Provable Security for PKI Schemes

To appear in ACM SIGSAC CCS 2024

Sara Wrótniak
University of Connecticut
Storrs, CT

Hemi Leibowitz*
The College of Management Academic Studies
Rishon Lezion, Israel

Ewa Syta†
Trinity College
Hartford, CT

Amir Herzberg
University of Connecticut
Storrs, CT

ABSTRACT

PKI schemes provide a critical foundation for applied cryptographic protocols. However, there are no rigorous security specifications for realistic PKI schemes, and therefore, no PKI schemes were proven secure. Cryptographic systems that use PKI are analyzed by adopting overly simplified models of the PKI, often, simply assuming securely-distributed public keys. This is problematic given the extensive reliance on PKI, the multiple failures of PKI systems, and the complexity of both proposed and deployed systems, which involve complex requirements and models.

We present game-based security specifications for PKI schemes, and analyze important and widely deployed PKIs: PKIX and two variants of Certificate Transparency (CT). All PKIs are based on the X.509v3 standard and its CRL revocation mechanism. Our analysis identified few subtle vulnerabilities, and provides reduction-based proofs showing that the PKIs ensure specific requirements under specific models (assumptions). To our knowledge, this is the first reduction-based proof of security for a realistic PKI scheme, e.g., supporting certificate chains.

KEYWORDS

PKI, provable-security

1 INTRODUCTION

Public Key Infrastructure (PKI) provides an essential foundation for applications that rely on public key cryptography, which is crucial to ensure security in open networks and systems. Early PKI ideas were proposed in 1978 [29], and the first version of the X.509 standard [10] was published in 1988. Since then, the deployment of PKI has been dominated by X.509, specifically, by the IETF PKIX standard, which adopts version 3 of X.509 (X.509v3) for Internet protocols, most notably, TLS/SSL [49]. *Certificate Transparency (CT)* [47, 32] is a recent, widely-deployed extension to PKIX, motivated by multiple PKI failures, mainly, rogue certificates issued by corrupt or negligent Certificate Authorities (CAs). A significant number of other PKI schemes were proposed recently, with different goals and properties, and different, non-trivial designs, including [52, 15, 38, 28, 57, 4, 61, 55, 56, 36, 17, 3, 58, 30].

Considering the importance, variety and complexity of (some) PKI schemes, it is essential to ensure their security. Currently, there are no rigorous security specifications for realistic PKI schemes, and therefore, no PKI schemes were proven secure. This situation stands in sharp contrast to the accepted norms in (applied and theoretical) modern cryptography, which require well-defined security requirements and reduction-based proofs of security. These norms began in the 1980s with the seminal papers defining secure encryption [19] and secure signature schemes [20]. We present the first complete[1] definitions and analysis for (certificate-based) PKI schemes and their security.

The lack of rigorous specifications and analysis for PKI schemes is especially alarming, since PKI provides a critical infrastructure to applied cryptography, i.e., security of many applied cryptographic systems depend on the security of the underlying PKI. Extensive efforts to prove cryptographic protocols may be moot when these protocols depend on an insecure PKI scheme. The concerns are even greater, considering that attacks against PKI are not only a theoretical threat, but also major concerns in practice [54, 14].

Rigorous security specifications are relevant to practical, emerging developments and applications of PKIs, too. The updated European Union eIDAS (Electronic Identification, Authentication, and Trust Services) regulation, known as eIDAS 2.0 [12], which is expected to be approved in 2024 with a 24-month implementation period for member states, aims to establish a comprehensive digital identity framework across the EU. This framework will heavily rely on the existing Web PKI. Surprisingly and alarmingly, as noted in [40, 24], Article 45 of eIDAS 2.0 sets out very specific technical requirements and constraints for the existing Web PKI. It essentially mandates that web browsers accept and trust CAs controlled by EU member states, does not allow for their removal even if deemed unsafe or malicious, and prohibits the enforcement of security mechanisms beyond what is approved by ETSI [1], potentially limiting important security enhancements such as CT. It is concerning that a large-scale, mandatory digital identity ecosystem is being developed around a PKI system that lacks formal specifications and proofs of security, yet is known for its many failures and pitfalls.

Further, rigorous definitions and analysis often allow the identification of subtle yet significant issues that otherwise may go unnoticed. We identified a few of these. First, the CRL design does not achieve the intuitively expected, guaranteed notion of revocation, but only a weaker notion, *Accountable-Revocation*. Second, in a similar fashion, CT does not achieve *Guaranteed-Transparency* and only ensures a weaker version of transparency, *HL-Transparency*.

*The work was partially completed during the author's PhD studies at the Dept. of Computer Science, Bar-Ilan University, Israel

†The work was partially completed during a visiting position at the Dept. of Computer Science and Engineering, University of Connecticut, Storrs, CT

[1] Basic PKI schemes, without even revocation and certificate chains, were presented in [9, 5, 18]; the long version of [5] describes how revocation could be added.

This assumes that a certificate has been logged by at least one honest logger. Third, the non-standard CTwAudit extension ([27]) avoids this assumption, but, again, only ensures a weaker notion, *Audited-Transparency*. Finally, as also noted in [34], in CT, to ensure awareness of issued certificates (transparency), we need one or more honest monitors that (together) cover *all* the logs that relying parties trust in validating transparency of certificates.

To address these concerns, we present the *first rigorous (game-based) definitions of security requirements for PKI schemes*, and then the *first reduction-based proofs of security* for practical PKI schemes. Table 1 summarizes our results.

Defining and proving security for PKI schemes is challenging, especially for post-X.509 schemes, whose requirements (goals) and designs are more advanced and complex. Different PKI schemes may assume different communication, synchronization and adversary models; even the set of entities may differ, e.g., CT also introduces *loggers* and *monitors* in addition to *Certificate Authorities (CAs)*. As a result, existing works often use informal security requirements and models for PKI schemes.

To illustrate the challenge of properly defining goals of PKI systems, consider *accountability*, a basic goal of PKI systems with surprising subtleties. Intuitively, accountability is the ability to *identify* the CA that is *responsible* for the issuing of an unauthorized certificate ψ. However, *who is the responsible CA* for ψ? Instinctively, we may expect this to be the *issuer* of ψ, i.e., in X.509 certificates, the CA identified in the issuer field of ψ, denoted as $\psi.issuer$. However, surely $\psi.issuer$ should be held accountable for ψ only if it actually issued (signed) ψ. On the contrary, $\psi.issuer$ should not be held accountable if the public verification key pk used to validate ψ is *not* a correct public-key of $\psi.issuer$. Also, obviously, $\psi.issuer$ can only be considered accountable if it is a real, supposedly trustworthy CA. For example, consider a scenario where a rogue CA issues a certificate ψ' which fraudulently specifies pk as a public verification key of $\psi.issuer$; this rogue CA should be held accountable, rather than the (benign or even non-existing) $\psi.issuer$.

Revocation is another basic goal of PKI systems which is not trivial to define. In fact, we found it necessary to define two variants of revocation: Guaranteed and Accountable. The Guaranteed variant is more intuitive; basically, a PKI scheme ensures *Guaranteed Δ-Revocation* if a certificate ψ revoked by a benign CA at some time t, will not be considered as valid by any benign party after time $t + \Delta$. However, during our analysis, we realized that PKIX does not ensure Guaranteed Δ-Revocation (Claim 1). Instead, PKIX ensures a weaker notion which we call *Accountable Δ-Revocation* (§3.4). Intuitively, Accountable Δ-Revocation means that if ψ was revoked at time t by its benign issuing CA yet considered valid after $t + \Delta$, then we can identify, and hold accountable, a rogue CA responsible for this failure to revoke ψ.

Transparency is another important PKI goal, underlying *Certificate Transparency (CT)* [31, 50]. Intuitively, transparency aims to ensure that certificates are available for scrutiny, in the form of a *public log* where certificates must appear within a specified time-frame after being issued. Transparency allows detection of rogue certificates (and applying accountability), before their use by attackers, and does not rely on victims detecting their use. CT was motivated by the detection of the issuance of over 530 fraudulent certificates by DigiNotar CA [60, 31]; the detection was only a month after the breach of the DigiNotar private key. This incident motivated the creation of public logs of certificates to enable swift detection of rogue certificates, aiming to avoid reliance on a set of trusted third parties [31].

However, our analysis shows that the CT specification [31, 50] ensures only a weak notion of transparency (HL-Transparency), which requires that a certificate has been logged by at least one honest logger. This is not due to a vulnerability of the cryptographic mechanisms used by CT, which were shown to be secure by Dowling et al. [13] and Chase and Meiklejohn [11]. While the underlying CT cryptography is sound, its deployment in CT is secure only if at least one logger is honest; e.g., as noted in [13, 11], a rogue logger can simply ignore some requests. The CT specifications[2] [47, 50] also do not specify or require gossip and audit, implicitly assumed by [13, 11].

We have also analyzed CTwAudit, which is a variant of CT, supported as an option by Google's Chrome, where the browser performs auditing of the loggers [27]. We show that CTwAudit ensures another weak variant of the transparency requirement; we call this variant Audited-Transparency. Note that the CTwAudit design raises privacy concerns: it may expose a user's browsing history through the audited certificates.

We use the *Modular Security Specifications (MoSS) framework* [21] to define PKI requirements and adversary, synchronization, network and other models assumed by different PKIs. The use of MoSS provides essential modularity; different PKIs support different requirements and assume different models. For example, we show that CT ensures accountability and Accountable Δ-Revocation under one model, and transparency only under a stronger model (assuming reliable communication with bounded delays).

We present pseudocode to rigorously define and analyze (minimally simplified[3]) the most well-known and widely-deployed PKI schemes: PKIX (X.509 version 3 with CRL as defined in [42]), CT [47, 50] and CTwAudit [27]. Table 1 summarizes the results of our analysis of these schemes.

Out of scope. We do not address how relying parties select their *trust anchors*, i.e., the identities of the 'root CAs'; and we mostly ignore constraints on the allowed certificate-chains, such as the *name, length* and *policy constraints* (defined in X.509v3 and PKIX). A model of such *trust decisions* for PKI systems was proposed by Maurer [37], extended by [35, 7], and others [22, 33, 62, 53, 6, 26]. These constraints and solutions are complementary and orthogonal to our results; additional constraints can only prevent an adversary from 'breaking' the PKI.

Contributions. The contributions of this work are as follows.

(1) Presents the first definition of a (non-trivial) PKI scheme, e.g., supporting certificate chains.

[2]Version 1 of the CT specifications [47] incorrectly states that the public logs can be *untrusted*.

[3]We included every aspect which appeared to possibly impact the requirements. Our most significant simplification is that we focus on the standard CRL revocation mechanism, while the specifications also allow OCSP and proprietary revocation mechanisms. We also simplify by assuming that each logger keeps only one log, that the public key used for the log is the same as the logger's public key, and that loggers' self-certified keys and issued SCTs do not have a validity period, i.e., never expire. We expect that automation will be necessary to extend our analysis to cover all aspects of the specifications.

Table 1: PKI requirements defined in this work, and properties we prove for prominent PKI systems.

	Requirements				
PKI scheme	**Unforgeability (Algorithm 2)**	**Accountability (Algorithm 3)**	**Δ-Revocation (Definition 3 and Algorithm 4)**		**Δ-Transparency (Definition 4 and Algorithm 5)**
			Accountable	**Guaranteed**	
PKIX (X.509 version 3 with CRL)	✓ (Theorem 3)		✗ (Claim 1)		✗ (n/a)
CT					HL-Transparency (Theorem 4)
CTwAudit					Audited-Transparency (Theorem 6)

(2) Presents the first rigorous security requirements for PKI schemes, including the *unforgeability*, *accountability*, *revocation*, and *transparency* requirements.
(3) Presents the first analysis and proofs of security for the most widely-deployed PKI standards, PKIX, CT and CTwAudit.
(4) Identifies subtle aspects of these PKIs, especially their failure to meet the stronger (and simpler, natural) revocation and transparency requirements; defines weaker notions and proves they are achieved.
(5) Introduces and constructs a *certificate scheme* (Section 4.1), an abstraction for applying signatures to structured information. Certificate schemes simplify definition and analysis of PKI schemes, and may have additional applications.

Organization. The paper is organized as follows. We define *PKI* schemes in §2 and define security requirements for PKI in §3. We present the specifications of PKIX in §4 and CT in Appendix D, and we analyze their security in §5 and Appendix E. Finally, we conclude and discuss future work in §6.

2 PKI SCHEMES

PKI schemes define how to issue, manage and use *certificates*. Usually, e.g., in X.509, a certificate is a signed object, containing some *certified information*. In §2.1, we describe common certificate fields and different certificate types. In §2.2, we discuss *basic PKI entities*. Then, we discuss basic *PKI functions*: certifying (issuing) and revoking certificates (§2.3), and evaluating the *validity* of a certificate (§2.4). Finally, in §2.5, we define *PKI schemes* and *transparent PKI schemes*, which are PKI schemes with additional entities and functions used to ensure transparency.

2.1 Certificate Fields and Types

A *certificate* is a string which encodes the value of multiple fields, as well as a signature, or other cryptographic mechanism, to validate the authenticity of the certificate. Different PKIs may certify different information (fields), use different encodings or different signature algorithms, and, in principle, may even use a different design (i.e., not a signature over an object). For example, in X.509 and PKIs based on it, certain certified information is encoded as a 'field', while other information is encoded as an 'extension'. That said, all deployed PKI schemes have similar designs.

PKI standards, including X.509, PKIX and CT, define and refer to specific named values in certificates. We refer to all of these named values as 'fields'. We list in Table 2 some of the important fields[4].

We use the *dot notation* to refer to a specific field in a certificate, e.g., $\psi.issuer$ refers to the value of the *issuer* field in certificate ψ. The exceptions are the *type* and *PKIadded* fields; they are not defined in X.509 or other existing PKIs, yet we found them important.

The *type* field is used to distinguish between *different types of certificates*. X.509 defines two types of certificates: *public key certificates* and *attribute certificates*, which we identify by $\psi.type = \text{'PubKey'}$ and $\psi.type = \text{'ATTR'}$, respectively. Public key certificates associate a public key with a particular subject ('owner' of the public key), whereas *attribute certificates* do not contain a public key (i.e., if $\psi.type = \text{'ATTR'}$, then $\psi.pk = \perp$). Further, the *type* field can mark other types of certificates issued by the PKI, e.g., *pre-certificates*, which can be submitted to loggers in CT, or *Route Origin Authorizations (ROAs)*, defined in the Resource Public Key Infrastructure (RPKI) [46], used to specify allowed announcements for IP prefixes. In the case of ROAs, the *issuer* certifying a ROA would have a public-key certificate proving its ownership of relevant IP prefix. Certificates can have additional fields indicating specific constraints of the certificates.

PKIs may add fields, such as the 'issuer' field in PKIX or the 'SCT-set' field in CT, to the submitted fields before creating a certificate. We use the $tbc.PKIadded$ field to identify such fields. We use the notation $\psi.tbc$ to refer to the entire set of (name, value) fields in certificate ψ.

