendOnlyCheck(idM, idL, sth, session,
 ↳ gos_sth, gos_session)@t4
   ∧ #t0 < #t1
   ⇒ (∃ 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, gos_sth,
 ↳ gos_session)@t0
  ∧ MonitorFetchesLogSnapshot(idM, idL, session)
 ↳ @t1
  ∧ MonitorBlamesCA(idCA, idL, stmt, ca_fields,
 ↳ sth, session)@t2
  ∧ DocumentedIntermediateCA(rootCA, ca_fields,
 ↳ idCA)@t3
  ∧ Gos_AppendOnlyCheck(idM, idL, sth, session,
 ↳ gos_sth, gos_session)@t4 ∧ #t0 < #t1"
```

Accountable authenticity does not hold; we find an attack against verifiability (see Lst 4): a malicious logger alone can bypass the test by first sending the client a correct certificate and an STH that proves inclusion, but then creating a new, incorrect, refutable inclusion proof for a different snapshot that does not contain this certificate. That proof will fail and the monitor realise that something is wrong—and correctly blame the logger, see next section. But without the actual certificate, the monitor cannot decide if the CA is involved or not, or if authenticity was actually violated by whatever certificate the client accepted. So, in contrast to CT, sharing STHs with monitors is not sufficient to achieve accountable authenticity, but sufficient for the monitor to detect equivocation.

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 the append-only property. We derive a test that instead achieves accountable transparency.

```
test gossip_inclusionViol:
    "∃ idM [..].
    MonitorFetchesLogSnapshot(idM, idL, session)
↳ @t1
    /* monitor starts own inclusion proof */
    ∧ InclusionProofFails(idM, idL, preCert,
↳ commit, session)@t2
    ∧ Gos_AppendOnlyViol(idM, idL, commit, session
↳ , gos_commit, gos_session)@t3
    /* the monitor finds that append-only between
↳ sth and commit is
    violated and blames the logger */
    /* but inclusion in this commit is not
↳ fulfilled */
    ∧ InclusionProofSucceeds(idL, preCert,
↳ gos_commit, gos_session)@t4"
    /* We require that the gossiped sth contains
↳ the preCert */
```

Now we can show that accountable transparency holds. (See Section G 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. Many of are marked as reusable in Tamarin, so they are used also used as helping lemmas. Overall, they 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.

Transparency. Table 2 also confirms the intuition that transparency is essential for achieving accountable authenticity—unless we can trust the logger. To hold CAs accountable, log entries must include the full certificate including the fields signed by the CA. Without them, loggers may be able to wrongly accuse CAs. More generally, *what data* is made transparent determines who can be held accountable: full certificates enable accountability of CAs, SCTs suffice to hold loggers accountable for omissions. Observe that SCT auditing covers both. STH gossiping, however, reveals STHs to monitors, but not CA signatures, so CAs cannot be held accountable. This explains the gap: while STH gossiping can provide accountable transparency, which could be a building block for accountable authenticity, it fails to achieve accountable authenticity because it can only blame loggers correctly, but not CAs.

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 [11]. Nevertheless, we can state that every handshake that is actually audited enjoys these guarantees (Lst 3).

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. But they could be exploited to more effect. In order to not get caught equivocating, the logger has to prove inclusion to clients but hide the entry from the monitor. Right now, the logger can easily distinguish client from

protocol	authenticity with trust in …				transparency
	noone	monitor	+ logger	+ external validator	(no trust)
PKI	—	—	—	● (Lst 1)	—
CT	✗	✗ (Lst 9)	● (Lst 10)	● (Lst 8)	✗ (Lst 5 and 7)
SCT Auditing	✗ (Lst 11 +receipt: ●, Lst 13)	● (Lst 2)	● (Lst 10)	● (Lst 8)	✓ (Lst 12)
Gossiping	✗	✗ (Lst 4)	● (Lst 10)	● (Lst 8)	● (Lst 14 to 16)

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

TABLE 2. RESULTS: ACCOUNTABILITY AND TRANSPARENCY IN CT.

monitor, because the client demands an inclusion proof, whereas the monitor fetches new entries (to perform its tests) and then recomputes the STH. Monitors could instead mirror the client's behavior to obtain the STH as a commitment on the current state of the log.

As of now, authenticity and transparency still rely on trust in the loggers. All browser require $n = 2$ SCTs from distinct loggers, these can be different for each certificate, but must come from a pool of $N = 7$ or $N = 8$ 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). Currently $N-(n-1)$ equals 6 or 7, depending on the browser. The logger pool is almost the same, so an attack would be largely independent of the browser.

9. Related Work

So far, formal analyses of CT have either been conducted manually, in the cryptographic standard model [34], or with automated tools in the Dolev-Yao model but with significant simplifications [20], [29].

Cryptographic analyses of CT. Wrótniak et al. [34] 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 validates but does not point to a 'responsible' CA, where a resposible CA is a trusted root CA that has signed the rogue certificate's fields or a CA that has 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 hard 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. Compared 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 or to blame more than one. Many works on accountability in protocol-like settings highlight the ability to blame (all) the responsible participants [35], [36], [37].

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 motivated the present analysis and helped formulate some tests (see below).

Automated Analysis of Accountability in CT. CT was first formally verified for accountability by Bruni et al. [27]. 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 may miss violations and might blame a party for a violation of a security property that cannot be violated. We do not further extend their model but instead obtain a broader and much more realistic model that captures the details of the actual specification while continuing to support an unbounded number of participants.

Cheval et al. [29] analyzed CT in ProVerif. Their focus is on accurately modeling the Merkle tree structure. They extend ProVerif to support an axiomatic modeling of ledgers. Besides CT, ledgers are also used in voting systems [38]. 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 5) 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).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. [34]'s model also considers SCT auditing [34, CTwAudit, Sec. D.2] (see Section 6). There, SCT auditing is given as an algorithm that, given a certificate with SCTs, returns the identity of the loggers that are deemed responsible for the lack of transparency should the corresponding entry not be 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 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. Logging the whole certificate and chain, log entries provide enough evidence to hold CAs accountable for the certificates they issued. Corrupt loggers can hide evidence, which constitutes a known attack against accountable authenticity and transparency. If one of the loggers in use is honest, transparency holds, but only for that particular logger. An honest monitor that knows the ground truth—every domain owner can be a monitor— 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 loggers (see Table 1) with almost[1] complete intersect. With Chrome's high market share, and since certificates are transmitted before the user agent is known, there is no incentive for websites to include additional loggers, let alone exclude loggers from Chrome's list. Essentially, Google decides which set of loggers is trusted, even for other browsers. Google's loggers are part of this set.

1. Currently, IPnG is included in some but not all

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 counteract the inherent tendency for centralization in SCT auditing.

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. 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 work, 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 [39] or similar entries [40] 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 we trust the OCSP responder. Future work should investigate possible synergies between certificate transparency and revocation. In Section H, we provide a simple model as evidence that this is worthwhile. If in CT, the subject were to revoke a rogue certificate the moment the monitor informs them about it and if the subject notified all monitors of this fact, then we could hold the monitor, log and server accountable for the authenticity of the certificate including its revocation status —provided an honest log or, alternatively, SCT Auditing.

11. Ethics considerations

None.

12. LLM usage considerations

No LLM usage.

References

[1] 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.

[2] 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-certficates.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] "Merkle Town," Accessed: Jul. 20, 2025. [Online]. Available: https://ct.cloudflare.com/.

[8] 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.

[9] 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.

[10] "Chrome Certificate Transparency Policy," CertificateTransparency, Accessed: May 27, 2025. [Online]. Available: https://googlechrome.github.io/CertificateTransparency/ct_policy.html.

[11] C. Thompson and E. Stark, "Opt-in SCT Auditing (public)," Sep. 30, 2020. [Online]. Available: https://docs.google.com/document/d/1G1Jy8LJgSqJ-B673GnTYIG4b7XRw2ZLtvvSlrqFcl4A/.

[12] "SecurityEngineering/Certificate Transparency - MozillaWiki," Accessed: Apr. 2, 2025. [Online]. Available: https://wiki.mozilla.org/SecurityEngineering/Certificate_Transparency.

[13] "Apple's Certificate Transparency policy," Apple Support, Accessed: May 27, 2025. [Online]. Available: https://support.apple.com/en-us/103214.

[14] "TLS Policy," GitHub, Accessed: May 27, 2025. [Online]. Available: https://github.com/brave/brave-browser/wiki/TLS-Policy.

[15] *SSLMate/certspotter*, SSLMate, Jul. 8, 2025. Accessed: Jul. 9, 2025. [Online]. Available: https://github.com/SSLMate/certspotter.

[16] C. Project. "Monitors : Certificate Transparency," Accessed: Jul. 9, 2025. [Online]. Available: https://certificate.transparency.dev/monitors/.

[17] 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.

[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.

[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.

[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] "Chrome Root Program Policy, Version 1.7," Accessed: Nov. 2, 2025. [Online]. Available: https://googlechrome.github.io/chromerootprogram/#2-common-ca-database.

[23] "Root Certificate Program - Apple," Accessed: Nov. 2, 2025. [Online]. Available: https://www.apple.com/certificateauthority/ca_program.html.

[24] "Mozilla Root Store Policy," Mozilla, Accessed: Nov. 2, 2025. [Online]. Available: https://www.

mozilla . org / en - US / about / governance / policies / security-group/certs/policy/.
[25] "Common CA Database by the Linux Foundation," Accessed: Oct. 14, 2025. [Online]. Available: https://www.ccadb.org/policy#5-audit-disclosures.
[26] 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.
[27] 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.
[28] 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.
[29] 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.
[30] S. A. Crosby and D. S. Wallach, "Efficient Data Structures for Tamper-Evident Logging,"
[31] 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.
[32] "SCT Auditing Google Source," Google. [Online]. Available: https://chromium.googlesource.com/chromium/src/+/refs/heads/main/services/network/sct_auditing/.
[33] "Expect-CT header - HTTP | MDN," Accessed: Jun. 30, 2025. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers/Expect-CT.
[34] 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.
[35] 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.
[36] 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.
[37] 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.
[38] 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/.
[39] 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.
[40] 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.
[41] "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
        fields = <$rootCA, $idCA, pk(~skCA), '
↳ pubKey-CA', ~sn, $now, $j>
        self_sig = sign(fields, skRootCA)
    in
    [ Fr(~skCA), Fr(~sn), !Root_CA_s0($rootCA,
↳ skRootCA, pkRootCA) /* Root CA that signs this
↳ CA */  ]
    -[
        CA_Registered($idCA, ~skCA, pk(~skCA)),
        NonRoot_CA_Init($idCA),
        DocumentedCA($rootCA, $idCA, fields),
        Time($now)
    ]->
    [   Out(fields), Out(self_sig),
        Out(<$idCA, pk(~skCA)>), /* publicly
↳ announce the public key and certificate on the
↳ network */
        !CA_s0($idCA, ~skCA, pk(~skCA), fields,
↳ self_sig), !CA_Documentation($rootCA, $idCA,
↳ fields, 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 [41] 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_fields = <$idCA, $id, pkID, 'pubKey',
↳  ~sn, $now, $j>
        cert_sig = sign(cert_fields, skCA)
        cert = <cert_fields, cert_sig>
    in
    [ !DomainOwner($id, pkID),
     !CA_s0($idCA, skCA, pkCA, ca_fields, CA_sig),
↳  Fr(~sn) ]
    -[ Certified($idCA, cert),
        GroundTruth($idCA, <$idCA, $id, pkID, ~sn
↳ >),
        Asserts_PKI($idCA, <$idCA, $id, pkID, ~sn
↳ >),

        CAAsserts($idCA, <$idCA, $id, pkID, ~sn>),

        Time($now)
    ]->
    [ Out(cert_fields), Out(cert_sig), Out(
↳ ca_fields), Out(CA_sig) /* announces full chain
↳  on the network */]
```