2.2 PKI Entities

A PKI scheme $\mathcal{P}$ is defined by a set of functions, some stateful and some stateless, and a set N of stateful entities. Entities in N can perform the stateful functions, e.g., issue certificates. We refer to the entities in N that issue public key certificates as *certificate authorities (CAs)*. There could be other entities in N, e.g., Certificate Transparency (CT) uses *loggers* and *monitors*. Entities in N may be *honest (benign)* or *corrupt*, i.e., controlled by an adversary. *Relying parties* are entities which use only stateless PKI functions, in particular, the certificate validation function, which allows the relying parties to decide whether to *rely on the certificate* (i.e., use the certified public key) or not.

The state of the entities in N is initialized using a dedicated *initialization* operation $\mathcal{P}$.Init. Typically, $\mathcal{P}$.Init outputs a *self-certified public key certificate*, i.e., a certificate ψ which certifies a key for its issuer ($\psi.issuer = \psi.subject$), and is validated (successfully) using the certified public key $\psi.pk$. Self-certified public key certificates can be output by any entity: a CA, logger or a monitor. Typically,

[4]We use the term 'field' for all of these named values, and have abstracted away the encoding. However, PKI specifications often use other terms to refer to these 'fields', and may use different encodings for different fields. For example, PKIX uses the terms dnsName *component* (in the *subjectAltName* extension) and *cA Boolean* (in the BasicConstraints extension).

Table 2: Common certificate fields.

Field	Description	Encoding in X.509
$\psi.issuer$	The entity that issued the certificate.	*distinguished name* in the *Issuer* certificate field. X.509 certificates may also include an *Issuer Alternative Name (IAN)* extension that may include additional identifiers for the issuer, e.g., as a DNS name.
$\psi.from$	The date and time at which the certificate becomes valid.	*notBefore* entry of the *Validity* certificate field.
$\psi.to$	The date and time at which the certificate should expire (become invalid).	*notAfter* entry of the *Validity* certificate field.
$\psi.serial$	A number that uniquely identifies the certificate.	*serialNumber* certificate field. In X.509, it must be positive and unique among the rest of the certificates issued by the issuing CA.
$\psi.subject$	An identifier for the subject of ψ.	*distinguished name* in the *Subject* certificate field. X.509 certificates may also include a *Subject Alternative Name (SAN)* extension, that may include additional identifiers for the subject, e.g., as a DNS name.
$\psi.pk$	A public key certified as the public key of the subject.	Part of the *Subject Public Key Info* certificate-field.
$\psi.is\_CA$	Whether the subject is a CA, i.e., authorized to issue public key certificates.	Part of the *Basic Constraints* extension.
$\psi.type, \psi.PKIadded$	The specific type of certificate, e.g., public key, and a set of fields added to the certificate by the PKI.	These fields are not defined in X.509; see discussion in §2.1.
$\psi.tbc$	The set of all (name, value) fields in ψ	This is a notation, not a field.

to validate a certificate ψ, we use a set $store.CAs$ of *trusted root certificates* which are self-certified by trusted CAs.

2.3 Certifying and Revoking Certificates

A certificate ψ is issued using the private certification key of a CA, which the CA maintains as part of its state st. To issue, the CA uses the PKI's $\mathcal{P}$.Certify operation, namely, $\mathcal{P}.\mathsf{Certify}(\mathsf{st}, clk, tbc) \rightarrow (\mathsf{st}, \psi/\perp)$, which takes as input the entity's local state st, local clock clk and the set of (name, value) fields to be certified tbc, and outputs an updated state st, and, if successful, a signed certificate ψ s.t. $\psi.tbc = tbc$.

Since certificates are typically issued for a specific time period, most PKI schemes provide a way to revoke certificates before their expiration date, for example, if a certificate is found to be fraudulently issued or the corresponding private key exposed. Revocation is done by the issuer using a dedicated revoke operation, denoted as $\mathcal{P}$.Revoke. The $\mathcal{P}$.Revoke operation takes as input a certificate ψ and outputs whether the revocation was successful or not, i.e.: $\mathcal{P}.\mathsf{Revoke}(\mathsf{st}, clk, \psi) \rightarrow (\mathsf{st}, \top/\perp)$. For example, $\mathcal{P}$.Revoke may fail if attempting to revoke an already expired or revoked certificate, or a certificate not issued by this issuer.

Most PKIs use a *non-revocation* mechanism to allow relying parties to verify that a certificate was not revoked at a given time t. X.509 defines two non-revocation mechanisms, *certificate revocation lists (CRLs)* and the *online certificate status protocol (OCSP)* allowing relying parties to verify non-revocation status of a certificate by obtaining a CRL or OCSP response valid at time t.

2.4 Certificate Validity

Each PKI has a criteria to determine whether a given certificate is valid or not. As an example of such criteria, consider a PKI where a certificate ψ is *valid* at time t, if (1) t is between $\psi.from$ and $\psi.to$, (2) ψ was certified by $\psi.issuer$. But, even such straightforward and intuitive criteria has some important subtleties.

In particular, how can we determine if ψ was *really* certified by $\psi.issuer$? In a simple setting, the validating party knows the validation key of $\psi.issuer$, typically by having the self-signed key of $\psi.issuer$ in $store.CAs$, the set of *trusted root CAs* self-certificates at the validating party. The set *store*, used to establish the trust, is an input to the validation function; *store* is often referred to as the *root store* or *trust anchor*. The sequence of certificates $\xi \equiv \psi_1 - \psi_2 - \ldots - \psi_r$ used to validate ψ, terminating in a certificate $\psi_r \in store$, is called a *certificate chain* or a *chain of trust*.

However, in a more practical setting, $\psi.issuer$ is not a trusted root CA, and therefore, will not exist in $store.CAs$. Instead, the trust in the public key of $\psi.issuer$ is established using a certificate for $\psi.issuer$, which should also be: (1) valid, and in particular, (2) signed by a trusted CA. This trusted CA "gains trust" either because its self-certified certificate appears in $store.CAs$, or because it is certified by a different trusted CA (and so on).

A PKI may have additional requirements for considering a certificate to be valid. PKIs often require some form of certification that indicates non-revocation, e.g., certificate ψ is not included in a CRL ψ_{CRL} valid at time t. To facilitate such additional requirements, the validation function accepts also an *auxiliary input aux*. A PKI implementation would define the structure and content of *aux*, based on its validity criteria. For example, in PKIX (§4.2) a certificate is considered valid only together with *aux* containing a valid certificate chain and certificate(s) of non-revocation.

Formally, the validity of a certificate ψ is determined using the stateless *certificate validation predicate* of the PKI, namely, $\mathcal{P}.\mathsf{Valid}(\psi, t, store, aux)$, where the inputs include the certificate ψ, the time t, the trust-anchor *store* and the auxiliary information *aux*. When $\mathcal{P}.\mathsf{Valid}(\psi, t, store, aux) = \top$, this means that $\mathcal{P}$.Valid managed to establish trust from ψ to an entity with a self-signed certificate in *store*, and this trust is in a form of a sequence of certificates which are chained together. To obtain this chain of certificates, executing $\mathcal{P}$.VCerts on the same input to $\mathcal{P}$.Valid would output one or more *certificate chains* which $\mathcal{P}$.Valid used to validate ψ.

2.5 Definition of a PKI Scheme

Definition 1 (PKI scheme). *A PKI scheme $\mathcal{P}$ is a set containing (at least) the following PPT algorithms:*

- $\mathcal{P}.\mathsf{Init}(\mathsf{st}, clk, params) \rightarrow (\mathsf{st}, \psi)$*: Takes as input the state*[5] st*, local clock clk, and parameters params, and returns the initialized local state* st *and a self-certified certificate ψ after performing initialization based on the input parameters params and time clk.*

[5]The state st is given in the input to Init (and other stateful operations) for a technical reason. We use the *Modular Security Specifications (MoSS) Framework* [21] to define PKI specifications (see §3.1). In MoSS, operations receive the state st as input. The Init operation is invoked during the MoSS execution, and, therefore, receives the state st as input, like other operations.

- $\mathcal{P}$.Certify(st, *clk*, *tbc*) $\rightarrow$ (st, $\psi/\perp$)*: Takes as input the state* st*, local clock clk and a set of (name, value) fields to be certified tbc, and returns an updated state* st *and either a string* ψ *which is a valid certificate or information which may be used to create a valid certificate, or a failure indicator* $\perp$.
- $\mathcal{P}$.Revoke(st, *clk*, ψ) $\rightarrow$ (st, $\top/\perp$)*: Takes as input the state* st*, local clock clk, and a certificate* ψ*, and returns an updated state* st *and* $\top$ *if* ψ *was revoked successfully or* $\perp$ *if the revocation failed.*
- $\mathcal{P}$.Valid(ψ, *t*, *store*, *aux*) $\rightarrow$ ($\top/\perp$)*: This (stateless) algorithm takes as input a certificate* ψ*, time t, a root store store, and auxiliary information aux, and returns either* $\top$ *or* $\perp$.
- $\mathcal{P}$.VCerts(ψ, *t*, *store*, *aux*) $\rightarrow$ ($\{\psi_i\}/\perp$)*: This (stateless) algorithm takes as input the same input as in* $\mathcal{P}$.Valid*, and returns information used in the validation of* ψ*, typically, a set of certificates* $\{\psi_i\}$.
- $\mathcal{P}$.Aux(st, ψ, *clk*, *store*, *aux*) $\rightarrow$ (st, *aux*/$\perp$)*: This is an optional algorithm requesting the entity, typically a CA, to provide auxiliary information aux, such as CRLs. This auxiliary information, possibly complemented by other auxiliary information, will be used to validate the given certificate* ψ *using root store store.*

PKI schemes might include additional inputs or operations. In particular, we next define *transparent* PKI schemes; such schemes have additional operations used to ensure that valid certificates are publicly available (i.e., "transparent"), allowing to detect discrepancies and suspect certificates. Specifically, transparent PKIs require the ability to instruct monitors to start the monitoring process of a given log ($\mathcal{P}$.Monitor), the ability to retrieve what is known to a monitor regarding a given subject ($\mathcal{P}$.Lookup), and the ability to decide if a given certificate is consistent with the local knowledge of a monitor ($\mathcal{P}$.Audit).

Definition 2 (Transparent PKI scheme). *A transparent PKI scheme* $\mathcal{P}$ *is a PKI scheme with the following additional PPT algorithms:*

- $\mathcal{P}$.Monitor(st, *clk*, ψ_L) $\rightarrow$ st*: Takes as input state* st*, local clock clk and certificate* ψ_L*, and returns an updated state* st*, and starts to monitor (certificates* logged *in) the log corresponding to certificate* ψ_L.
- $\mathcal{P}$.Lookup(st, *clk*, *subject*) $\rightarrow$ (st, Ψ)*: Takes as input state* st*, local clock clk, and an identifier subject, and returns an updated state* st *and a set* $\Psi = \{(\psi_1, aux_1), (\psi_2, aux_2), ...\}$ *of all pairs* (ψ_i, aux_i) *known to the entity s.t.* ψ_i *is the certificate of the given subject, i.e.,* $\psi_i.subject = subject$*, and* aux_i *is the corresponding auxiliary information for* ψ_i.
- $\mathcal{P}$.Audit(st, ψ, *aux*) $\rightarrow$ $\{\psi_L\}/\perp$*: Takes as input state* st*, certificate* ψ*, and auxiliary information aux. Returns* $\perp$ *or a set of log certificates* $\{\psi_L\}$*. Intuitively, return* $\perp$ *if the pair* (ψ, aux) *is known to the invoked entity, or invalid. Otherwise, return* $\{\psi_L\}$*, identifying the set of faulty logs.*

3 PKI REQUIREMENTS

Formally defined requirements are necessary to prove whether a given PKI implementation meets specific requirements, and if so, under what assumptions. In §3.1, we briefly discuss the MoSS framework [21], which we use to define the PKI requirements. Then, we define the following requirements: Existential Unforgeability (§3.2), Accountability (§3.3), Guaranteed and Accountable Δ-Revocation (§3.4), and HL, Audited and Guaranteed Δ-Transparency (§3.5).

3.1 Modular Security Specifications

PKI schemes, deployed and proposed, vary greatly in their designs, operate under different models (assumptions) and aim to satisfy different requirements (goals). To define requirements which apply to different PKI schemes, even if they assume different models, we use the *Modular Security Specifications (MoSS) Framework* [21] for our specifications. MoSS separates the definition of *requirements* from the *models*, simplifying the definition of requirements and allowing for the evaluation of requirements satisfied by PKI schemes under different models. For example, we show that CT does not provide Guaranteed Δ-Transparency and only a weaker notion, HL Δ-Transparency (Table 1).

To illustrate the importance of separating models from requirements, consider this *simplified* Δ-Revocation requirement: a certificate revoked at time t by an honest CA would be considered invalid at any time after $t + \Delta$ (Definition 3). The delay Δ can be a function of network delays, clock bias and design decisions (e.g., periodicity of issuing CRLs). Satisfaction of the Δ-Revocation requirement depends on the clock synchronization and network/communication delay models.

This work follows the MoSS framework [21]; we briefly recall here concepts required for this work. MoSS defines an execution process $\mathbf{Exec}_{\mathcal{A},\mathcal{P}}(params)$ for a given adversary algorithm $\mathcal{A}$ and protocol $\mathcal{P}$. The execution consists of a series of events. In each event e, the adversary decides on the entity invoked $ent[e]$, the operation $opr[e]$, the input $inp[e]$, the real-time clock $\tau[e]$ and the local clock $clk[e]$ of entity $ent[e]$; and the adversary receives the output $out[e]$ of the protocol. When the execution loop of protocol $\mathcal{P}$ with parameters *params* and adversary $\mathcal{A}$ is terminated (by the adversary), the execution outputs a *transcript* (trace), denoted $T \leftarrow \mathbf{Exec}_{\mathcal{A},\mathcal{P}}(params)$. The transcript contains the set of entities T.N, the set of faulty entities T.F as identified by the adversary, the adversary's output $T.out_{\mathcal{A}}$, and the values of all inputs and outputs for each event, e.g., $T.ent[e]$.

We use the extended execution process[6] from [21], which allows three entity-corruption operations: *Get-state* (exposing the state of the entity), *Set-state* (setting the state of the entity) and *set-output* (causing the entity to output specified value).

To define assumptions and restrict the adversary, MoSS uses *model predicates*, which are computed over the execution transcript. For example, Algorithm 1 shows the $\pi^{\text{Drift}}_{\Delta_{clk}}$ model predicate, which ensures bounded clock drift (and may help to understand why we use both $clk[e]$ and $\tau[e]$). This is one of the models (assumptions) from [21] that we use in our security analysis (§5).

Algorithm 1 The $\pi^{\text{Drift}}_{\Delta_{clk}}$ model predicate (from [21])

1: **return** $\forall \hat{e} \in \{1, \ldots, T.e\}$: ▷ *For each event*
2: $\quad |T.clk[\hat{e}] - T.\tau[\hat{e}]| \leq \Delta_{clk} \wedge$ ▷ *Local clock within* Δ_{clk} *drift from real time*
3: $\quad$ **if** $\hat{e} \geq 2$ **then** $T.\tau[\hat{e}] \geq T.\tau[\hat{e}-1]$ ▷ *Monotonically increasing real time*

[6] In [21], these operations are included in the set $\mathcal{X}$, which is specified as part of the notation of the execution process.

An adversary $\mathcal{A}$ *satisfies* model predicate[7] π, if for every protocol $\mathcal{P}$ it holds that $\Pr[\pi(\mathbf{Exec}_{\mathcal{A},\mathcal{P}}(params)) = \perp] \in Negl(|params|)$, i.e., there is a negligible probability that the transcript of a random execution of protocol $\mathcal{P}$ with adversary $\mathcal{A}$ will fail to satisfy π.

Similarly, MoSS uses *requirement predicates*, also computed over the execution transcript T, to define the requirements for a PKI scheme. We define four requirements in the rest of this section, e.g., accountability (Algorithm 3) and transparency (Algorithm 5). A PKI scheme $\mathcal{P}$ *satisfies requirement predicate* $\pi_{\mathcal{R}}$ *assuming model predicate* $\pi_{\mathcal{M}}$, if the probability that $\pi_{\mathcal{R}}(\mathbf{Exec}_{\mathcal{A},\mathcal{P}}(params)) = \perp$ is negligible, where $\mathbf{Exec}_{\mathcal{A},\mathcal{P}}(params)$ is the transcript of a random execution of $\mathcal{P}$ interacting with any PPT adversary $\mathcal{A}$ that satisfies model predicate $\pi_{\mathcal{M}}$.

3.2 The Existential Unforgeability Requirement

The first requirement we define is *Existential Unforgeability* (Algorithm 2). Defining unforgeability for a PKI scheme is more complex than in signature schemes, given that PKI's primary purpose is to establish trust in public keys. Indeed, an attacker may be able to obtain a fraudulent yet valid certificate, where a *fraudulent* certificate is a certificate with a public key which was not self-certified by the subject (the conventions for self-certifying' a certificate are described below). If an attacker controls an entity's key or can obtain a fraudulent yet valid certificate, then the attacker may be able to issue a valid certificate without proper use of the 'Certify' operation.

Therefore, intuitively, for a PKI scheme to satisfy Existential Unforgeability, it means that for every certificate ψ_0, either:

(1) ψ_0 is invalid (Line 3) for the given time t, root store *store* and auxiliary information *aux*, or
(2) the 'Certify' operation of the PKI scheme was used at the entity $\psi_0.issuer$ to certify the non-PKIadded fields of ψ_0 (Line 4), which implies that the adversary may have gotten either ψ_0 or another certificate with the same non-PKIadded fields as ψ_0 correctly from $\psi_0.issuer$ using the 'Certify' operation, or
(3) $\psi_0.issuer$ is rogue (Line 5), which implies that the issuer may have generated certificates without correctly using the 'Certify' operation, or
(4) the validation of ψ_0 uses a valid fraudulent certificate ψ_1 for the issuer of ψ_0 (Line 6), which implies that the private key corresponding to $\psi_1.pk$ may be known or controlled by the adversary.

To be able to identify such a fraudulent public key certificate ψ_1, we require PKI schemes to follow two conventions[8]:

(1) Whenever a benign entity ι generates a public key pk, it would output a pair $(\text{'SelfCert'}, \psi)$ where the issuer and subject fields are both ι, i.e., $\psi.issuer = \psi.subject = \iota$, and $\psi.pk = pk$. We refer to ψ as a *self-certificate*.
(2) The PKI supports a function $\mathcal{P}$.VCerts which takes the same input as $\mathcal{P}$.Valid. When a certificate ψ_0 is found valid by $\mathcal{P}$.Valid, then $\mathcal{P}$.VCerts should return the set of certificates used by $\mathcal{P}$.Valid to validate ψ_0. These certificates may include certificates from *aux* and from *store*.

We use the first convention in the SelfCert function (Lines 22-27). We utilize the second convention and SelfCert in CanFindFraudIssuerCert (Lines 15-21) to find, in the output of $\mathcal{P}$.VCerts, a certificate ψ_1 which certifies a fraudulent public key for $\psi_0.issuer$.

Algorithm 2 π_{EUF}: Existential Unforgeability Requirement

Input: execution transcript T.
Output: $\perp$ if adversary presented a valid yet forged certificate ψ_0, where $\psi_0.issuer$ is benign and $\mathcal{P}$.VCerts did not output a valid public key certificate ψ_1 s.t. $\psi_1.subject = \psi_0.issuer$ and $\psi_1.pk$ was not self-certified by $\psi_0.issuer$. Otherwise, output $\top$ (adversary lost).

1: **procedure** $\pi_{\text{EUF}}(T)$
2: $(\psi_0, t, store, aux) \leftarrow T.out_{\mathcal{A}}$ ▷ *Adversary's output*
3: **if** $\neg\mathcal{P}.\text{Valid}(\psi_0, t, store, aux) \vee$ ▷ *If ψ_0 invalid, or*
4: FieldsProperlyCertified$(T, \psi_0) \vee$ ▷ *fields were certified at issuer, or*
5: $\psi_0.issuer \notin T.\text{N} - T.\text{F} \vee$ ▷ *issuer is rogue, or*
6: CanFindFraudIssuerCert$(T, \psi_0, t, store, aux)$ ▷ *there is a fraudulent issuer cert,*
7: **then return** $\top$ ▷ *then adversary failed*
8: **else return** $\perp$ ▷ *else, adversary wins*
9: **end procedure**
10: **procedure** FieldsProperlyCertified(T, ψ)
11: **return** $\exists e$ s.t. $T.opr[e] =$ 'Certify' $\wedge$ ▷ *If 'Certify' operation was executed*
12: $T.inp[e] = \left\{ (field, val) \;\middle|\; \begin{array}{l} \exists (field, val) \in \psi.tbc \wedge \\ field \notin \psi.PKIadded \end{array} \right\} \wedge$ ▷ *On non-PKIadded fields of $\psi.tbc$*
13: $T.ent[e] = \psi.issuer$ ▷ *By the entity claimed as the issuer*
14: **end procedure**
15: **procedure** CanFindFraudIssuerCert$(T, \psi_0, t, store, aux)$
16: $vcerts \leftarrow \mathcal{P}.\text{VCerts}(\psi_0, t, store, aux)$ ▷ *Certificates used for validating ψ_0*
17: **return** $\exists \psi_1 \in vcerts$ **s.t.** ▷ *Contain a certificate ψ_1 such that*
18: $\left[\begin{array}{l} \mathcal{P}.\text{Valid}(\psi_1, t, store, aux) \vee \\ \left[\begin{array}{l} \psi_1 \in store \wedge \\ \psi_1.subject = \psi_1.issuer \end{array}\right] \end{array}\right] \wedge$ ▷ *ψ_1 is a valid certificate, or* ▷ *ψ_1 is in store and* ▷ *has same subject and issuer*
19: $\psi_1.subject = \psi_0.issuer \wedge$ ▷ *The subject of ψ_1 is the issuer of ψ_0*
20: $\neg$SelfCert$(T, \psi_0.issuer, \psi_1.pk)$ ▷ *ψ_1 is fraudulent*
21: **end procedure**
22: **procedure** SelfCert(T, ι, pk)
23: **return** $\exists e, \psi$ **s.t.** $T.ent[e] = \iota \wedge$ ▷ *Entity ι*
24: $T.out[e] = (\text{'SelfCert'}, \psi) \wedge$ ▷ *outputted certificate ψ*
25: $\psi.pk = pk \wedge$ ▷ *certifying given pk*
26: $\psi.issuer = \psi.subject = \iota$ ▷ *using the same identity*
27: **end procedure**

3.3 The Accountability Requirement

We now define the *Accountability* requirement (Algorithm 3). Defining Accountability turned out to be more complex than we initially anticipated. Intuitively, for a PKI scheme to satisfy Accountability, an 'accountable' root CA should be identifiable for every valid certificate ψ_0. However, this is where things become less straightforward.

Following the discussion in §3.2, there is no guarantee that a valid certificate's non-PKIadded fields were properly certified using the 'Certify' operation. Therefore, our Accountability requirement does not automatically consider the issuer listed in a certificate ψ, or a root CA used to validate ψ, to be accountable for ψ. Rather, we say that a certificate ψ' is *accountable* for certificate ψ if the

[7] MoSS [21] defines more general models and requirements, which supports non-negligible probability β of failures for some predicates; this is not required for our models and requirements.
[8] The conventions do not require changes to existing PKI implementations.

'Certify' operation of the PKI scheme was used at $\psi'.subject$ to certify the non-PKIadded fields of ψ, or the 'Certify' operation of the PKI scheme was used at $\psi'.subject$ to certify the non-PKIadded fields of another certificate which is accountable for ψ.

The Accountability requirement requires that there is either:

(1) a bad root certificate (Line 5), where a *bad* certificate is a certificate which has a rogue subject or is a fraudulent certificate, where *fraudulent* is defined in Section 3.2, or
(2) a root certificate which is accountable for a valid bad certificate (Line 6), or
(3) a root certificate which is accountable for ψ_0 (Line 7).

Our definition of accountability only ensures that a bad or accountable root certificate is in the output of $\mathcal{P}$.VCerts; it does not say *how* such a root certificate from the output of $\mathcal{P}$.VCerts would be identified - this may depend on the PKI. If a root certificate ψ_r which appears like it should be accountable for ψ_0 is identified, however, then the subject of ψ_r can be audited (using some legal or other procedures) to check if it admits to owning $\psi_r.pk$. If ψ_0 is a fraudulent certificate and $\psi_r.subject$ admits to owning $\psi_r.pk$, then $\psi_r.subject$ can be held accountable for ψ_0, depending on legal or other policies. Otherwise, if $\psi_r.subject$ denies owning $\psi_r.pk$, then ψ_r should be removed from the root store, since either ψ_r is a bad certificate or $\psi_r.subject$ is not being honest when it denies ownership of $\psi_r.pk$. Subjects of other (non-root) certificates which appear like they should be accountable for ψ_0 may similarly be audited and may possibly held accountable if they admit to owning the relevant public keys.

3.4 Revocation Requirements

Certificates sometimes need to be revoked prior to their expiration date. Revocation is initiated by the issuer for various reasons, including the loss or compromise of the private key corresponding to a certified public key, or when the certificate contains incorrect or outdated information.

It takes time to communicate the revocation of a certificate to the relying parties; we use Δ to denote the maximum allowed time. The 'grace period' Δ is typically required for three reasons: (1) the bias between the clock of $\psi.issuer$ and the clock of the relying party validating ψ, (2) the communication delay from $\psi.issuer$ to the relying parties, and (3) the time for any proof that ψ is non-revoked which was issued prior to its revocation to become stale, i.e., expire.

Guaranteed Δ-Revocation and the Zombie certificate attack. Preferably, we would like revocation to be fully enforced, i.e., if the issuer revokes a certificate at time t, then no relying party should accept the certificate after $t+\Delta$. We refer to this property as *Guaranteed Δ-Revocation*. This ensures that revocation by a benign CA is always effective, with at most Δ delay; a relying party cannot be misled to rely on a revoked certificate after $t+\Delta$.

The most well-known revocation mechanism is the X.509 standard *Certificate Revocation List (CRL)* mechanism, the 'basic' revocation mechanism of PKIX and CT; see §4.2 or [45] for details. The CRL is a timestamped list of revoked certificates signed by their issuer. CRLs are issued periodically, say once every Δ seconds. A relying party R considers certificate ψ as non-revoked at time t, if R received a CRL from the issuer of ψ, issued at $t-\Delta$ or later, which does not list ψ as a revoked certificate.

Algorithm 3 π_{ACC}: Accountability Requirement

Input: execution transcript T.
Output: $\perp$ if adversary presented a valid certificate ψ_0 and the output of $\mathcal{P}$.VCerts includes no bad root certificate, no root certificate which is accountable for a valid bad certificate among the certificates in the output of $\mathcal{P}$.VCerts, and no root certificate which is accountable for ψ_0. Otherwise, output $\top$ (adversary lost).

1: **procedure** $\pi_{\text{ACC}}(T)$
2: $(\psi_0, t, store, aux) \leftarrow T.out_{\mathcal{A}}$ ▷ *Adversary's output*
3: $vcerts \leftarrow \mathcal{P}.\text{VCerts}(\psi_0, t, store, aux)$
4: **if** $\neg\mathcal{P}.\text{Valid}(\psi_0, t, store, aux) \vee$ ▷ *Acc. holds if ψ_0 is invalid, or*
5: $\left[\begin{array}{l}\exists\psi_r \in store \cap vcerts \text{ s.t.}\\ \quad \text{BadCert}(T, \psi_r)\end{array}\right] \vee$ ▷ *a root cert is bad*
6: $\left[\begin{array}{l}\exists\psi_r \in store,\ \psi' \in vcerts \text{ s.t.}\\ \quad \text{Accountable}(T, \psi', \psi_r, vcerts) \wedge\\ \quad \mathcal{P}.\text{Valid}(\psi', t, store, aux) \wedge\\ \quad \text{BadCert}(T, \psi')\end{array}\right] \vee$ ▷ *a root cert is accountable for a valid bad cert ψ'*
7: $\left[\begin{array}{l}\exists\psi_r \in store \cap vcerts \text{ s.t.}\\ \quad \text{Accountable}(T, \psi_0, \psi_r, vcerts)\end{array}\right]$ ▷ *a root cert is accountable for ψ_0*
8: **then return** $\top$ ▷ *then adversary failed*
9: **else return** $\perp$ ▷ *else, adversary wins*
10: **end procedure**
11: **procedure** $\text{Accountable}(T, \psi, \psi', vcerts)$
12: **return** $\text{FieldsCertified}(T, \psi, \psi'.subject) \vee$ ▷ *Fields were certified at $\psi'.subject$*
13: $\left[\begin{array}{l}\exists\psi'' \in vcerts \text{ s.t.}\\ \quad \text{FieldsCertified}(T, \psi'', \psi'.subject) \wedge\\ \quad \text{Accountable}(T, \psi, \psi'', vcerts)\end{array}\right]$ ▷ *Fields of a cert ψ'' which is accountable for ψ were certified at $\psi'.subject$*
14: **end procedure**
15: **procedure** $\text{BadCert}(T, \psi)$
16: **return** $\psi.subject \notin T.\text{N} - T.\text{F} \vee$ ▷ *$\psi.subject$ is not benign, or*
17: $\neg\text{SelfCert}(T, \psi.subject, \psi.pk)$ ▷ *ψ is fraudulent (see Algorithm 2)*
18: **end procedure**
19: **procedure** $\text{FieldsCertified}(T, \psi, ent)$
20: **return** $\exists e$ s.t. $T.opr[e] =$ 'Certify' $\wedge$ ▷ *If 'Certify' operation was executed*
21: $T.inp[e] = \left\{(field, val) \;\middle|\; \begin{array}{l}\exists(field, val) \in \psi.tbc \wedge\\ field \notin \psi.PKIadded\end{array}\right\} \wedge$ ▷ *On non-PKIadded fields of $\psi.tbc$*
22: $T.ent[e] = ent$ ▷ *By entity ent*
23: **end procedure**

The PKIX specifications [45] require the CRL to be validated using a valid public key certificate ψ' issued to the issuer of ψ, i.e., $\psi'.subject = \psi.issuer$. Additionally, they require that the trust anchor for the certification path of ψ' be the same as the trust anchor used to validate ψ. However, as we next show, the CRL mechanism *fails to ensure Guaranteed Δ-Revocation*; we later define the weaker *Accountable Δ-Revocation* property, which the CRL mechanism ensures.