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 for each CA kind, 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($now)
    ]->
    [ Out(skCA) ]

rule compromiseRootCA:
    [ !Root_CA_s0($idCA, skCA, pkCA) ]
    -[
        Corrupted($idCA),
        Time($now)
    ]->
    [ 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

			proof details			
type	model	assumption 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
	Audit		—	✓	3.42	8
	Gossip	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
	Audit		SUFF	✓	20.06	167
	Gossip		SUFF∗,+	✓	12.86	26
AA	PKI Ext.	external validator	—	✓	23.82	50
	CT Ext.	monitor validator	—	✓	27.92	58
		client blames log	SUFF	✓	167.72	1736
		monitor blames log	SUFF∗	✓	205.88	1279
	CT	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.

	proof details	
Lemma	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.

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.

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 which 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 D.3.

```
rule Monitor_Init:
    [  ] −[ Time($now), Monitor_Registered($idM) ]→
 ↳  [ !Monitor($idM) ]

rule Logger_Init:
    [ Fr(~skL) ]
    −[
        Logger_Registered($idL),
        Time($now)
    ]→
    [ !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 start of tracking a specific subject.

```
rule StartMonitoring:
    [ !Monitor($idM), !Logger($idL, pkL) ]
    −[
        Time($now)
    ]→
    [ !LoggerDict($idM, $idL) ]


rule monitorTrackSubject:
    [ !Monitor($IdM), !DomainOwner(su, pubK) ]
    −[ Time($now) ]→
    [ !TrackSubject($IdM, su) ]
```

```
rule PreCertify: /* CA issues a pre-certificate */
    let
        preC_fields = <$idCA, $id, pkID, 'pre-cert
↳ ', ~sn, $now, $j>
        preC_sig = sign(preC_fields, skCA)
        preC = <preC_fields, preC_sig>
    in
    [ !DomainOwner($id, pkID),
    !CA_s0($idCA, skCA, pkCA, ca_fields, ca_sig),
↳ Fr(~sn) ]
    -[ PreCertified($idCA, preC),
        GroundTruth($idCA, <$idCA, $id, pkID, ~sn
↳ >),
        Asserts_PKI($idCA, <$idCA, $id, pkID, ~sn
↳ >),

        Time($now)
    ]->
    [ Out(preC_fields), Out(preC_sig),
↳ Certificate_Store($idCA, preC), Out(ca_fields),
↳  Out(ca_sig) ]

/* CA embeds both SCTs in the pre-cert */
rule PubKeyCertify:
    let
    preC_fields = <$idCA, $id, pkID, 'pre-cert',
↳ sn, $i, $j>

    sct1_fields = <$idL1, 'SCT', $sct1, <$idCA,
↳ $id, pkID, sn, $i, $j>>
    sct2_fields = <$idL2, 'SCT', $sct2, <$idCA,
↳ $id, pkID, sn, $i, $j>>

    sct1 = <$idL1, 'SCT', $sct1, sct1_sig>
    sct2 = <$idL2, 'SCT', $sct2, sct2_sig>

    pub_fields = <$idCA, $id, pkID, 'pubKey', sn,
↳ $i, $j, sct1, sct2>
    pub_sig = sign(pub_fields, skCA)
    in
    [
        !CA_s0($idCA, skCA, pkCA, ca_fields,
↳ CA_sig),
        /* 2 SCTs: */  In(sct1), In(sct2),
        !Logger($idL1, pkL1),
        !Logger($idL2, pkL2),
        Certificate_Store($idCA, <preC_fields,
↳ sig_preC>)
    ]
    -[ Eq(verify(sct1_sig, sct1_fields, pkL1), true
↳ ()),
        Eq(verify(sct2_sig, sct2_fields, pkL2),
↳ true()),
        Eq(verify(sig_preC, preC_fields, pkCA),
↳ true()),
        _restrict( not ($idL1 = $idL2) ), /* the 2
↳  SCTs were from distinct loggers */