Claim 1 (The Zombie certificate attack). *PKIX and CT, using the CRL revocation mechanism, fail to ensure the Guaranteed Δ-Revocation requirement.*

Proof. Let ψ_0 be a certificate issued by a benign CA $\psi_0.issuer$, with validity period $[\psi_0.from, \psi_0.to]$, which is later revoked by $\psi_0.issuer$ at time t_R s.t. $\psi_0.from < t_R < \psi.to - \Delta$, s.t. for some *store*, *aux*, and t, t' s.t. $t < t_R$ and $t_R + \Delta < t' \leq \psi.to$, $\text{PKIX}_C.\text{Valid}(\psi_0, t, store, aux)$ returns $\top$ and $\text{PKIX}_C.\text{VCerts}(\psi_0, t, store, aux)$ returns a PKIX_C-valid certificate chain $\xi = \psi_0 - \psi_1 - \ldots - \psi_r$ s.t. for each ψ_i in the chain, $\psi_i.to \geq t'$.

Consider an attacker which controls a rogue CA $\iota_R \in T.\mathsf{F}$ which is either the trust anchor of ψ_0's certificate chain, or an intermediate CA (of ψ_0's certificate chain) which has a chain rooted by the same trust anchor as of ψ_0 and the certificates in the chain are valid at time t' w.r.t. *store* and some aux'. That is, there exists ψ'_i s.t. $\psi_i.subject = \iota_R$ and $\mathsf{PKIX}_C.\mathsf{Valid}(\psi'_i, t', store, aux')$ returns $\top$ and $\mathsf{PKIX}_C.\mathsf{VCerts}(\psi'_i, t', store, aux')$ returns a PKIX_C-valid certificate chain rooted by ψ_r, i.e., a chain $\xi' = \psi'_i - \ldots - \psi_r$. In either case, the adversary can generate aux'' and a certificate ψ''_1 such that $\psi''_1.issuer = \iota_R$ and $\psi''_1.subject = \psi_0.issuer$, and where the corresponding private key of ψ''_1 is known to the attacker (of course, $\psi''_1.pk$ is *not* the public key used by the benign $\psi_0.issuer$), and where $\xi'' = \psi_0 - \psi''_1 - \psi'_i - \ldots - \psi_r$ is a PKIX-valid certificate chain w.r.t. time t', *store*, and aux''. Therefore, the attacker can next generate a (fake) CRL ψ''_{CRL} which has a validity period s.t. $t' \in [\psi''_{CRL}.from, \psi''_{CRL}.to]$, which does not list ψ_0 as revoked, and which can be verified using $\psi''_1.pk$.

Consider a benign relying party ι which receives ψ_0 at time t' together with aux''', which includes all of aux'' (including the certificate chain ξ'') and the CRL ψ''_{CRL}, and suppose that the relying party does not receive any other CRL conflicting with this time period (e.g., the real CRL which covers time t' from the benign $\psi_0.issuer$). The CRL ψ''_{CRL} will therefore be considered valid at time $t' > t_R + \Delta$ w.r.t. *store* and aux'', and therefore ι will consider ψ_0 to be non-revoked, in contrast to the Guaranteed Δ-Revocation requirement. □

Accountable Δ-Revocation. This requirement *allows* ψ to be considered valid even after being revoked by its benign issuer CA $\psi.issuer$, provided that the PKI identifies a different certificate $\psi_R \neq \psi$ in the output of $\mathcal{P}.\mathsf{VCerts}(\psi, t, store, aux)$ which is a valid-yet-rogue certificate for a benign subject, i.e., $\psi_R.subject \in T.\mathsf{N} - T.\mathsf{F}$ but $\psi_R.pk$ is not a correct public key generated by $\psi_R.subject$. Namely, the benign $\psi.issuer$ did not output a corresponding self-signed certificate for $\psi_R.pk$. If the PKI can identify the rogue CA certificate ψ_R, and assuming that the PKI ensures *accountability*, then we can identify one or more CAs which could be held accountable for ψ_R, including a root CA which is (ultimately) accountable for ψ_R, or a bad root certificate which should be removed from the root store. For example, in the Zombie certificate attack of Claim 1, we will have $\psi_R = \psi'$. To validate ψ after its revocation, using the fake CRL ψ'_{CRL}, the attacker must include $\psi' = \psi_R$; therefore, we can identify the rogue issuer ι_R.

The Guaranteed Δ-Revocation requirement is stronger, as it prevents the use of revoked certificates. In contrast, Accountable Δ-Revocation only provides accountability after an attack has occurred. Notably, the stipulation in [45] that the trust anchor for the certification path of ψ' must be the same as the trust anchor used to validate ψ underscores this point. This requirement would not be necessary if the designers were only aiming for Accountable Δ-Revocation. We believe this indicates a clear intention to *prevent* the use of revoked certificates (Guaranteed Δ-Revocation), rather than merely offering post-attack accountability (Accountable Δ-Revocation). Importantly, ensuring Guaranteed Δ-Revocation is indeed feasible for PKI. For instance, PKIX could have mandated the validation of the CRL using the same public key that is used for validating the certificate itself.

Algorithm 4 $f^{\mathsf{Rev}}_{\Delta}$: the Δ-Revocation function

Input: transcript T.
Output: 'G' if the adversary fails and Δ-Revocation is guaranteed, 'A' if Δ-Revocation is only accountable, and ⊥ if the adversary wins.

procedure $f^{\mathsf{Rev}}_{\Delta}(T)$
  $(\psi, t, store, aux) \leftarrow T.out_{\mathcal{A}}$ ▷ *Extract adversary's output*
  if $\left[\begin{array}{ll} \mathcal{P}.\mathsf{Valid}(\psi, t, store, aux) \wedge & \triangleright\ \psi \text{ is valid} \\ \psi.issuer \in T.\mathsf{N} - T.\mathsf{F} \wedge & \triangleright\ \psi\text{'s issuer is benign} \\ \textsc{CertifyAndRevokeRequested}(T, \psi, t, \Delta_{\mathsf{Rev}}) & \triangleright\ \textit{See Algorithm 13} \end{array}\right]$
    if $\textsc{CanFindFraudulentCert}(T, \psi, t, store, aux)$ ▷ *See Algorithm 14*
      then return 'A' ▷ *ψ is 'revoked-yet-valid', but fraudulent cert was detected*
      else return ⊥ ▷ *Adversary wins!*
  else return 'G' ▷ *Adversary fails! ⟹ Guaranteed revocation*
end procedure

We define both the Guaranteed and Accountable Δ-Revocation requirement predicates in Definition 3. In §5, we show that PKIX and CT, with the CRL revocation mechanism, ensure Accountable Δ-Revocation.

Definition 3 (Δ-Revocation requirements). *We define the Guaranteed and Accountable Δ-Revocation predicates as:*

$$\pi^{GtdRev}_{\Delta}(T) = \left\{ \begin{array}{l} \top \text{ if } f^{\mathsf{Rev}}_{\Delta}(T) = \text{'G'}, \\ \bot \text{ otherwise} \end{array} \right\}$$

$$\pi^{AccRev}_{\Delta}(T) = \left\{ \begin{array}{l} \top \text{ if } f^{\mathsf{Rev}}_{\Delta}(T) \in \{\text{'G'}, \text{'A'}\}, \\ \bot \text{ otherwise} \end{array} \right\}$$

Algorithm 4 defines the function $f^{\mathsf{Rev}}_{\Delta}$. □

3.5 Transparency Requirements

Accountability, as described in §3.3, is a reactive defense, whose goal is to *deter* rogue or negligent behavior by root CAs. For many years, this reactive measure was seen as sufficient under the assumption that root CAs, and CAs certified by them, are respectable and trustworthy entities who would not risk being implicated in issuing rogue certificates, intentionally or otherwise. However, repeated cases of rogue certificates issued by compromised or dishonest CAs, have proven this assumption to be overly optimistic. Punishing root CAs is non-trivial: beyond negative publicity, punishment has often been ineffective [51, 2, 25]. Furthermore, punishing a CA requires that a rogue certificate is *discovered*. An attacker could reduce the risk of discovery by minimizing the exposure of the rogue certificate. Efforts such as Perspectives Project [59] and the EFF SSL Observatory[16] provide some assistance in discovering rogue certificates but are not sufficient to address this concern.

This motivated *transparent* PKI designs, most notably, the standardized and deployed *Certificate Transparency (CT)* [31, 50]. To support transparency, a certificate must be logged at one or more *loggers*. Loggers are parties committed to including certificates in a *public log* they maintain, and to making this log available to third parties called *monitors*. Each monitor keeps tabs on the certificates logged by one or, usually, more loggers; in addition, monitors may detect suspicious certificates and inform interested parties, such as domain owners and relying parties.

In other words, transparency aims to prevent a CA from stealthily generating a rogue certificate ψ to attack select victims. Transparency facilitates early detection of rogue certificates issued by a

corrupt, compromised or negligent CA. By demanding that a valid certificate must be transparent, we ensure the detection of rogue and suspicious certificates. There is some unavoidable delay from the time a certificate is submitted to a log until the relevant monitors are aware of it. This delay is primarily due to the significant time allowed between receiving a certificate and including it in a new version of the log.

The Δ-Transparency Requirements. Similarly to the case with revocation, we found that the CT standard [50] does not satisfy the natural, strong and *guaranteed* transparency. *Guaranteed-Transparency* means that an honest monitor which is monitoring the relevant logs is always aware of the logging of a certificate[9] after a specific delay.

CT, however, satisfies only a weaker notion, *HL-Transparency*, where transparency only holds if a certificate is logged by at least one benign logger. The Chrome implementation of CT ([27], Appendix D.2) also ensures another weak notion of transparency, *Audited-Transparency*. In this case, an honest monitor which is monitoring the relevant (possibly all corrupt) logs might be unaware of a valid certificate[10], but when presented with such a certificate during an *audit*, the monitor outputs a log certificate of a corrupt logger responsible for this lack of transparency.

Definition 4 (Δ-Transparency requirements). *We define the Guaranteed, HL and Audited Δ-Transparency predicates as:*

$$\pi_{\Delta}^{GtdTra}(T) = \left\{ \begin{array}{l} \top \text{ if } f_{\Delta}^{\mathsf{Tra}}(T,\bot) = \text{`G'} \\ \bot \text{ otherwise} \end{array} \right\}$$

$$\pi_{\Delta}^{HLTra}(T) = \left\{ \begin{array}{l} \top \text{ if } f_{\Delta}^{\mathsf{Tra}}(T,\top) = \text{`G'} \\ \bot \text{ otherwise} \end{array} \right\}$$

$$\pi_{\Delta}^{AudTra}(T) = \left\{ \begin{array}{l} \top \text{ if } f_{\Delta}^{\mathsf{Tra}}(T,\bot) \in \{\text{`G'}, \text{`A'}\}, \\ \bot \text{ otherwise} \end{array} \right\}$$

Algorithm 5 defines the function $f_{\Delta}^{\mathsf{Tra}}$. □

Algorithm 5 $f_{\Delta}^{\mathsf{Tra}}$: the Δ-Transparency function

Input: transcript T and the HL flag.
Output: 'G', if Δ-Transparency is guaranteed; if the HL input flag is set, then we require that one of the SCTs of the certificate is from an honest log. 'A', if Δ-Transparency is (only) audited. ⊥ if neither guaranteed nor audited Δ-Transparency holds (adversary wins).

procedure $f_{\Delta}^{\mathsf{Tra}}(T, \text{HL})$
  $(\psi, t, store, aux, \iota_M) \leftarrow T.out_{\mathcal{A}}$ ▷ *Extract adversary's output*
  if HL ∧ NoHonestLog$(\psi, t, store, aux)$ **return** 'G' ▷ *HL w/o honest log (Alg. 15)*
  if $\psi.from \le t - \Delta \wedge$ ▷ *ψ issued*
    $\mathcal{P}$.Valid$(\psi, t, store, aux) \wedge$ ▷ *ψ is valid*
    $\iota_M \in T.\mathsf{N} - T.\mathsf{F} \wedge$ ▷ *ι_M is benign*
    MonitoringCertLogs$(T, \psi, t - \Delta, store, aux, \iota_M) \wedge$ ▷ *See Algorithm 16*
    MonitorIsUnaware(T, ψ, t, ι_M) ▷ *See Algorithm 17*
    if Audited$(T, \psi, t, aux, \iota_M)$ ▷ *See Algorithm 18*
      if FailedIdentifyCorrupted$(T, \psi, t, store, aux, \iota_M)$ ▷ *See Algorithm 19*
        then return ⊥ ▷ *Adversary wins!*
        else return 'A' ▷ *Audited transparency*
  else return 'G' ▷ *Adversary fails! Guaranteed transparency*
end procedure

4 PROVABLY-SECURE PKI SCHEMES

In this work, we present three implementation of PKI schemes based on two PKIs used in practice: PKIX following [45] (see §4.2) and CT following the CT 2.0 specification [50] (see Appendix D.1), using PKIX for aspects not covered in [50]. In addition, we provide a specification of CTwAudit, a variant of CT augmented by an auditing mechanism [27] (see Appendix D.1). While this variant is not standardized, we found it important to include, as it is supported as an option by Google Chrome, the most popular browser.

In all implementations, we use the CRL revocation mechanism[11]. The schemes cannot, realistically, cover all aspects of the corresponding (lengthy and not fully specified) RFCs, but we have done our best to retain all aspects related to the security requirements. For example, we include the details of certificate chains and basic constraints, but omit the (less deployed and less relevant) length, name and policy constraints.

4.1 Certificate Scheme

Typical PKI schemes use an encoding scheme Θ to encode the certificate fields (Table 2), and a signature scheme $\mathcal{S}$ to sign the encoded fields. In practice, the PKI schemes use one or more encoding schemes, e.g., BER, DER, PEM, and SPKI [45, 48, 23, 41], and one or more signature schemes, e.g., RSA and ECDSA [43, 44]. We could have specified a PKI implementation with a specific encoding scheme Θ and signature scheme $\mathcal{S}$, and then prove security by reduction to the unforgeability of $\mathcal{S}$; however, this would be inconvenient and inefficient, given the many possible encodings and signature schemes.

Instead, we define an abstract *certificate scheme* C and its existential unforgeability requirement (Definition 5). We then present a simple, generic design of a certificate scheme from a signature scheme $\mathcal{S}$ and an encoding scheme Θ (Definition 6), and prove it satisfies unforgeability by a reduction to the unforgeability of the signature scheme. Then, we present the implementations of the PKI schemes (PKIX and CT) using any certificate scheme C.

The new certificate scheme abstraction offers several advantages. Firstly, it simplifies the description of PKI schemes. Secondly, it eases the analysis and reductions. Thirdly, analyzing multiple PKI schemes, each using different encoding and signature schemes, would be impractical without this abstraction. Fourthly, it may enable PKI schemes that utilize different constructions than those in Definition 6, such as signatures over digests of accumulators or

[9] The monitor may not be aware of *all* the fields of the certificate. In particular, a certificate ψ may include a field $\psi.PKIadded$ listing fields which were added to the certificate by the PKI, possibly after the other fields were logged. The monitor should be aware, however, of the other fields in the certificate (those which are not in $\psi.PKIadded$).

[10] This is a simplification. More precisely, in addition to the fields in $\psi.PKIadded$ of which the monitor may be unaware, the monitor may also be unaware of the other fields in the certificate (those which are not in $\psi.PKIadded$).

[11] We chose CRL over other existing revocation mechanisms for its simplicity and the fact that it is not very different from other mechanisms. For example, OCSP introduces the issues of OCSP responders as distinct parties, stapled OCSP and more, which introduces more complexity. While the implementation and analysis can be extended to support additional revocation mechanisms such as OCSP, we leave it for future work.

Merkle trees, allowing validation with only parts of the certified data. Finally, certificate schemes appear useful for applications beyond PKI schemes.

Definition 5 (Certificate scheme). *A certificate scheme C is defined as a tuple of PPT algorithms:*

$$C = (\mathsf{KeyGen}, \mathsf{Certify}, \mathsf{Verify}, \mathsf{Decode})$$

where:

- $C.\mathsf{KeyGen}(1^n) \to (\{0,1\}^*, \{0,1\}^*)$*: Takes as input security parameter 1^n, and returns a pair (sk, pk) of keys, where sk is a private (certification) key and pk is a corresponding public verification key.*
- $C.\mathsf{Certify}_{sk}(tbc) \to \psi$*: Takes as input a set $tbc = \{(n_i, v_i)\}$* (to be certified) *of name-value pairs, where n_i is an alphanumeric* field name *and $v_i \in \{0,1\}^*$ is a* field value*, and using the private certifying key sk, returns $\psi \in \{0,1\}^*$. Field names are unique, i.e., $(\forall i \neq j)(n_i \neq n_j)$. We say that ψ is a* certificate *of tbc.*[12]
- $C.\mathsf{Verify}_{pk}(\psi) \to \{\top, \bot\}$*: Takes as input a certificate $\psi \in \{0,1\}^*$, and using the public verification key pk, returns $\top$ if the certificate is valid, i.e., was certified using the private certifying key corresponding to the public key pk, and $\bot$ otherwise.*
- $C.\mathsf{Decode}(\psi) \to tbc \cup \{\bot\}$*: Takes as input a certificate $\psi \in \{0,1\}^*$, and returns a set tbc, of name-value pairs, as defined for the $C.\mathsf{Certify}$ algorithm above, or $\bot$ (when decoding fails).*

Dot notation. We use dot notation to extract the value of a field from a tbc set, e.g., $tbc.issuer$ denotes the value of the *issuer* field in tbc. We also use dot notation to denote the value of a field in a certificate's tbc set, e.g., $\psi.issuer$ denotes the value of the *issuer* field of $tbc \leftarrow C.\mathsf{Decode}(\psi)$.

We say that certificate scheme C ensures *correctness* if for every $(sk, pk) \leftarrow C.\mathsf{KeyGen}(1^n)$ and for every set of name-value pairs tbc, with unique names, the following holds:

(1) $C.\mathsf{Verify}_{pk}(C.\mathsf{Certify}_{sk}(tbc)) = \top$, and
(2) $C.\mathsf{Decode}(C.\mathsf{Certify}_{sk}(tbc)) = tbc$

We say that C ensures *Existential Unforgeability* if for every PPT $\mathcal{A}$, the probability of $\mathcal{A}$ to win in the *Existential Certificate Forgery* game (Algorithm 6), $\Pr[ECF(n, C, \mathcal{A})]$, is negligible.

Algorithm 6 The Existential Certificate Forgery (ECF) game

$\underline{ECF(n, C, \mathcal{A})}$:
1: $(sk, pk) \leftarrow C.\mathsf{KeyGen}(1^n)$ ▷ *Generate keys*
2: $\psi \leftarrow \mathcal{A}^{C.\mathsf{Certify}_{sk}(\cdot)}(pk)$ ▷ *Give $\mathcal{A}$ oracle access and pk*
3: $tbc \leftarrow C.\mathsf{Decode}(\psi)$ ▷ *Decode tbc from ψ*
4: **if** $C.\mathsf{Verify}_{pk}(\psi) \wedge$ ▷ *ψ is valid*
5: $tbc \neq \bot \wedge$ ▷ *tbc is not empty*
6: $tbc \notin \{\mathcal{A}$'s inputs to $C.\mathsf{Certify}_{sk}$ oracle$\}$ ▷ *$\mathcal{A}$ did not cheat*
7: **then return** 1 ▷ *$\mathcal{A}$ won*
8: **else return** 0 ▷ *$\mathcal{A}$ failed*

We now define a generic construction $C^{\mathcal{S},\Theta}$ of a certificate scheme from a public-key signature $\mathcal{S}$ and a pair of invertible encoding functions $\Theta = (\Theta_C, \Theta_S)$, both mapping sets of name-value pairs to binary strings. PKIX, CT and other deployed PKI schemes use certificate schemes following this construction. The invertible encoding schemes (Θ_C, Θ_S) are detailed, and composed of different encoding for specific certificate types, making the use of the certificate-scheme abstraction and this generic construction essential to understand, analyze and prove security of PKIs.

Definition 6. *Let $\Theta = (\Theta_C, \Theta_S)$ be a pair of invertible functions from sequences of name-value pairs with unique names, to binary strings, and let $\mathcal{S}$ be a signature scheme. The $C^{\mathcal{S},\Theta}$ certificate scheme is defined as:*

$$C^{\mathcal{S},\Theta}.\mathsf{KeyGen}(1^n) \equiv \mathcal{S}.\mathsf{KeyGen}(1^n)$$
$$C^{\mathcal{S},\Theta}.\mathsf{Certify}_{sk}(tbc) \equiv \Theta_S(\{(\text{`tbs'}, \Theta_C(tbc)), (\text{`}\sigma\text{'}, \mathcal{S}.\mathsf{Sign}_{sk}(\Theta_C(tbc)))\})$$
$$C^{\mathcal{S},\Theta}.\mathsf{Verify}_{pk}(\psi) \equiv \mathcal{S}.\mathsf{Verify}_{pk}(\Theta_S^{-1}(\psi)[\text{`tbs'}], \Theta_S^{-1}(\psi)[\text{`}\sigma\text{'}])$$
$$C^{\mathcal{S},\Theta}.\mathsf{Decode}_{pk}(\psi) \equiv \Theta_C^{-1}(\Theta_S^{-1}(\psi)[\text{`tbs'}])$$

Lemma 2. *Let $\mathcal{S}$ be an existentially-unforgeable signature scheme, and $\Theta = (\Theta_C, \Theta_S)$ be a pair of invertible functions (from sequences of name-value pairs with unique names to binary strings). Then $C^{\mathcal{S},\Theta}$, defined in Definition 6, is an existentially-unforgeable certificate scheme.*

Proof: Direct reduction to the security of $\mathcal{S}$. □

4.2 PKIX (with CRLs)

We now explain the construction of PKIX_C, which implements PKIX and CRL using an underlying certificate scheme C.

Entities and state. In PKIX_C, the set N of stateful entities is comprised of certificate authorities (CAs) which issue and revoke public key certificates. Some CAs are *root CAs* (trust anchors), which relying parties trust directly (by maintaining their self-signed certificates in *store*). Other CAs are *intermediate CAs*, which relying parties trust based on a certificate chain ending at a root CA. Each CA maintains a local state st which contains the following information:

- $\mathsf{st}.\iota$: the identifier of the CA.
- $\mathsf{st}.sk$: the CA's secret signing key.
- $\mathsf{st}.pk$: the CA's public verification key.
- $\mathsf{st}.CRL$: the list of all revoked certificates.
- $\mathsf{st}.\Delta_r$: CRL's validity period.
- $\mathsf{st}.certs$: the set of certificates issued by the CA.

Certificate fields and scheme. PKIX_C and CT use the generic certification scheme $C^{\mathcal{S},\Theta}$ of Definition 6, where the encoding schemes are defined in [45] and the certificate fields listed in Table 2.

Implementation. We now present the PKIX implementation, with CRLs (Algorithms 7-11), consisting of the operations in Definition 1.

PKIX_C.Init *(Algorithm 7).* The PKIX Init function generates the CA's keypair (Line 4) and self-certifies the public key (Line 5). It initializes the state st of the CA: the CA's identity $\mathsf{st}.\iota$, the $\mathsf{st}.\Delta_r$ parameter (the CRL validity period) from *params*, and define the sets of certificates issued ($\mathsf{st}.certs$) and revoked ($\mathsf{st}.CRL$) by the CA as empty.

PKIX_C.Certify *(Algorithm 8).* PKIX certifies only public-key certificates. Hence, the algorithm sets the certificate's type to 'public-key' (Line 5), and then signs tbc using $C.\mathsf{Certify}(\mathsf{st}, clk, tbc)$ and the CA's secret signing key (Line 7). Then, a copy of ψ is stored locally (Line 8) and the algorithm outputs ψ.

[12] $C.\mathsf{Certify}$ adds the 'issuer', 'type', and 'PKIadded' fields to $\psi.tbc.PKIadded$.

PKIX$_C$.Revoke *(Algorithm 9).* The algorithm verifies that the certificate to be revoked was issued by the CA (Line 1). If so, the algorithm adds the certificate to the list of revoked cert (Line 2).

PKIX$_C$.VCerts *(Algorithm 10).* In PKIX, this function simply returns the set of all certificates in *store* as well as certificates in *aux*, including certificates in chains.

PKIX$_C$.Valid *(Algorithm 11).* The PKIX$_C$.Valid algorithm ensures two main things: (1) the inputted certificate ψ has a valid chain of certificates from ψ to one of the root CAs, and that (2) every certificate in the chain (including ψ) was not revoked. The algorithm starts by validating that the inputted certificate ψ has a type *'PubKey'* (Line 1). Then, the algorithm verifies that the inputted *aux* parameter contains a chain ξ (Line 5), which contains a sequence of certificates from ψ to ψ_{root} (Lines 6-7), and that for each certificate $\xi[k]$:

(1) The inputted time t is within the validity period of $\xi[k]$ (Lines 8 and 13), and
(2) $\xi[k+1]$ has indeed issued $\xi[k]$ (Lines 7, 9, 12 and 14), and
(3) For $\xi[k]$ from ψ to the second-to-last certificate before ψ_{root}, that $\xi[k{+}1]$ is a public key certificate (Line 15) for a certificate authority (Line 16).

To ensure that none of the certificates were revoked (Lines 10 and 17), the algorithm ensures that *aux* contains a valid CRL ψ_{CRL} with valid certificate chain ξ_{CRL} (Line 20) for every certificate $\xi[k]$ such that:

(1) ψ_{CRL} is a CRL certificate (Line 21), and
(2) ψ_{CRL} has the same issuer as $\xi[k]$ (Line 22), and
(3) The relevant CRL does not contain the serial of $\xi[k]$ (Line 23), and
(4) That ψ_{CRL} has a valid chain that terminates in a root CA (Line 24).

If any of the aforementioned checks fail, the algorithm outputs $\perp$, otherwise, the certificate is considered valid and therefore, the algorithm outputs $\top$.

PKIX$_C$.Aux *(Algorithm 12).* First, the algorithm removes all the certificates that expired from the current CRL (Line 1). Then, it generates an updated CRL, i.e., defines data to be certified *tbc* with CRL type (Line 2), sets the issuer and validity period (Lines 3-5), and sets the list of revoked certificates (Line 6). Finally, the algorithm signs *tbc* using C.Certify and the CA's secret signing key (Line 7) and outputs the CRL certificate along with the local state (Line 8).

5 SECURITY ANALYSIS

We analyze the security of the three schemes we present (PKIX$_C$, CT$_{C,h}$, and CTwAudit$_{C,h}$) against the security requirements defined in §3. First, in §5.1, we describe the models assumed by these protocols, following [21]; we provide the corresponding model predicates in Appendix C. Then, in §5.2, we prove that all three PKIs satisfy Existential Unforgeability, Accountability and Accountable Revocation. Lastly, in §5.3 we sum-up the results of the analysis of Δ-Transparency for CT$_{C,h}$ and CTwAudit$_{C,h}$; we present the complete analysis in Appendix E.

5.1 Model predicates

We reuse the following adversary and clock drift model predicates defined in [21]:

- The π^{F} predicate ensures that all benign entities, i.e., not in T.F, follow the protocol correctly. T.F is the set of faulty entities outputted by the adversary. More precisely, the adversary can view the state, corrupt the state, or corrupt the output of the entities in T.F but not of any other entity.
- The $\pi^{\text{Drift}}_{\Delta_{clk}}$ predicate ensures that clock drifts from real time are bounded by Δ_{clk}. We presented it in Algorithm 1 (§3.1).

In addition, we reuse the standard communication model predicate $\pi^{\text{Com}}_{\Delta_{com}}$, also from [21], which ensures reliable communication between non-faulty parties, with delays bounded by Δ_{com}. The $\pi^{\text{Com}}_{\Delta_{com}}$ predicate in [21] ensures that if an operation outputs a ('send', m, j) triplet, then after at most Δ_{com}, entity j would receive m. Since the CT implementation in §D.1 has several request-response operations, we used a simpler notation for sending messages, and make the following small adjustment to the predicate:

(1) Whenever an entity ι includes in its output a triplet of the form (α-req,ι',x), where α-req is one of the request operations of CT, then, within Δ_{com}, there is an α-req event at ι' with input x.
(2) Whenever an entity ι' outputs a pair of the form (α-resp,y) in the output of an α-req operation, and this α-req was invoked by some entity ι, then, within Δ_{com}, there is an α-resp event at ι with input y.

The π^{Init} model predicate (Algorithm 22) ensures correct initialization of all entities at the beginning of the execution.