        Certified($idCA, $idL1, sct1, $idL2, sct2)
↳ ,
        CAAsserts($idCA, <$idCA, $id, pkID, sn>),

        Time($now),
        CheckValidUntil($now, $j)
    ]->
    [ Out(pub_fields), Out(pub_sig) /* outgoing
↳ pubKey cert */ ]
```

4.3. Log Data Structure.

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 [26]. For example, the action `LoggerComputesAddLog($LogIdentity ↳ , cert)` represents additions to the log maintained by `$LogIdentity`.

A similar approach has been used before by Bruni et al. [27] and extended by Künnemann et al. [28] 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 known root certificate. There are different notions in literature whether the root certificate itself is considered a part of the chain or not. In our model, the root CA certificate is not part of the chain as every participant in the system has access to them anyways. Then, it issues the corresponding SCT and adds the chain and pre-certificate to its publicly visible log. Following [3], the entry on the Merkle tree later (which we interchangeably call log in the following) only contains the pre-certificate without the signature. Note that this has practical reasons, as later with inclusion proofs, the requesting party needs to specify which entry it wants to receive a proof of inclusion. The party might not have the full chain available, nor is it guaranteed that the exact same chain is stored in the log (think of corruption cases). The promise by the log that clients receive as SCT is only computed over the pre-certificate fields and not a chain. Thus we can only hold a logger liable for the pre-certificate fields and not the complete chain. However, [3] also specifies that loggers MUST store the full chain for auditing purposes. Our action fact `LoggerComputesAddLog($idL, ↳ $log_id, <<preC_fields, sig_preC>, < ↳ ca_fields, ca_sig>>, $now)`, thus contains both the pre-certiificate fields and the full chain. Inclusion proofs are later formalized over the pre-certificate fields while monitors inspecting the full log will also get access to the full chain.

```
rule add_preC_to_log:
    let
        preC_fields = <$idCA, $id, pkID, 'pre-cert
↳ ', sn, $i, $j>
        ca_fields = <$rootCA, $idCA, pkCA, 'pubKey
↳ -CA', snCA, $i0, $j0>

        sct_fields = <$idL, 'SCT', $now, <$idCA,
↳ $id, pkID, sn, $i, $j>>
        sct_sig = sign(sct_fields, skL)
        sct = <$idL, 'SCT', $now, sct_sig>

        chain = <<preC_fields, preC_sig>, <
↳ ca_fields, ca_sig>>
```

```
    in
    [
        In(preC_fields), In(preC_sig), In(
↳ ca_fields), In(ca_sig), /* incoming pre-cert
↳ chain */
        !Logger_s($idL, skL, pk(skL)),
        !Root_CA($rootCA, pkRootCA),
        !Log($idL, $log_id)
    ]
    −[
        Eq(verify(preC_sig, preC_fields, pkCA),
↳ true()),
        Eq(verify(ca_sig, ca_fields, pkRootCA),
↳ true()),
        LoggerComputesAddLogStmt($idL, $log_id, <
↳ $idCA, $id, pkID, sn>, $now),
        LoggerComputesAddLog($idL, $log_id,
↳ preC_fields, chain, $now),
        LoggerAsserts($idL, sct),

        Time($now),
        CheckIssuedTime($now, $i),
        CheckValidUntil($now, $j)
    ]→
    [ Out(sct) ]
```

4.4. 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 three 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.
3) Entry Validity: Every entry in the log corresponds to a valid pre-certificate with a full chain ending in a trusted root.

We add the ability to break precisely these 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($idL, ~skL, pk(~skL)) ]
    −[ StartedLog($idL, $log_id),
        StartedLogMalicious($idL, $log_id),
        Time($now)
    ]→
    [  ]

rule forge_log_view [color= #d40808]:
    let
        preC_fields = <$idCA, $id, pkID, 'pre-cert
↳ ', sn, $i, $j>
        ca_fields = <$rootCA, $idCA, pkCA, 'pubKey
↳ -CA', snCA, $i0, $j0>
    in
    [ In(preC_fields), In(preC_sig), In(ca_fields)
↳ , In(ca_sig),
      /* Incoming certificate that is added */
      !Corrupted_Logger($idL, skL, pk(skL)) ]
    −[
        LoggerComputesAddLogStmt($idL, $log_id, <
↳ $idCA, $id, pkID, sn>, $now),
        LoggerComputesAddLog($idL, $log_id,
↳ preC_fields, <<preC_fields, preC_sig>, <
↳ ca_fields, ca_sig>>, $now),
        LoggerForgesAdd($idL),
        Time($now)
    ]→
    [ ]
```

The forge log entry rule allows corrupted loggers to add arbitrary entry to any of their logs without possessing a valid pre-certificate chain.

4.5. Visibility Predicates and Maximum Merge Delay.

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` action fact. Scheduled additions are pending and, in the sense of MMD, become visible to inspecting parties after the delay has passed.

Visibility of Merkle Tree Leafs. 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. Likewise, an entry is not visible if it has never been added. We formalize visibility as predicates based on the `#time` of snapshot creation.

```
predicate: EntryVisible(Logger, preC_fields,
↳ log_id, #time, commit_time) <=>
    (∃ different_time chain #t.
        LoggerComputesAddLog(Logger, log_id,
↳ preC_fields, chain, different_time)@t /* there
↳ has been an addition to log_id with the
↳ preC_fields prior to #time */
        ∧ ¬(commit_time = different_time) /* ...
↳ that addition was visible */
        ∧ #t < #time
    )

predicate: EntryMissing(Logger, preC_fields,
↳ log_id, #time, commit_time) <=>
    not (∃ different_time chain #t.
            LoggerComputesAddLog(Logger, log_id,
↳ preC_fields, chain, different_time)@t
            ∧ not (commit_time = different_time)
↳ /* or there has never been a visible addition
↳ to the log */
            ∧ #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` 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.

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

Note here that these predicates are not concerned about the chain. The visibility predicate is exclusively a property defined on the Merkle Tree of the log. We will reuse these restrictions to formulate inclusion proofs on that Merkle Tree. Following [3], an inclusion proof alone does not provide the recipient with any guarantees on the chain validity or existence.

Visibility of Chains. A monitor that inspects a log is usually interested in more than just a proof of inclusion of a pre-certificate. Think of the case where a monitor finds a potentially rogue pre-certificate entry included in the Merkle Tree structure. Without a chain of certificates it is impossible to proof which CA issued the certificate and thus who is to blame. For this case, we adapt the visibility predicates to capture the pre-certificate and its chain together.