The CT$_{C,h}$ PKI requires periodic operations; to support that, we define the $\pi^{\text{WakeAt}}_{\Delta_{clk},\Delta_w}$ predicate shown in Algorithm 20. This model supports waking up at a specified local time (within Δ_w) and allows passing values to the wake-up event[13]; it is likely to be useful for the analysis of other protocols. According to the predicate, if ('WakeAt', t, $context$) is output, t is the current local time or later, and execution did not end too early, then there is a Wakeup event at the same entity at local time[14] at least t and at most $t + \Delta_w$, with input $context$.

Finally, the π^{1LogCert} model predicate ensures that honest monitors do not receive multiple different log certificates with the same log identifier as input to the 'Monitor' operation (Algorithm 21).

5.2 Analysis of Existential Unforgeability, Accountability and Accountable Revocation

We first show that the three PKIs satisfy Existential Unforgeability, Accountability and Accountable Δ_{Rev}-Revocation.

Theorem 3. *Let C be an existentially-unforgeable certificate scheme*[15]*. PKIX$_C$, CT$_{C,h}$ and CTwAudit$_{C,h}$ satisfy the following requirements:*

[13] The $\pi^{\text{Wake-up}}_{\Delta_{clk}}$ predicate in [21] supports waking up after a delay (within Δ_{clk}), which is not optimal for CT$_{C,h}$ as it could cause gradual drift from the correct period time. Also, it does not support passing values from the request to the wake-up event.

[14] The reason why we use the *local time* is to keep the model realistic; if we required the wake-ups to be at *real time* at least t and at most $t + \Delta_w$, then entities may be able to use wake-ups to determine something about the real time, which should not be possible.

[15] From Lemma 2, this is equivalent to the unforgeability of the underlying signature scheme.

- *Existential unforgeability, under the* $\pi^{\text{Init}} \wedge \pi^{\mathsf{F}}$ *model.*
- *Accountability, under the trivial (always true) model.*
- *Accountable* Δ_{Rev}*-Revocation, under the* $\pi^{\text{Init}} \wedge \pi^{\mathsf{F}} \wedge \pi^{Drift}_{\Delta_{clk}}$ *model, where* $\Delta_{\text{Rev}} \equiv \Delta_{clk} + \Delta_r$ *and* Δ_r *is the CRL validity period used in* PKIX_C.

Proof. The proof is identical for the three PKIs; for convenience, we refer to PKIX_C. We show that if each of the three requirements does not hold, then there is a PPT adversary $\mathcal{A}_C$ that can forge C-certificates with non-negligible probability, contradicting the assumption that C is an existentially-unforgeable certificate scheme.

Existential Unforgeability. Assume to the contrary that PKIX_C does not ensure existential unforgeability. By definition, this means that there exists a PPT adversary $\mathcal{A}_{\mathsf{EUF}}$ that satisfies:

$$\Pr\left[\begin{array}{c}\pi_{\mathsf{EUF}}(T) = \bot, \text{ where} \\ T \leftarrow \mathbf{Exec}_{\mathcal{A}_{\mathsf{EUF}},\mathsf{PKIX}_C}(params)\end{array}\right] \notin Negl(|params|) \quad (1)$$

From Equation (1), with non-negligible probability over the transcripts T of executions of PKIX_C with $\mathcal{A}_{\mathsf{EUF}}$, we have $\pi_{\mathsf{EUF}}(T) = \bot$. Following the implementation of π_{EUF} (Algorithm 2) and of PKIX_C.VCerts and PKIX_C.Valid (Algorithms 10 and 11), the adversary $\mathcal{A}_{\mathsf{EUF}}$ managed to generate a certificate ψ_0 which is valid for time t, root store *store* and auxiliary information *aux*, and yet: (1) $\psi_0.issuer$ is a benign entity, (2) the PKIX_C.Certify operation was not invoked at the benign issuer $\psi_0.issuer$ with the non-PKIadded fields of ψ_0 given as input, and (3) the validation of ψ_0 does not use a fraudulent certificate for the issuer $\psi_0.issuer$. In addition, from PKIX_C.Valid we know that $\psi_0.type$ ='PubKey', since ψ_0 is valid.

We first show that $\mathcal{A}_{\mathsf{EUF}}$ could not have generated ψ_0 by abusing PKIX_C's implementation and could not have used C.Certify (through PKIX_C) at $\psi_0.issuer$ to certify $tbc \leftarrow C.\text{Decode}(\psi_0)$. Then, we complete the proof by showing reduction to the security of C, i.e., the existence of $\mathcal{A}_{\mathsf{EUF}}$ means that C is not a secure certificate scheme.

According to the implementation of PKIX_C (Figure 1), the private key of an entity is generated using C.KeyGen in the PKIX_C.Init operation (Algorithm 7) and stored locally in the state st. Following π^{Init} (Algorithm 22), no operation is called before the Init operation has been called at an entity, and the Init operation is always called with the correct inputs. Since the MoSS execution process ensures correctness of the states of benign entities, then following Line 1 of PKIX_C.Init, an entity calls C.KeyGen from PKIX_C.Init at most once, the first time that PKIX_C.Init is called at the entity. Non-faulty entities output a self-signed certificate only for a public key generated by the entity using C.KeyGen in PKIX_C.Init, and never output the corresponding private key. Thus, the adversary did not have direct access to the private key, and therefore, could not use it directly to generate ψ_0.

Moreover, the only time the private key is accessed is when used by the entity to certify information using C.Certify in PKIX_C.Init, PKIX_C.Certify and PKIX_C.Aux. However, out of these three functions, the only function where a certificate with type 'PubKey' is certified is PKIX_C.Certify, and as mentioned earlier, following π_{EUF}, we know that PKIX_C.Certify was not invoked at the benign issuer $\psi_0.issuer$ with the non-PKIadded fields of ψ_0 given as input. Following the implementation of PKIX_C.Certify (Algorithm 8), this implies that $\mathcal{A}_{\mathsf{EUF}}$ could not have used C.Certify (through PKIX_C) at $\psi_0.issuer$ to certify $tbc \leftarrow C.\text{Decode}(\psi_0)$.

Consider an adversary $\mathcal{A}_C$ that receives as input a public key pk and oracle access to $C.\text{Certify}_{sk}(\cdot)$. $\mathcal{A}_C$ runs $\mathcal{A}_{\mathsf{EUF}}$ internally against PKIX_C, with the following changes: instead of benign entities calling C.KeyGen, $\mathcal{A}_C$ sets $(\text{st}.sk, \text{st}.pk) \leftarrow (\bot, pk)$, and whenever a benign entity calls C.Certify, $\mathcal{A}_C$ replaces this call with a call to the oracle. If $\mathcal{A}_{\mathsf{EUF}}$ succeeds with non-negligible probability in π_{EUF} against PKIX_C, then $\mathcal{A}_C$ also succeeds with non-negligible probability in the *ECF* game (Algorithm 6), which contradicts the assumption that C is a secure certificate scheme. □

Accountability. Assume to the contrary that PKIX_C does not ensure accountability. By definition, there exists a PPT adversary $\mathcal{A}_{\mathsf{ACC}}$ which satisfies:

$$\Pr\left[\begin{array}{c}\pi_{\mathsf{ACC}}(T) = \bot, \text{ where} \\ T \leftarrow \mathbf{Exec}_{\mathcal{A}_{\mathsf{ACC}},\mathsf{PKIX}_C}(params)\end{array}\right] \notin Negl(|params|) \quad (2)$$

From Equation (2), with non-negligible probability over the transcripts T of executions of PKIX_C with $\mathcal{A}_{\mathsf{ACC}}$, $\pi_{\mathsf{ACC}}(T) = \bot$. Following π_{ACC} (Algorithm 3) and the implementation of PKIX_C.VCerts and PKIX_C.Valid (Algorithms 10 and 11), the adversary $\mathcal{A}_{\mathsf{ACC}}$ managed to generate a certificate ψ_0 which is valid for time t, root store *store* and auxiliary information *aux* with type 'PubKey', and yet, the output of $\mathcal{P}$.VCerts for ψ_0, t, *store*, and *aux* includes no bad root certificate, no root certificate which is accountable for a valid bad certificate among the certificates in the output of $\mathcal{P}$.VCerts, and no root certificate which is accountable for ψ_0.

We first show that PKIX_C's implementation will not classify a certificate as valid unless there is a corresponding certificate chain, and then complete the proof by showing reduction to the security of C, i.e., the existence of $\mathcal{A}_{\mathsf{ACC}}$ means that C is not a secure certificate scheme.

Line 2 in PKIX_C.Valid requires that there exists a root certificate ψ_{root} in *store* such that ExistsValidChain$(\psi_0, t, \psi_{root}, aux)$ (Lines 4-18) is true. This means that there exists an acceptable chain ξ of certificates based on *aux* and *store* from ψ_0 to ψ_{root}. For such a chain $\xi = \psi_0 - \psi_1 - \ldots - \psi_r$, PKIX_C.Valid ensures the following: (1) Line 2 ensures that $\psi_r \in store$, (2) Lines 7 and 12 ensure that $(\forall i < r)\psi_i.issuer = \psi_{i+1}.subject$, (3) Lines 9 and 14 ensure that $(\forall i > 0)C.\text{Verify}_{\psi_i.pk}(\psi_{i-1}) = \top$, and (4) the implementation of ExistsValidChain ensures that if $\mathsf{PKIX}_C.\text{Valid}(\psi_0, t, \psi_{root}, aux) = \top$ then $(\forall i < r)\mathsf{PKIX}_C.\text{Valid}(\psi_i, t, store, aux) = \top$. Further, since PKIX_C.VCerts outputs the union of all certificates in *store* and *aux*, surely every certificate in ξ is in the set returned by PKIX_C.VCerts.

Consider the certificates in the chain $\xi = \psi_0 - \psi_1 - \ldots - \psi_r$ described above. All of the certificates in ξ are in the output of $\mathsf{PKIX}_C.\text{VCerts}(\psi_0, t, store, aux)$, all except for ψ_r are valid certificates, and ψ_r is in *store*. Since $\pi_{\mathsf{ACC}}(T) = \bot$, then ψ_r is not a bad certificate (i.e., $\psi_r.subject$ is a benign entity and $\psi_r.pk$ was self-certified by $\psi_r.subject$). We will show that for some certificate $\psi_i \in \{\psi_{r-1}, \ldots \psi_1\}$ such that $\psi_{i+1}.pk$ belongs to a benign entity, $\mathcal{A}_{\mathsf{ACC}}$ forged ψ_i such that ψ_i is valid w.r.t. $\psi_{i+1}.pk$. For contradiction, assume that this is not the case. Let ψ_j be the 'lowest' certificate in ξ before the first bad certificate, i.e., none of $\{\psi_r, \ldots \psi_j\}$ is a bad certificate. If none of the certificates in ξ is a bad certificate, then let $\psi_j = \psi_0$.

All of the certificates $\{\psi_{r-1}, \dots \psi_j\}$ are valid, none of them is a bad certificate (so $\forall \psi_i \in \{\psi_{r-1}, \dots \psi_j\}$, $\psi_i.subject$ is benign and $\psi_i.pk$ was self-certified by $\psi_i.subject$), and from PKIX_C.Valid, $(\forall i > 0)\mathcal{C}.\mathsf{Verify}_{\psi_i.pk}(\psi_{i-1}) = \top$. From the implementation of PKIX_C (Figure 1), $\mathcal{A}_{\mathsf{ACC}}$ could not have abused PKIX_C to obtain any valid certificate certified using a private key of a benign entity although the PKIX_C.Certify operation (Algorithm 8) was not used at the benign entity to certify the non-PKIadded fields of the certificate. Since none of the certificates $\{\psi_{r-1}, \dots \psi_j\}$ is a bad certificate, then for every certificate $\psi_i \in \{\psi_{r-1}, \dots \psi_1\}$, $\psi_{i+1}.pk$ belongs to a benign entity. We assumed that $\mathcal{A}_{\mathsf{ACC}}$ did not forge certificate $\psi_i \in \{\psi_{r-1}, \dots \psi_1\}$ such that $\psi_{i+1}.pk$ belongs to a benign entity such that ψ_i is valid w.r.t. $\psi_{i+1}.pk$.

If $j > 0$ and ψ_{j-1} is a valid bad certificate, then it follows that for all $i \geq j$, the PKIX_C.Certify operation was used at $\psi_i.subject$ to certify the non-PKIadded fields of ψ_{i-1}, which implies that ψ_r is accountable for ψ_{j-1}. Thus, ψ_r is accountable for a valid bad certificate from the output of PKIX_C.VCerts$(\psi_0, t, store, aux)$, which contradicts the assumption that $\pi_{\mathsf{ACC}}(T) = \bot$. If $j = 0$, then it follows that for all $i > 0$, the PKIX_C.Certify operation was used at $\psi_i.subject$ to certify the non-PKIadded fields of ψ_{i-1}, which implies that ψ_r is accountable for ψ_0. This also contradicts the assumption that $\pi_{\mathsf{ACC}}(T) = \bot$.

Therefore, for some valid certificate $\psi_i \in \{\psi_{r-1}, \dots \psi_1\}$ such that $\psi_{i+1}.pk$ belongs to a benign entity, $\mathcal{A}_{\mathsf{ACC}}$ forged ψ_i such that ψ_i is valid w.r.t. $\psi_{i+1}.pk$.

Following similar reasoning to the proof of Existential Unforgeability for PKIX_C, it follows that there exists an adversary $\mathcal{A}_C$ which runs $\mathcal{A}_{\mathsf{ACC}}$ internally against PKIX_C such that if $\mathcal{A}_{\mathsf{ACC}}$ succeeds with non-negligible probability in π_{ACC} against PKIX_C, then $\mathcal{A}_C$ also succeeds with non-negligible probability in the *ECF* game (Algorithm 6), which contradicts the assumption that C is a secure certificate scheme. □

Accountable Δ_{Rev}-Revocation. Let $\Delta_{\mathsf{Rev}} = \Delta_{clk} + \Delta_r$. Assume to the contrary that PKIX_C does not ensure Accountable Δ_{Rev}-Revocation. By definition, this means that there exists a PPT adversary $\mathcal{A}_{\mathsf{Rev}}$ that satisfies:

$$\Pr\left[\begin{array}{c} \pi^{AccRev}_{\Delta_{\mathsf{Rev}}}(T) = \bot, \text{ where} \\ T \leftarrow \mathbf{Exec}_{\mathcal{A}_{\mathsf{Rev}}, \mathsf{PKIX}_C}(params) \end{array}\right] \notin Negl(|params|) \quad (3)$$

From Equation (3) and following the π^{AccRev}_{Δ} predicate (in Definition 3), with non-negligible probability over the transcripts T of executions of PKIX_C with $\mathcal{A}_{\mathsf{Rev}}$, we have $f^{\mathsf{Rev}}_{\Delta_{\mathsf{Rev}}}(T) \notin \{\text{'G'}, \text{'A'}\}$, where the $f^{\mathsf{Rev}}_{\Delta}$ function is defined in Algorithm 4. Following the $f^{\mathsf{Rev}}_{\Delta}$ function, this means that the adversary $\mathcal{A}_{\mathsf{Rev}}$ can output values $(\psi, t, store, aux)$ such that PKIX_C.Valid$(\psi, t, store, aux) = \top$, even though ψ was certified by its benign issuer $\psi.issuer$ and was revoked at least Δ_{Rev} before t by $\psi.issuer$, and there is no fraudulent certificate that can be blamed for this discrepancy.