```
predicate: ChainVisible(Logger, chain, log_id, #
↳ time, commit_time) <=>
   (∃ preC_fields different_time #t.
       LoggerComputesAddLog(Logger, log_id,
↳ preC_fields, chain, different_time)@t /* there
↳ has been an addition to log_id prior to #time
↳ */
       ∧ ¬(commit_time = different_time) /* ...
↳ that addition was visible */
       ∧ #t < #time
       )

predicate: NoChainVisible(Logger, stmt, log_id, #
↳ time, commit_time) <=>
   not (∃ different_time #t.
           LoggerComputesAddLogStmt(Logger,
↳ log_id, stmt, different_time)@t
           ∧ not (commit_time = different_time)
↳ /* or there has never been a visible addition
↳ to the log */
           ∧ #t < #time
       )
       /* OR: for all additions with cert_fields,
↳  the chain is invalid */
       ∨ ((∀ preC_fields different_time stmt
↳ chain #t2 #t.
           LoggerComputesAddLogStmt(Logger,
↳ log_id, stmt, different_time)@t
           ∧ LoggerComputesAddLog(Logger, log_id,
↳  preC_fields, chain, different_time)@t
           ∧ #t < #time
           ∧ ChainCheck(chain)@t2
           ⇒ ChainInvalid(chain)@t2
       ) ∧ (∀ different_time preC_fields chain #t
↳ .
           LoggerComputesAddLog(Logger, log_id,
↳ preC_fields, chain, different_time)@t
           ∧ #t < #time
           ⇒ (∃ #t2. ChainCheck(chain)@t2)
       ))
```

These last two predicates might appear odd at first sight. In Section D.12 we will make use of them to distinguish whether a monitor can find a valid chain for a given pre-certificate entry in the log or not. By construction and if a monitor additionally verifies a found chain themselves, a monitor is guaranteed to find a rogue certificate with valid chain, if it exists in the log. This makes our model as expressive as a real-world CT monitor.

4.6. 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 D.7 and Section D.8, 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_M:
    [ Fr(~session), !Monitor($idM), !Logger($idL,
↳ pkL) ]
   -[
       FetchLogSnapshot($idM, $idL, ~session),
       MonitorFetchesLogSnapshot($idM, $idL, ~
↳ session),
       Time($now),
   ]->
   [ !TLS_M_to_L($idM, $idL, ~session) ]
```

We add a `~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. `$log_id` is dynamically instantiated and by that potentially attacker controlled. We restrict that the instantiated `$log_id` corresponds to a log that the logger

maintains, i.e., the id exists. As honest loggers only maintain a single log, an attacker can only interfer here if the logger is corrupted too. 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 `~commitment` as well as the current time.

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.

```
rule LoggerSnapshot_M:
    [ !Logger_s($idL, skL, pk(skL)), Fr(~
↳ commitment) /* abstract representation of the
↳ STH/commitment */, !TLS_M_to_L($idM, $idL,
↳ session) ]
   −[
        LoggerPresentsView($idL, $idM, $log_id,
↳ $now, ~commitment, session),
        LoggerHandsOutSTH($idL, ~commitment,
↳ session),

        Time($now)
   ]→
    [ !TLS_L_to_M($idL, $idM, $snapshot, ~
↳ commitment, 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. This secure channel has a session identifier that is used to relate the steps of the proof run.

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.

4.7. Inclusion Proof. We formulate inclusion proofs using the visibility predicates presented in Section D.5. 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:
    "∀ IdFetcher IdL preC_fields commit session #
↳ t1.
    InclusionProofFails(IdFetcher, IdL,
↳ preC_fields, commit, session)@t1
    ⇒ (∃ log_id time #t0.
        LoggerPresentsView(IdL, IdFetcher, log_id,
↳  time, commit, session)@t0
            /* Logger handed out an STH to the
↳ recipient... */
        ∧ #t0 < #t1 /* ...prior to the check */
        ∧ EntryMissing(IdL, preC_fields, log_id, #
↳ t0, time)
            /* ...and the entry is not visible in
↳ that view */
        )"
```

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

```
restriction inclusionProofWorks:
    "∀ IdL preC_fields commit session #t5.
↳ InclusionProofSucceeds(IdL, preC_fields, commit
↳ , session)@t5
    ⇒ (∃ id log_id time #t2.
        LoggerPresentsView(IdL, id, log_id, time,
↳ commit, session)@t2
            /* Logger handed out a snapshot to the
↳  recipient... */
        ∧ #t2 < #t5
            /* ...prior to the check */
        ∧ EntryVisible(IdL, preC_fields, log_id, #
↳ t2, time)
            /* and the entry is visible in that
↳ view */
        )"
```

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

4.8. 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 should not be possible if entries have been removed from the log in between.

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 all additionally fetched snapshots. Similar to the inclusion proof, we use restrictions for that.

```
rule MonitorStoresSnapshot:
    [ !Logger($idL, pkL), !Monitor($idM), !
↳ TLS_L_to_M($idL, $idM, $snapshot, commit,
↳ session) ]
   −[
        SnapshotStored($idM, $idL, session, commit
↳ ),
        MonitorSnapshotStored($idM, $idL, session,
↳  commit),
        Time($now)
   ]→
    [  ]
```

The implementation for a client doing the same is analogous. The formalization of an unsuccessful append-only proof is given below.

```
restriction AppendOnlyFails:
```

```
  "∀ commit session id IdL #t6.
   AppendOnlyViolation(id, IdL, commit, session)
↳ @t6
   /* Append-only violation found by id with
↳ violating IdL is only possible if a
↳ counterexample is found */
       ⇒ (∃ preC_fields chain time t log_id1
↳ log_id2 time2 commit2 session2 #t0 #t1 #t2 #t3
↳ #t4 #t5.
                LoggerComputesAddLog(IdL, log_id1,
↳  preC_fields, chain, t)@t0
                ∧ EntryVisible(IdL, preC_fields,
↳ log_id1, #t2, time)
                /* there is an entry in the log
↳ that is visible at #t2 and time period time in
↳ log_id1 */
                ∧ FetchLogSnapshot(id, IdL,
↳ session2)@t1
                /* someone starts fetching a log
↳ view */
                ∧ LoggerPresentsView(IdL, id,
↳ log_id1, time, commit2, session2)@t2
                /* The Logger presents the view
↳ for log_id1 */
                ∧ SnapshotStored(id, IdL, session2
↳ , commit2)@t3
                /* this view is received stored by
↳  id */
                ∧ FetchLogSnapshot(id, IdL,
↳ session)@t4
                /* a second view is fetched by id
↳ (later) */
                ∧ LoggerPresentsView(IdL, id,
↳ log_id2, time2, commit, session)@t5
                /* the logger presents the view of
↳  a possibly different log (log_id2) */
                ∧ EntryMissing(IdL, preC_fields,
↳ log_id2, #t5, time2)
                /* one entry that was visible in
↳ the past is now not visible anymore -> clear
↳ append-only violation */
                ∧ #t0 < #t1 ∧ #t1 < #t2 ∧ #t2 < #
↳ t3
                ∧ #t2 < #t4
                ∧ #t4 < #t5 ∧ #t3 < #t5
                ∧ #t5 < #t6
           )"
```

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. The formalization is semantically analogous to the previous one by asserting that every entry that was visible before, should still be visible.

4.9. 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 (pre-certificate) 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_CT_attempt:
    "∀ id idL idL1 idL2 preCert commit session #t1
↳  #t2 #t3.
        ClientAcceptsCert(idL1, idL2, preCert)@t1
↳ /* preCert fields reconstructed from the
↳ accepted pubKey certificate */
        ∧ FetchLogSnapshot(id, idL, session)@t2
        ∧ InclusionProof(id, idL, preCert, commit,
↳  session)@t3
        ∧ ((idL = idL1) ∨ (idL = idL2))
        ∧ #t1 < #t2 ∧ #t2 < #t3
            ⇒
            InclusionProofSucceeds(idL, preCert,
↳ commit, session)@t3"
```

We are able to find a counterexample for this: The proof fails because the maximum-merge delay has not passed yet. This is a limitation that we will keep in the following lemmas. We always need to consider the MMD when reasoning about visibility in CT logs.

Listing 5. Refuted lemma: CT does not have transparency.

```
lemma transparency_w_MMD:
    "∀ id idL idL1 idL2 preCert commit session #t1
↳  #t2 #t3.
        ClientAcceptsCert(idL1, idL2, preCert)@t1
↳ /* preCert fields reconstructed from the
↳ accepted pubKey certificate */
        ∧ FetchLogSnapshot(id, idL, session)@t2
        ∧ InclusionProof(id, idL, preCert, commit,
↳  session)@t3
        ∧ ((idL = idL1) ∨ (idL = idL2))
        ∧ #t1 < #t2 ∧ #t2 < #t3
        ∧ ProofRunsAfterMMD(idL, commit, session)
            ⇒
        (∃ #t. InclusionProofSucceeds(idL, preCert
↳ , commit, session)@t)"
```

Now the only violation possible is that the logger is corrupted and presents a snapshot that does not include the entry.

If the logger is assumed honest, we can prove transparency in CT as shown below.

Listing 6. CT has transparency if logger is honest.

```
lemma transparency_CT_w_honest_MMD:
    "∀ id idL idL1 idL2 preCert commit session #t1
↳  #t2 #t3.
        ClientAcceptsCert(idL1, idL2, preCert)@t1
↳ /* preCert fields reconstructed from the
↳ accepted pubKey certificate */
        ∧ FetchLogSnapshot(id, idL, session)@t2
        ∧ InclusionProof(id, idL, preCert, commit,
↳  session)@t3
        ∧ ((idL = idL1) ∨ (idL = idL2))
        ∧ #t1 < #t2 ∧ #t2 < #t3
        ∧ ProofRunsAfterMMD(idL, commit, session)
        ∧ not (∃ #t. Corrupted(idL)@t)
            ⇒
        (∃ #t. InclusionProofSucceeds(idL, preCert
↳ , 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. Ultimately, we would like to achieve trans-

parency without the honest logger assumption, but in CT we cannot.

4.10. 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 client_inclusionViol:
    "∃ idL1 idL2 idClient preCert commit session #
↳ t0 #t1 #t2.
        ClientAcceptsCert(idL1, idL2, preCert)@t0
        ∧ ((idL = idL1) ∨ (idL = idL2))
        ∧ ClientFetchesLogSnapshot(idClient, idL,
↳ session)@t1
        ∧ InclusionProofFails(idClient, idL,
↳ preCert, commit, session)@t2
        ∧ ProofRunsAfterMMD(idL, commit, session)
        ∧ #t0 < #t1
    "
```

Listing 7. Refuted lemma: CT does not have accountable transparency.

```
test LoggerViolatesInclusion_client:
lemma T_CT_1:
    client_inclusionViol accounts for
        "∀ idL idL1 idL2 preCert commit session
↳ idClient #t0 #t1 #t2.
        ClientAcceptsCert(idL1, idL2, preCert)@t0
↳ /* (1)*/
        ∧ ((idL = idL1) ∨ (idL = idL2))
        ∧ ClientFetchesLogSnapshot(idClient, idL,
↳ session)@t1 /* (2.1) */
        ∧ InclusionProof(idClient, idL, preCert,
↳ commit, session)@t2
            /* (2.2)*/
        ∧ ProofRunsAfterMMD(idL, commit, session)
↳ /* (3) */
        ∧ #t0 < #t1
            ⇒
            InclusionProofSucceeds(idL, preCert,
↳ commit, session)@t2
            /* ...then the inclusion proof must
↳ suceed, in other words: the entry is visible in
↳  the snapshot and transparency holds. */
        "
```

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.

4.11. Accountable authenticity.

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 `CAFakesCert` action fact to the trace.

```
rule monitor_externalCheck_Fake:
    let
        stmt = <$idCA, $id, pkID, sn>
        pub_fields = <$idCA, $id, pkID, 'pubKey',
↳ sn, $i, $j, sct1, sct2>
        ca_fields = <$rootCA, $idCA, pkCA, 'pubKey
↳ -CA', snCA, $i0, $j0>
    in
    [   In(pub_fields), In(pub_sig), In(ca_fields)
↳ , In(ca_sig),
        !Root_CA($rootCA, pkRootCA), !TrackSubject
↳ ($idM, $id),
    ]
    -[ Eq(verify(pub_sig, pub_fields, pkCA), true()
↳ ),
        Eq(verify(ca_sig, ca_fields, pkRootCA),
↳ true()),
        RogueCert($idCA, stmt), /* see restriction
↳  */
        CAFakesCert($idCA, stmt, ca_fields),
        MonExtFakeCheck(ca_fields, stmt),
        Time($now)
    ]->
    [ ]

rule monitor_externalCheck_Verified:
    let
        stmt = <$idCA, $id, pkID, sn>
        pub_fields = <$idCA, $id, pkID, 'pubKey',
↳ sn, $i, $j, sct1, sct2>
        ca_fields = <$rootCA, $idCA, pkCA, 'pubKey
↳ -CA', snCA, $i0, $j0>
    in
    [   In(pub_fields), In(pub_sig), In(ca_fields)
↳ , In(ca_sig),
        !TrackSubject($idM, $id), !Root_CA($rootCA
↳ , pkRootCA)
    ]
    -[
```

```
        Eq(verify(pub_sig, pub_fields, pkCA), true
↳ ()),
        Eq(verify(ca_sig, ca_fields, pkRootCA),
↳ true()),
        BenignCert($idCA, stmt), /* see
↳ restriction */
        MonExtCheckVerified($idCA, stmt, ca_fields
↳ ),
        MonExtFakeCheck(ca_fields, stmt),
        Time($now)
    ]->
    [ ]
```

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

```
test root_ca_rogue_monitorExt:
    "∃ stmt idCA ca_fields #t0 #t1.
        CAFakesCert(idCA, stmt, ca_fields)@t0
            /* stmt contradict known ground truth
↳ of monitor */
        ∧ UndocumentedIntermediateCA(rootCA, idCA,
↳  ca_fields)@t1
    "

test ca_rogue_monitorExt:
    "∃ rootCA stmt ca_fields #t0 #t1.
        CAFakesCert(idCA, stmt, ca_fields)@t0
            /* stmt contradicts known ground truth
↳  of monitor */
        ∧ DocumentedIntermediateCA(rootCA,
↳ ca_fields, idCA)@t1
    "
```

Listing 8. CT also has acc. auth. assuming external validation.