We first consider the validation of ψ by PKIX_C.Valid. We see that ψ is valid if there is some root certificate $\psi_{root} \in store$ such that there is a valid certificate chain from ψ to ψ_{root}. This is validated by ExistsValidChain, which searches aux for a chain $\xi \in aux$, where for some i holds $\xi[i] = \psi$. In Lines 6-17, ExistsValidChain performs several checks for $\xi[i]$ (and the rest of the relevant part of ξ). In particular, Line 13 checks that $t \in [\psi.from, \psi.to]$ and Line 17 calls CertIsNotRevoked to verify that aux also contains a valid CRL certificate ψ_{CRL} with $\psi_{CRL}.type =$ 'CRL', where ψ is not listed as revoked ($\psi.serial \notin \psi_{CRL}.CRL$). In Line 24, we check that the CRL has a valid certificate chain by calling recursively ExistsValidChain, which then confirms, in Line 13, that $t \in [\psi_{CRL}.from, \psi_{CRL}.to]$.

Following PKIX_C.Revoke (Algorithm 9), we next observe that once a certificate ψ is revoked at its benign issuer using operation PKIX_C.Revoke, then the serial number, $\psi.serial$, is added to st.CRL. There is no operation which removes entries from st.CRL until the entries are expired (i.e., if $\psi.to < clk$, then PKIX_C.Revoke removes ψ from st.CRL). Furthermore, in any future call to PKIX_C.Aux (Algorithm 12), if $clk \leq \psi.to$, then the contents of st.CRL will include the serial number $\psi.serial$ and will be certified in $\psi_{CRL}.CRL$ together with the current value of clk as $\psi_{CRL}.from$ (Line 4) and $clk + \Delta_r$ as $\psi_{CRL}.to$ (Line 5).

Let t_R be the (real) time when ψ was revoked at $\psi.issuer$. From the clock-drift model $\pi^{\mathrm{Drift}}_{\Delta_{clk}}$ (Algorithm 1), if PKIX_C.Aux was invoked before t_R to generate ψ_{CRL} which does not include $\psi.serial$ (since ψ was not revoked yet), then clk is less than $t_R + \Delta_{clk}$, so $\psi_{CRL}.to$ is less than $t_R + \Delta_{clk} + \Delta_r$. This implies that any such 'old' CRL certificate would be expired at time $t \geq t_R + \Delta_{clk} + \Delta_r$, i.e., $t \notin [\psi_{CRL}.from, \psi_{CRL}.to]$ so this ψ_{CRL} could not be used to validate ψ in PKIX_C.Valid.

On the other hand, if PKIX_C.Aux is invoked after time t_R to generate ψ_{CRL}, then there are two cases: either $clk \leq \psi.to$ or $clk > \psi.to$. In the first case, if $clk \leq \psi.to$, then ψ_{CRL} will include $\psi.serial$, i.e., $\psi.serial \in \psi_{CRL}.CRL$, so this ψ_{CRL} could not be used to validate ψ in PKIX_C.Valid. In the second case, if $clk > \psi.to$, then $\psi_{CRL}.from > \psi.to$. This implies that for any t, if t is in the validity period of ψ, i.e., $t \in [\psi.from, \psi.to]$, then t is less than $\psi_{CRL}.from$, so $t \notin [\psi_{CRL}.from, \psi_{CRL}.to]$. Therefore, this ψ_{CRL} also could not be used to validate ψ in PKIX_C.Valid.

Thus, there is no way to obtain a CRL certificate ψ_{CRL} from the benign entity $\psi.issuer$ using PKIX_C.Aux such that ψ_{CRL} could be used to validate ψ in PKIX_C.Valid w.r.t. a time $t \geq t_R + \Delta_{clk} + \Delta_r$. Yet, according to Lines 20-24 in PKIX_C.Valid, there is such a certificate ψ_{CRL} which furthermore has a valid certificate chain rooted at ψ_{root}. This leaves us with three possibilities: (1) the adversary got the benign entity $\psi.issuer$ to certify ψ_{CRL} with its private key in some operation other than PKIX_C.Aux; (2) ψ_{CRL} was not validated using a public key belonging to $\psi.issuer$; or (3) the adversary forged ψ_{CRL} such that it validates correctly using a public key belonging to $\psi.issuer$, even though the adversary did not know the private key of $\psi.issuer$. The first case is not possible, because according to the implementation of PKIX_C, certificates with type 'CRL' are only certified with the entity's private key in the PKIX_C.Aux operation. The second case would imply that PKIX_C.VCerts includes a fraudulent certificate, which is not the case according to $\pi^{AccRev}_{\Delta_{\mathsf{Rev}}}$ (see Line 4 in the $f^{\mathsf{Rev}}_{\Delta}$ function). Consequently, $\mathcal{A}_{\mathsf{Rev}}$ must have forged ψ_{CRL}.

It follows that there exists an adversary $\mathcal{A}_C$ which runs $\mathcal{A}_{\mathsf{Rev}}$ internally against PKIX_C such that if $\mathcal{A}_{\mathsf{Rev}}$ succeeds with non-negligible probability in $\pi^{AccRev}_{\Delta_{\mathsf{Rev}}}$ against PKIX_C, then $\mathcal{A}_C$ also

succeeds with non-negligible probability in the *ECF* game (Algorithm 6), which contradicts the assumption that C is a secure certificate scheme.

□

5.3 Analysis of Transparency

We now provide an overview of the main results of our analysis of $\mathsf{CT}_{C,h}$ and $\mathsf{CTwAudit}_{C,h}$. The proofs and details are in Appendix E.

5.3.1 $\mathsf{CT}_{C,h}$ *ensures HL* Δ_{Tra}*-Transparency.*

Theorem 4. *Let C be an existentially-unforgeable certificate scheme and h be a collision-resistant hash. Denote* $\Delta_{\mathsf{Tra}} \equiv 6\cdot\Delta_{clk}+2\cdot\Delta_{MMD}+2\cdot\Delta_{w}+5\cdot\Delta_{com}$*. Then,* $\mathsf{CT}_{C,h}$ *satisfies the HL* Δ_{Tra}*-Transparency requirement under model predicate:*

$$\pi^{Init} \wedge \pi^{\mathsf{F}} \wedge \pi^{1LogCert} \wedge \pi^{Drift}_{\Delta_{clk}} \wedge \pi^{Com}_{\Delta_{com}} \wedge \pi^{WakeAt}_{\Delta_{clk},\Delta_{w}} \quad (4)$$

Proof. See Appendix E.1. □

5.3.2 $\mathsf{CT}_{C,h}$ *does not ensure Guaranteed* Δ*-Transparency or Audited* Δ*-Transparency.*

Theorem 5. *Let C be a secure certificate scheme and h be a collision-resistant hash, and let Δ be any finite delay. Then,* $\mathsf{CT}_{C,h}$ does not *satisfy the Guaranteed* Δ*-Transparency requirement or the Audited* Δ*-Transparency requirement under the model predicate of Equation 4.*

Proof. See Appendix E.2. □

5.3.3 $\mathsf{CTwAudit}_{C,h}$ *ensures Audited* $\Delta^{AUD}_{\mathsf{Tra}}$*-Transparency.*

Theorem 6. *Let C be an existentially-unforgeable certificate scheme and h be a collision-resistant hash. Denote* $\Delta^{AUD}_{\mathsf{Tra}} \equiv 7\cdot\Delta_{clk}+2\cdot\Delta_{MMD}+2\cdot\Delta_{w}+5\cdot\Delta_{com}$*, where Δ_{MMD} is the maximal merge delay. Then,* $\mathsf{CTwAudit}_{C,h}$ *satisfies the Audited* $\Delta^{AUD}_{\mathsf{Tra}}$*-Transparency requirement under the model predicate of Equation 4.*

Proof. See Appendix E.3. □

6 CONCLUSIONS AND FUTURE WORK

PKI provides the foundation for the security of many deployed, critical distributed systems, in particular, the web and other applications using TLS, software signing, email security, and more. In spite of that, this work is the first to define security specifications for realistic PKI systems, supporting revocation, transitive trust (typically, certificate chains) and transparency, and considering realistic models allowing for corruptions, clock drift, delays and more.

A possible reason for this fundamental security infrastructure to remain without precise specifications and analysis may be that defining the requirements is tricky; they appear 'obvious' yet are hard to clearly define.

We applied our specifications to analyze the security of the two predominant PKI systems: PKIX and CT, as well as $\mathsf{CTwAudit}_{C,h}$, a variant of CT implemented in the Chrome browser. Our analysis exposed several subtle issues with these systems, related to revocation and transparency. Due to these issues, the systems satisfy only relaxed variants of revocation and transparency.

This work makes only the first steps toward provable security of PKI schemes. Much work remains, including design and analysis of PKIs that will meet the stronger revocation and transparency requirements; formally defining other PKI schemes for the entire ecosystem (including browser specific implementations) and applying our specifications to such schemes; extending our specifications for non-certificate PKIs (e.g., [39]) or to additional properties (e.g., privacy, non-equivocation); and showing similar specifications and proofs under UC [8] or another framework allowing compositions of protocols. We also hope that similar specifications and analysis can be applied to other applied cryptographic protocols, e.g., blockchains.

REFERENCES

[1] European Telecommunications Standards Institute (ETSI). [n. d.] https://www.etsi.org/. ().

[2] Hadi Asghari, Michel Van Eeten, Axel Arnbak, and Nico ANM van Eijk. 2013. Security Economics in the HTTPS Value Chain. In *Twelfth Workshop on the Economics of Information Security (WEIS 2013), Washington, DC.*

[3] Louise Axon and Michael Goldsmith. 2017. PB-PKI: A Privacy-aware Blockchain-based PKI. In *SECRYPT.*

[4] David Basin, Cas Cremers, Tiffany Hyun-Jin Kim, Adrian Perrig, Ralf Sasse, and Pawel Szalachowski. 2014. ARPKI: Attack Resilient Public-Key Infrastructure. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security.* ACM, 382–393.

[5] Alexandra Boldyreva, Marc Fischlin, Adriana Palacio, and Bogdan Warinschi. 2007. A Closer Look at PKI: Security and Efficiency. In *International Workshop on Public Key Cryptography.* Springer, 458–475.

[6] Johannes Braun. 2015. *Maintaining Security and Trust in Large Scale Public Key Infrastructures.* Ph.D. Dissertation. Technische Universität.

[7] Johannes Braun, Franziskus Kiefer, and Andreas Hülsing. 2013. Revocation & Non-Repudiation: When the first destroys the latter. In *European Public Key Infrastructure Workshop.* Springer, 31–46.

[8] Ran Canetti. 2020. Universally composable security. *Journal of the ACM (JACM)*, 67, 5, 1–94.

[9] Ran Canetti, Daniel Shahaf, and Margarita Vald. 2016. Universally Composable Authentication and Key-exchange with Global PKI. In *Public-Key Cryptography–PKC 2016.* Springer, 265–296.

[10] BLUE BOOK CCITT. 1988. Recommendations X. 509 and ISO 9594-8. Information Processing Systems-OSI-The Directory Authentication Framework (Geneva: CCITT). (1988).

[11] Melissa Chase and Sarah Meiklejohn. 2016. Transparency Overlays and Applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security.* ACM, 168–179.

[12] Council of European Union. 2021. Com/2021/281, Revision of the eIDAS Regulation - European Digital Identity (EUid). (2021).

[13] Benjamin Dowling, Felix Günther, Udyani Herath, and Douglas Stebila. 2016. Secure Logging Schemes and Certificate Transparency. In *European Symposium on Research in Computer Security.* Springer, 140–158.

[14] John Dyer. 2015. China Accused of Doling Out Counterfeit Digital Certificates in 'Serious' Web Security Breach. VICE News. (Apr. 2015).

[15] Peter Eckersley. 2012. Sovereign Key Cryptography for Internet Domains. https://git.eff.org/?p=sovereign-keys.git;a=blob;f=sovereign-key-design.txt;hb=HEAD. (2012).

[16] Electronic Frontier Foundation (EFF). [n. d.] The EFF SSL Observatory. Retrieved May 30, 2019 from https://www.eff.org/observatory.

[17] Conner Fromknecht, Dragos Velicanu, and Sophia Yakoubov. 2014. A Decentralized Public Key Infrastructure with Identity Retention. *IACR Cryptology ePrint Archive*, 2014, 803.

[18] Sebastian Gajek, Mark Manulis, Olivier Pereira, Ahmad-Reza Sadeghi, and Jörg Schwenk. 2008. Universally Composable Security Analysis of TLS. In *International Conference on Provable Security.* Springer, 313–327.

[19] Shafi Goldwasser and Silvio Micali. 1984. Probabilistic Encryption. *Journal of Computer and System Sciences*, 28, 2, 270–299.

[20] Shafi Goldwasser, Silvio Micali, and Ronald L Rivest. 1988. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on computing*, 17, 2, 281–308.

[21] Amir Herzberg, Hemi Leibowitz, Ewa Syta, and Sara Wrótniak. 2021. MoSS: Modular Security Specifications framework. In *CRYPTO' 2021.* Full version at: https://eprint.iacr.org/2020/1040, 33–63.

[22] Amir Herzberg, Yosi Mass, Joris Mihaeli, Dalit Naor, and Yiftach Ravid. 2000. Access Control Meets Public Key Infrastructure, Or: Assigning Roles to Strangers. In *Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000.* IEEE, 2–14.

[23] P. Hoffman, M. Blanchet, E. Lafon, Y. Galand, C. Elphick, and M. Moeller. 2002. International organization for standardization, information technology - asn.1 - basic encoding rules (ber). In *ITU-T Recommendation X.690 | ISO/IEC 8825-1:2002.* Also covers DER, 1–105.

[24] Jacob Hoffman-Andrews. 2023. Article 45 Will Roll Back Web Security by 12 Years. ACLU. https://www.eff.org/deeplinks/2023/11/article-45-will-roll-back-web-security-12-years. (Nov. 2023).

[25] Joel Hruska. 2015. Apple, Microsoft buck trend, refuse to block unauthorized Chinese root certificates. ExtremeTech. (Apr. 2015).

[26] Jingwei Huang and David M Nicol. 2017. An anatomy of trust in public key infrastructure. *International Journal of Critical Infrastructures*, 13, 2-3, 238–258.

[27] Google LLC Joe DeBlasio. [n. d.] Opt-out SCT Auditing in Chrome. Other. https://docs.google.com/document/d/16G-Q7iN3kB46GSW5b-sfH5MO3nKSYyEb77YsM7TMZGE/. ().