```
lemma A_CTExt_1 [heuristic=O "acc_oracle"]:
    root_ca_rogue_monitorExt, ca_rogue_monitorExt
↳ accounts for
    "∀ rootCA stmt ca_fields #t0 #t1.
        MonExtFakeCheck(ca_fields, stmt)@t0
            /* Monitor receives the stmt of the
↳ certificate */
        ∧ MonitorChecksCCADB(rootCA, ca_fields)@t1
        ⇒
            (∃ idCA #t. GroundTruth(idCA, stmt)@t
↳ ∧ #t < #t0)
    "
```

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.

4.12. 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
        ca_fields = <$rootCA, $idCA, pkCA, 'pubKey
↳ -CA', snCA, $i0, $j0>
        cert_fields = <$idCA, $subj, ke, 'pre-cert
↳ ', sn, $i, $j>
        chain = <<cert_fields, cert_sig>, <
↳ ca_fields, ca_sig>>
        stmt = <$idCA, $subj, ke, sn>
    in
    [ !LoggerDict($idM, $idL), !TrackSubject($idM,
↳  su), In(cert_fields), In(cert_sig), In(
↳ ca_fields), In(ca_sig), !TLS_L_to_M($idL, $idM,
↳  $snapshot, commit, session), !Root_CA($rootCA,
↳  pkRootCA) ]
    -[
        Eq(verify(cert_sig, cert_fields, pkCA),
↳ true()),
        Eq(verify(ca_sig, ca_fields, pkRootCA),
↳ true()),
        MonitorFindsCert($idM, $idL, stmt, commit,
↳  session),
        MonitorBlamesCA_CT($idCA, $idL, stmt,
↳ ca_fields, commit, session),
        MonitorObservesEntry($idM, $idL, chain,
↳ commit, session), /* see restriction */
        RogueCert($idCA, stmt), /* see restriction
↳  */
        MonitorInspectsLog($idM, $idL, stmt,
↳ ca_fields, commit, session), /* monitor
↳ explores log entries */
        MonitorInspectsLog_noTrust($idL, stmt,
↳ commit, session), /* monitor explores log
↳ entries + allowed to sleep */

        Revoked(stmt), /* [in Revocation model]
↳ Monitor initiates revocation */
        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_auth_on_log:
    "∃ idL stmt idL1 idL2 idM ca_fields idCA
↳ commit session #t0 #t1 #t2 #t3.
        ClientAcceptsChain(idL1, idL2, rootCA,
↳ ca_fields, idCA, stmt)@t0
        ∧ MonitorFetchesLogSnapshot(idM, idL,
↳ session)@t1
        ∧ MonitorBlamesCA_CT(idCA, idL, stmt,
↳ ca_fields, commit, session)@t2
        ∧ UndocumentedIntermediateCA(rootCA, idCA,
↳  ca_fields)@t3 /* rootCA blamed */
        ∧ ((idL = idL1) ∨ (idL = idL2))
        ∧ ProofRunsAfterMMD(idL, commit, session)
        ∧ #t0 < #t1
    "

test root_ca_auth_on_log:
    "∃ idL stmt idL1 idL2 idM ca_fields rootCA
↳ commit session #t0 #t1 #t2 #t3.
        ClientAcceptsChain(idL1, idL2, rootCA,
↳ ca_fields, idCA, stmt)@t0
        ∧ MonitorFetchesLogSnapshot(idM, idL,
↳ session)@t1
        ∧ MonitorBlamesCA_CT(idCA, idL, stmt,
↳ ca_fields, commit, session)@t2
```

```
        ∧ DocumentedIntermediateCA(rootCA,
↳ ca_fields, idCA)@t3
        ∧ ((idL = idL1) ∨ (idL = idL2))
        ∧ ProofRunsAfterMMD(idL, commit, session)
        ∧ #t0 < #t1
    "
```

Listing 9. Refuted lemma: CT has **no** acc. auth. even if the monitor is trusted.

```
lemma A_CT_1 [heuristic=O "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
        ∧ MonitorFetchesLogSnapshot(idM, idL,
↳ session)@t1
        ∧ MonitorInspectsLog(idM, idL, stmt,
↳ ca_fields, commit, session)@t2
        ∧ MonitorChecksCCADB(rootCA, ca_fields)@t3
        ∧ ((idL = idL1) ∨ (idL = idL2))
        ∧ ProofRunsAfterMMD(idL, commit, session)
        ∧ #t0 < #t1
            ⇒ (∃ #t. GroundTruth(idCA, stmt)@t)"
```

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_honest_log:
    "∃ idM idL1 idL2 idL rootCA ca_fields stmt
↳ commit session #t0 #t1 #t2 #t3.
        ClientAcceptsChain(idL1, idL2, rootCA,
↳ ca_fields, idCA, stmt)@t0
        ∧ MonitorFetchesLogSnapshot(idM, idL,
↳ session)@t1
        /* Monitor fetches from the log after a
↳ client worked with the certificate (it should
↳ be existing by now, acceptable assumption since
↳  we do not use this to blame the logger) */
        ∧ MonitorBlamesCA_CT(idCA, idL, stmt,
↳ ca_fields, commit, session)@t2 /* idCA free */
        ∧ DocumentedIntermediateCA(rootCA,
↳ ca_fields, idCA)@t3
        ∧ ProofRunsAfterMMD(idL, commit, session)
        ∧ ((idL = idL1) ∨ (idL = idL2))
        ∧ #t0 < #t1
        ∧ not (∃ #t. Corrupted(idL)@t) /*
↳ considered logger is honest */
    "

"∃ idM idL1 idL2 idL idCA ca_fields stmt commit
↳ session #t0 #t1 #t2 #t3.
        ClientAcceptsChain(idL1, idL2, rootCA,
↳ ca_fields, idCA, stmt)@t0
        ∧ MonitorFetchesLogSnapshot(idM, idL,
↳ session)@t1
        ∧ MonitorBlamesCA_CT(idCA, idL, stmt,
↳ ca_fields,commit, session)@t2
        ∧ UndocumentedIntermediateCA(rootCA, idCA,
↳  ca_fields)@t3
        ∧ ProofRunsAfterMMD(idL, commit, session)
        ∧ ((idL = idL1) ∨ (idL = idL2))
        ∧ #t0 < #t1
        ∧ not (∃ #t. Corrupted(idL)@t) /*
↳ considered logger is honest */
    "
```

Listing 10. CT has acc. auth. if logger is honest.

```
lemma A_CT_2 [heuristic=O "acc_oracle"]:
    ca_rogue_honest_log, root_ca_rogue_honest_log
↳ accounts for
    "∀ idM idL1 idL2 idL rootCA ca_fields idCA
↳ stmt commit session #t0 #t1 #t2 #t3.
        ClientAcceptsChain(idL1, idL2, rootCA,
↳ ca_fields, idCA, stmt)@t0
        ∧ MonitorFetchesLogSnapshot(idM, idL,
↳ session)@t1
        ∧ MonitorInspectsLog(idM, idL, stmt,
↳ ca_fields, commit, session)@t2
        ∧ MonitorChecksCCADB(rootCA, ca_fields)@t3
        ∧ ProofRunsAfterMMD(idL, commit, session)
        ∧ ((idL = idL1) ∨ (idL = idL2))
        ∧ #t0 < #t1
        ∧ not (∃ #t. Corrupted(idL)@t)
            ⇒
                (∃ #i. GroundTruth(idCA, stmt)@i)
    "
```

5. 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.

5.2. Accountable Authenticity without an honest monitor. We revisit our accountable authenticity property from Section 6 and show that it does not hold without assuming an honest monitor. The case tests in this lemma are reused from Lst 2.

Listing 11. Refuted Lemma: Accountable Authenticity without an honest monitor is not achieved.

```
lemma A_CTAudit_3:
    ca_rogue_audit, root_ca_rogue_audit accounts
↳ for
    "∀ ca_fields stmt rootCA idL #t0 #t1.
        AuditAuthCheck_noTrust(idL, ca_fields,
↳ stmt)@t0
            /* cert with stmt is audited to
↳ monitor */
        ∧ MonitorChecksCCADB(rootCA, ca_fields)@t1
            ⇒
                (∃ idCA #t. GroundTruth(idCA, stmt
↳ )@t)"
```

5.3. 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.

Listing 12. SCT auditing provides transparency.

```
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.

6. CT with Receipts (Details)

So far we excluded monitor misbehavior from our accountability analysis or were able to blame them with unrealistic tests as seen in Section H. CT does not provide us with a mechanism suited for accountability tests, as a party distinct from the monitor cannot verify whether the monitor purposefully ignored a rogue certificate, let alone whether a monitor even saw the certificate in the first place.

To address this limitation, we further extend our model of SCT Auditing with *receipts* issued by the monitor back to the logger. The receipt proves to a logger (and any other party in posession of them) that the monitor has seen a specific snapshot of the log. SCT Auditing is in particular of interest here, as we were able to show that accountable authenticity can be achieved in SCT Auditing given honest monitors, but allowed corrupted loggers. With this receipt approach we aim to achieve accountable authenticity even in the presence of corrupted monitors.

We adapt our model by giving a key pair to every monitor and include the signing process of a signature.

```
rule signReceipt:
    let
        receipt = sign(<$idM, chain>, skM)
    in
    [ CraftReceipt($idM, chain) , !MonitorKey_s(
↳ $idM, pkSKM, skM) ]
```

```
    −[ ]→
    [ !TLS_M_to_C(receipt) ]
```

`CraftReceipt` is a fact emitted from the SCT Auditing rules where a monitor receives an audited certificate chain from a client. The receipt is a simple signature over that received chain and the monitor's identifier. The signature is then sent back to the client via a TLS channel, modeled by the `TLS_M_to_C` fact.

The client then, if necessary, is able to forward the receipt to the domain owner.

```
rule domainOwnerBlamesMonitor:
    let
        stmt = <$idCA, $idServer, key, sn>
    in
    [ In(<$idM, $idL, $snapshot, commit, session,
↳ receipt>), !MonitorKey($idM, pkSKM), !
↳ TrackSubject_Contract($idM, $idServer), In(stmt
↳ ) ]
    −[
        Eq(verify(receipt, <$idM, $idL, $snapshot,
↳  commit, session>, pkSKM), true()),
        Time($now),
        DO_MonitorBlame($idM, $idCA, stmt, commit,
↳  session),
        DomainOwnerChecksReceipt($idM, stmt,
↳ commit, session),
        InclusionProofSucceeds($idL, stmt, commit,
↳  session),
        RogueCert($idCA, stmt)
    ]→
    [ ]
```

In general we assumed in SCT Auditing that every received certificate is checked. To allow corrupted monitors to diverge from this behavior, we allow them to *sleep* on an entry, i.e., seeing the entry but not acting on it. In this case, the `DO_MonitorBlame` can be reached.

Further rules that are left out here consider the variants where we cannot blame a monitor. This is the case if either the monitor acted benignly when confronted with the rogue certificate, i.e., raised an alert to the domain owner, or if the monitor did not need to react, as the certificate was benign.

6.1. Accountability Analysis. In this scenario we assume that at some point a rogue certificate that is actively used is found by the domain owner. The domain owner has access to the ground truth so accountable authenticity via an external check is easily achieved, as seen in section 4. But additionally the domain owner contracted a monitor service to do exact that, but there was no alert raised. We show that the CT Receipt model can be utilized to design a test that also considers monitor misbehavior and combined with previous tests components concerning the intermediate or root CA misbehavior provides us with accountable authenticity also in the case where the monitor is corrupted. The tests are provided below.

```
test root_ca_rogue:
    "∃ idM stmt idCA CA_fields #t0 #t1 #t2.
        AssertsNot(idCA, stmt, CA_fields)@t0
        ∧ UndocumentedIntermediateCA(rootCA, idCA,
↳  CA_fields)@t1
        ∧ DO_MonitorBenign(idM, CA_fields, stmt)
↳ @t2
    "

test root_ca_rogue_monitor_rogue:
    "∃ stmt idCA CA_fields #t0 #t1 #t2.
        AssertsNot(idCA, stmt, CA_fields)@t0
        ∧ UndocumentedIntermediateCA(rootCA, idCA,
↳  CA_fields)@t1
        ∧ DO_MonitorBlame(idM, idCA, CA_fields,
↳ stmt)@t2
        ∧ not [root_ca_rogue]
    "
```

Two more analogous tests instead blame the intermediate CA as seen before. Together they construct the following lemma, which verifies.

Listing 13. SCT Auditing with receipts provides acc. auth.

```
lemma A_Receipt_1:
    root_ca_rogue,
    root_ca_rogue_monitor_rogue,
    ca_rogue_receipt,
    ca_rogue_receipt_monitor_rogue
    accounts for
    "∀ CA_fields stmt rootCA idL idM #t0 #t1 #t2 #
↳ t3.
        AuditAuthCheck_noTrust(idL, CA_fields,
↳ stmt)@t0
        ∧ MonitorChecksCCADB(rootCA, CA_fields)@t1
        ∧ DomainOwnerChecksReceipt(idM, CA_fields,
↳  stmt)@t2
        ∧ StatementCheckedExternally(CA_fields,
↳ stmt)@t3
        /* all later checks performed by the
↳ domain owner */
            ⇒
                (∃ idCA #t. GroundTruth(idCA, stmt
↳ )@t)"
```

6.2. Discussion. Using receipts issued by monitors to loggers, we were able to design tests that allow us to blame corrupted monitors that purposefully ignored rogue certificates. Together with previous tests blaming corrupted CAs, we achieve accountable authenticity even in the presence of corrupted monitors. Receipts illustrate a possible mechanism to be able to hold monitors accountable for their actions. We moved multiple tests to the domain owner here, which is a natural choice as the domain owner has access to the ground truth and is particularly interested in correct behavior of the contracted monitor. We relied on the fact that the domain owner becomes access to such an receipt, e.g., by the client forwarding it. It is not straightforward how to ensure that the domain owner actually receives such receipts in practice. While domain owners run their own web server, i.e., they are addressable, sending the receipt to the *correct* web server boils down to the authenticity problem we started with. Additionally, we assumed that the monitor issues such receipts for every received certificate. While a corrupted monitor could choose to not issue a receipt, note that a client could always demand a receipt or otherwise refuse to accept the certificate to protect themselves.

With an increasing complexity on the domain owner side, one could raise the argument that a monitor distinct from the domain owner is not necessary anymore. Recall that everyone can become a monitor and self-run monitors

have the advantage that the running party immediately trusts this monitor. However, SCT Auditing assumes the presence of a powerful monitor that gets access to all certificates that clients eventually see. We already discussed in section 9 that self-run monitors with these capabilities are not realistic. A mechanism to hold monitors accountable is in particular of interest if domain owners have to rely on such a powerful third-party monitor.

7. Modelling SCT Gossiping (Details)

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

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 use similar restrictions to axiomatically assert append-only proofs (Section D.8), but this time the gossiped proof is not necessarily handed out to the monitor but to any party to reflect the nature of gossiping.

```
rule monitorFindsAppendOnlyViolation_Gossiped:
    [
        !LoggerDict($idM, $idL),
        !TLS_C_to_M($idC, $idM, $gos_snapshot,
↳ gossiped_sth, gossiped_session),
        !TLS_L_to_M($idL, $idM, $snapshot, sth,
↳ session) ]
    -[
        Gos_AppendOnlyViol($idM, $idL, sth,
↳ session, gossiped_sth, gossiped_session),
        Gos_AppendOnlyCheck($idM, $idL, sth,
↳ session, gossiped_sth, gossiped_session),
        Time($now)
    ]->
    [ ]
```

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

```
restriction appendOnlyFails_Gossiped:
   "∀ idM IdL session sth gossiped_sth
↳ gossiped_session #t5.
    Gos_AppendOnlyViol(idM, IdL, sth, session,
↳ gossiped_sth, gossiped_session)@t5
      ⇒
       (∃ id1 preC_fields chain time t time2 log_id
↳  log_id2 #t0 #t1 #t2 #t3 #t4.
        LoggerComputesAddLog(IdL, log_id,
↳ preC_fields, chain, t)@t0
        ∧ EntryVisible(IdL, preC_fields, log_id, #
↳ t2, time) /* There is an entry that was visible
↳ ... */
        ∧ FetchLogSnapshot(id1, IdL,
↳ gossiped_session)@t1
        ∧ LoggerPresentsView(IdL, id1, log_id,
↳ time, gossiped_sth, gossiped_session)@t2 /* ...
↳  for a given proof at #t2 */
        ∧ FetchLogSnapshot(idM, IdL, session)@t3
        ∧ LoggerPresentsView(IdL, idM, log_id2,
↳ time2, sth, session)@t4
        ∧ EntryMissing(IdL, preC_fields, log_id2,
↳ #t4, time2)
        /* one entry that was visible in the
↳ gossiped view is now not visible anymore ->
↳ clearly a violation */
        ∧ #t0 < #t1 ∧ #t1 < #t2 ∧ #t2 < #t3 ∧ #t3
↳  < #t4 ∧ #t4 < #t5
       )"
```

The other rules consider the opposite case and are analogously adapted.

7.1. Transparency.

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

Listing 14. Refuted lemma: STH gossiping does not provide transparency.

```
 "∀ idL1 idL2 preCert idL id log_id time idM
↳ gos_sth gos_session sth session #t0 #t1 #t2 #t3
↳  #t4.
        ClientAcceptsCert(idL1, idL2, preCert)@t0
        ∧ ((idL = idL1) ∨ (idL = idL2))
        ∧ LoggerPresentsView(idL, id, log_id, time
↳ , gos_sth, gos_session)@t1
            /* view is after the SCT has been used
↳  in a validation */
        ∧ LoggerPresentsView(idL, idM, log_id,
↳ time, sth, session)@t2
        ∧ #t0 < #t1 ∧ #t1 < #t2
        ∧ MonitorChecksInclusion(idM, idL, preCert
↳ , sth, session)@t3
        ∧ AppendOnlyVerified_Gossiped(idM, idL,
↳ sth, session, gos_sth, gos_session)@t4
            /* monitor uses gossiped knowledge and
↳  verifies append-only */
              ⇒
                InclusionProofSucceeds(idL,
↳ preCert, sth, session)@t3"
                    /* 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, but the lemmas is still falsified.

Listing 15. Refuted lemma: STH gossiping does not provide transparency, even with client inclusion proofs.

```
lemma no_transparency_gossip_with_client_proof:
    "∀ idL1 idL2 preCert idL id log_id time idM
↳ gos_sth gos_session sth session #t0 #t1 #t2 #t3
↳  #t4 #t5.
        ClientAcceptsCert(idL1, idL2, preCert)@t0
        ∧ ((idL = idL1) ∨ (idL = idL2))
        ∧ LoggerPresentsView(idL, id, log_id, time
↳ , gos_sth, gos_session)@t1
            /* view is after the SCT has been used
↳  in a validation */
        ∧ LoggerPresentsView(idL, idM, log_id,
↳ time, sth, session)@t2
        ∧ #t0 < #t1 ∧ #t1 < #t2
        ∧ MonitorChecksInclusion(idM, idL, preCert
↳ , sth, session)@t3
        ∧ AppendOnlyVerified_Gossiped(idM, idL,
↳ sth, session, gos_sth, gos_session)@t4
        ∧ InclusionProofSucceeds(idL, preCert,
↳ gos_sth, gos_session)@t5
            ⇒
```

```
            InclusionProofSucceeds(idL,
↳ preCert, sth, session)@t3"
                /* then this entry should be
↳ visible to the monitor */
```

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 is sufficient achieve accountable transparency.

7.2. 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.

```
test gossip_inclusionViol:
    "∃ [...].
        MonitorFetchesLogSnapshot(idM, idL,
↳ session)@t1 /* Monitor fetches an up-to-date
↳ snapshot from the log */
        ∧ InclusionProofFails(idM, idL, preCert,
↳ sth, session)@t2 /* On this snapshot, the
↳ inclusion proof of preCert fails */
        ∧ Gos_AppendOnlyViol(idM, idL, sth,
↳ session, gos_sth, gos_session)@t3
        /* additionally, the monitor finds that
↳ append-only between this snapshot and the
↳ gossiped_one is violated */
        ∧ InclusionProofSucceeds(idL, preCert,
↳ gos_sth, gos_session)@t4
        /* as property for the gossiped snapshot
↳ we require that the client successfully ran an
↳ inclusion proof on it (compare to last lemma
↳ that failed because of this) */
        "
```

This test is based on the findings in Section G.1, 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.

Listing 16. STH gossiping provides accountable transparency (with client inclusion proofs).

```
lemma T_Gossip [heuristic=O "acc_oracle"]:
    gossip_inclusionViol accounts for
    "∀ idM idL preCert sth session gos_sth
↳ gos_session #t1 #t2 #t3 #t4.
        MonitorFetchesLogSnapshot(idM, idL,
↳ session)@t1
        ∧ InclusionProof(idM, idL, preCert, sth,
↳ session)@t2
            /* monitor runs an inclusion proof for
↳  preCert (as before with transparency) */
        ∧ Gos_AppendOnlyCheck(idM, idL, sth,
↳ session, gos_sth, gos_session)@t3
            /* monitor checks append-only between
↳ both snapshots */
        ∧ InclusionProofSucceeds(idL, preCert,
↳ gos_sth, gos_session)@t4
            /* client proofs inclusion of stmt in
↳ gos_sth */
            /* append only between gos_sth and sth
↳  is checked */
            ⇒
            InclusionProofSucceeds(idL, preCert,
↳ sth, session)@t2"
            /* then this inclusion proof succeeds
↳ */
```

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

8. Accountability with naïve 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.

8.1. 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.

8.2. 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 time0 time1 commit
↳ session log_id1 log_id2 #j #k #l.
                    LoggerComputesAddLogStmt(IdL,
↳ log_id1, stmt, time0)@j
```

```
                    /* SCT exists already */
                  ∧ LoggerPresentsView(IdL, IdM,
↳  log_id2, time1, commit, session)@k
                    /* Monitor receives a view
↳  from the log after it has been added. Here the
↳  logger may be dishonest and is able to use a
↳ different view */
                  ∧ not (time0 = time1)
                    /* only considered after
↳ MMD */
                  ∧ 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
        ∧ DocumentedIntermediateCA(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
        ∧ MonitorChecksCCADB(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
        ∧ DocumentedIntermediateCA(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
        ∧ MonitorChecksCCADB(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
        ∧ DocumentedIntermediateCA(rootCA,
↳ ca_fields, idCA)@t2
        ∧ MonitorLiveness(stmt, #t1)
    "


lemma A_Revoke_3:
    monitor_sleeps_unconditional_intm_CA,
    monitor_sleeps_unconditional_root_CA accounts
↳ for
    "∀ idL1 idL2 rootCA CA_fields idCA stmt #t1 #
↳ t2.
        MonitorLiveness(stmt, #t1)
        ∧ ClientAcceptsChain(idL1, idL2, rootCA,
↳ ca_fields, idCA, stmt)@t1
        ∧ MonitorChecksCCADB(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 (Lst 12). 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
        ∧ DocumentedIntermediateCA(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
        ∧ MonitorChecksCCADB(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 Section F 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.