[28] Tiffany Hyun-Jin Kim, Lin-Shung Huang, Adrian Perrig, Collin Jackson, and Virgil Gligor. 2013. Accountable Key Infrastructure (AKI): A Proposal for a Public-Key Validation Infrastructure. In *Proceedings of the 22nd international conference on World Wide Web*. ACM, 679–690.

[29] Loren M Kohnfelder. 1978. *Towards a practical public-key cryptosystem*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[30] Murat Yasin Kubilay, Mehmet Sabir Kiraz, and Haci Ali Mantar. 2018. CertLedger: A new PKI model with Certificate Transparency based on blockchain. *arXiv preprint arXiv:1806.03914*.

[31] Ben Laurie. 2014. Certificate transparency. *Communications of the ACM*, 57, 10, 40–46.

[32] Ben Laurie and Emilia Kasper. 2012. Revocation Transparency. *Google Research, September*.

[33] Dimitrios Lekkas. 2003. Establishing and managing trust within the Public Key Infrastructure. *Computer Communications*, 26, 16, 1815–1825.

[34] Bingyu Li, Jingqiang Lin, Fengjun Li, Qiongxiao Wang, Qi Li, Jiwu Jing, and Congli Wang. 2019. Certificate transparency in the wild: exploring the reliability of monitors. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2505–2520.

[35] John Marchesini and Sean Smith. 2005. Modeling Public Key Infrastructure in the Real World. In *European Public Key Infrastructure Workshop*. Springer, 118–134.

[36] Stephanos Matsumoto and Raphael M Reischuk. 2017. IKP: Turning a PKI Around with Decentralized Automated Incentives. In *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 410–426.

[37] Ueli Maurer. 1996. Modelling a Public-Key Infrastructure. In *European Symposium on Research in Computer Security*. Springer, 325–350.

[38] Marcela S Melara, Aaron Blankstein, Joseph Bonneau, Edward W Felten, and Michael J Freedman. 2015. CONIKS: Bringing Key Transparency to End Users. In *USENIX Security Symposium*, 383–398.

[39] [n. d.] Namecoin. (). https://www.namecoin.org/.

[40] Scientists Organisations and Researchers as signed. 2023. Joint statement of scientists and NGOs on the EU's proposed eIDAS reform. Other. https://nce.mpi-sp.org/index.php/s/cG88cptFdaDNyRr. (Nov. 2023).

[41] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. 1999. SPKI Certificate Theory. RFC 2693 (Experimental). RFC. Fremont, CA, USA: RFC Editor, (Sept. 1999). DOI: 10.17487/RFC2693.

[42] R. Housley, W. Polk, W. Ford, and D. Solo. 2002. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 3280 (Proposed Standard). RFC. Obsoleted by RFC 5280, updated by RFCs 4325, 4630. Fremont, CA, USA: RFC Editor, (Apr. 2002). DOI: 10.17487/RFC3280.

[43] J. Jonsson and B. Kaliski. 2003. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. RFC 3447 (Informational). RFC. Obsoleted by RFC 8017. Fremont, CA, USA: RFC Editor, (Feb. 2003). DOI: 10.17487/RFC3447.

[44] M. Lepinski and S. Kent. 2008. Additional Diffie-Hellman Groups for Use with IETF Standards. RFC 5114 (Informational). RFC. Fremont, CA, USA: RFC Editor, (Jan. 2008). DOI: 10.17487/RFC5114.

[45] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. 2008. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280 (Proposed Standard). RFC. Updated by RFCs 6818, 8398, 8399. Fremont, CA, USA: RFC Editor, (May 2008). DOI: 10.17487/RFC5280.

[46] M. Lepinski and S. Kent. 2012. An Infrastructure to Support Secure Internet Routing. RFC 6480 (Informational). RFC. Fremont, CA, USA: RFC Editor, (Feb. 2012). DOI: 10.17487/RFC6480.

[47] B. Laurie, A. Langley, and E. Kasper. 2013. Certificate Transparency. RFC 6962 (Experimental). RFC. Obsoleted by RFC 9162. Fremont, CA, USA: RFC Editor, (June 2013). DOI: 10.17487/RFC6962.

[48] S. Josefsson and S. Leonard. 2015. Textual Encodings of PKIX, PKCS, and CMS Structures. RFC 7468 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, (Apr. 2015). DOI: 10.17487/RFC7468.

[49] E. Rescorla. 2018. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, (Aug. 2018). DOI: 10.17487/RFC8446.

[50] B. Laurie, E. Messeri, and R. Stradling. 2021. Certificate Transparency Version 2.0. RFC 9162 (Experimental). RFC. Fremont, CA, USA: RFC Editor, (Dec. 2021). DOI: 10.17487/RFC9162.

[51] Steven B Roosa and Stephen Schultze. 2010. The "Certificate Authority" Trust Model for SSL: A Defective Foundation for Encrypted Web Traffic and a Legal Quagmire. *Intellectual property & technology law journal*, 22, 11, 3.

[52] Mark Dermot Ryan. 2014. Enhanced certificate transparency and end-to-end encrypted mail. In *NDSS*.

[53] Wazan Ahmad Samer, Laborde Romain, Barrere Francois, and Benzekri AbdelMalek. 2011. A formal model of trust for calculating the quality of X. 509 certificate. *Security and Communication Networks*, 4, 6, 651–665.

[54] Nicolas Serrano, Hilda Hadan, and Jean L. Camp. 2019. A complete study of P.K.I. (PKI's Known Incidents). Available at SSRN, https://ssrn.com/abstract=3425554. (July 2019).

[55] Ewa Syta, Iulia Tamas, Dylan Visher, David Isaac Wolinsky, and Bryan Ford. 2015. Certificate Cothority: Towards Trustworthy Collective CAs. *Hot Topics in Privacy Enhancing Technologies (HotPETs)*, 7.

[56] Ewa Syta, Iulia Tamas, Dylan Visher, David Isaac Wolinsky, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ismail Khoffi, and Bryan Ford. 2016. Keeping Authorities "Honest or Bust" with Decentralized Witness Cosigning. In *Security and Privacy (SP), 2016 IEEE Symposium on*. Ieee, 526–545.

[57] Pawel Szalachowski, Stephanos Matsumoto, and Adrian Perrig. 2014. PoliCert: Secure and Flexible TLS Certificate Management. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 406–417.

[58] Alin Tomescu and Srinivas Devadas. 2017. Catena: Efficient Non-equivocation via Bitcoin. In *2017 38th IEEE Symposium on Security and Privacy (SP)*. IEEE, 393–409.

[59] Dan Wendlandt, David G Andersen, and Adrian Perrig. 2008. Perspectives: Improving SSH-style Host Authentication with Multi-Path Probing. In *USENIX Annual Technical Conference*. Vol. 8, 321–334.

[60] Wikipedia contributors. 2021. Diginotar — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=DigiNotar&oldid=1036090956. [Online; accessed 7-August-2021]. (2021).

[61] Jiangshan Yu, Vincent Cheval, and Mark Ryan. 2016. DTKI: A New Formalized PKI with Verifiable Trusted Parties. *The Computer Journal*, 59, 11, 1695–1713.

[62] Michelle Zhou, Prithvi Bisht, and VN Venkatakrishnan. 2011. Strengthening XSRF Defenses for Legacy Web Applications Using Whitebox Analysis and Transformation. In *Information Systems Security*. Springer, 96–110.

Algorithm 7 PKIX_C.Init: initialization

Input: local state st, local clock clk and parameters $params$, including identifier $params.\iota$, security parameter $params.global.1^\kappa$, and acceptable revocation delay $params.global.\Delta_r$.
Output: updated state and self-signed certificate ψ.
procedure PKIX_C.Init (st, clk, $params$)

1: **if** st $\neq \perp$ **then return** (st, $\perp$) ▷ *Do not initialize non-empty state*
2: $\mathsf{st}.\Delta_r \leftarrow params.global.\Delta_r$ ▷ *Save CRL interval* Δ_r
3: $\mathsf{st}.\iota \leftarrow params.\iota$ ▷ *Save identity* (ι)
4: $(\mathsf{st}.sk, \mathsf{st}.pk) \leftarrow C.\mathsf{KeyGen}(params.global.1^\kappa)$ ▷ *Generate signature key pair*
5: $tbc \leftarrow \left\{ \begin{array}{c} (\text{'pk'}, \mathsf{st}.pk),\ (\text{'type'}, \textit{'SelfCert'}), \\ (\text{'issuer'}, \mathsf{st}.\iota), (\text{'subject'}, \mathsf{st}.\iota) \end{array} \right\}$
6: $\psi \leftarrow C.\mathsf{Certify}_{\mathsf{st}.sk}(tbc)$ ▷ *Certify a self-signed certificate*
7: $(\mathsf{st}.certs, \mathsf{st}.CRL) \leftarrow \perp$ ▷ *Init both sets as empty*
8: **return** (st, ('SelfCert', ψ))
end procedure

Algorithm 8 PKIX_C.Certify: certificate issuance

Input: local state st, local clock clk and data to be certified tbc.
Output: updated state and and certificate ψ.
procedure PKIX_C.Certify (st, clk, tbc)

1: **if** $tbc.subject = \mathsf{st}.\iota$ **then return** (st, $\perp$) ▷ *Self-signed cert only from 'Init'!*
2: **if** $tbc.type \in$ {'SelfCert', 'CRL'} **then return** (st, $\perp$) ▷ *Type cannot be 'SelfCert' or 'CRL'!*
3: **if** $tbc.serial = \perp$ **then return** (st, $\perp$) ▷ *Serial number must be nonempty!*
4: $tbc.issuer \leftarrow \mathsf{st}.\iota$ ▷ *Set issuer field*
5: **if** $tbc.type = \perp$ **then** $tbc.type \leftarrow$ *'PubKey'* ▷ *Default type is 'PubKey'*
6: $tbc.PKIadded$ += {'issuer', 'type', 'PKIadded'} ▷ *Set PKIadded field*
7: $\psi \leftarrow C.\mathsf{Certify}_{\mathsf{st}.sk}(tbc)$ ▷ *Issue certificate*
8: $\mathsf{st}.certs$ += ψ ▷ *Store locally*
9: **return** (st, ψ)
end procedure

Algorithm 9 PKIX_C.Revoke: certificate revocation

Input: local state st, local clock clk and certificate ψ.
Output: updated state, $\top$ if revocation completed successfully, and $\perp$ otherwise.
procedure PKIX_C.Revoke (st, clk, ψ)

1: **if** $\psi \in \mathsf{st}.certs \wedge \psi.to \geq clk$ **then** ▷ *ψ was issued by this CA and is not expired*
2: $\mathsf{st}.CRL$ += $\psi.serial$ ▷ *Add to local CRL*
3: **return** (st, $\top$) ▷ *ψ revoked successfully*
4: **end if**
5: **return** (st, $\perp$) ▷ *Failure*
end procedure

Algorithm 10 PKIX_C.VCerts: output certificates used for validation

Input: certificate ψ, time t, root store *store* and auxiliary information *aux*.
Output: a set of certificates $\{\psi_i\}$.
procedure PKIX_C.VCerts(ψ, t, *store*, *aux*)

1: **return** $store \cup \{\psi' \mid \exists \xi \in aux \textbf{ s.t. } \psi' \in \xi\}$
end procedure

Algorithm 11 PKIX_C.Valid: certificate validation

Input: certificate ψ, time t, root store *store* and auxiliary information *aux*.
Output: $\top$ if ψ is a valid certificate, and $\perp$ otherwise.
procedure PKIX_C.Valid(ψ, t, *store*, *aux*)

1: **if** $\psi.type \neq$ *'PubKey'* **then return** $\perp$ ▷ *Non-'PubKey' invalid*
2: **if** $\exists \psi_{root} \in store$ **s.t.** ExistsValidChain$(\psi, t, \psi_{root}, aux)$ **then return** $\top$ ▷ *Valid chain*
3: **return** $\perp$ ▷ *Otherwise, ψ is invalid*
end procedure
4: **procedure** ExistsValidChain(ψ', t, ψ_{root}, *aux*)
5: **return** $\exists \xi \in aux, i \leq j \leq |\xi|$ **s.t.** ▷ *Part of a chain in aux*
6: $\xi[i] = \psi' \wedge$ ▷ *Starts with ψ'*
7: $\xi[j].issuer = \psi_{root}.subject \wedge$ ▷ *End issuer is root subject*
8: $t \in [\xi[j].from, \xi[j].to] \wedge$ ▷ *End cert is not expired*
9: $C.\mathsf{Verify}_{\psi_{root}.pk}(\xi[j]) \wedge$ ▷ *End verified by root pk*
10: CertIsNotRevoked$(\xi[j], t, \psi_{root}, aux) \wedge$ ▷ *Cert $\xi[j]$ is not revoked*
11: $\forall i \leq k < j$: ▷ *For all previous certs*
12: $\xi[k].issuer = \xi[k+1].subject \wedge$ ▷ *Issuer is next subject*
13: $t \in [\xi[k].from, \xi[k].to] \wedge$ ▷ *Cert $\xi[k]$ is not expired*
14: $C.\mathsf{Verify}_{\xi[k+1].pk}(\xi[k]) \wedge$ ▷ *Verified by next cert pk*
15: $\xi[k+1].type =$ *'PubKey'* $\wedge$ ▷ *Next is a 'PubKey' cert*
16: $\xi[k+1].is\_CA \wedge$ ▷ *Next cert is for a CA*
17: CertIsNotRevoked$(\xi[k], t, \psi_{root}, aux)$ ▷ *Cert $\xi[k]$ is not revoked*
18: **end procedure**
19: **procedure** CertIsNotRevoked(ψ'', t, ψ_{root}, *aux*)
20: **return** $\exists \xi_{CRL} \in aux, \psi_{CRL} \in \xi_{CRL}$ **s.t.** ▷ *There is a cert in aux*
21: $\psi_{CRL}.type =$ 'CRL' $\wedge$ ▷ *Which is a CRL*
22: $\psi_{CRL}.issuer = \psi''.issuer \wedge$ ▷ *Issued by $\psi''.issuer$*
23: $\psi''.serial \notin \psi_{CRL}.CRL \wedge$ ▷ *ψ'' is not in the CRL*
24: ExistsValidChain$(\psi_{CRL}, t, \psi_{root}, aux)$ ▷ *With valid chain to root*
25: **end procedure**

Algorithm 12 PKIX_C.Aux: CRL generation

Input: local state st, certificate ψ, local clock clk, root store *store* and auxiliary information *aux*.
Output: updated state and current CRL.
procedure PKIX_C.Aux(st, ψ, clk, *store*, *aux*)

1: $\mathsf{st}.CRL$ −= $\left\{ \begin{array}{c} \psi.serial \in \mathsf{st}.CRL \\ \textbf{s.t. } \psi.to < clk \end{array} \right\}$ ▷ *Remove expired certificates from CRL*
2: $tbc \leftarrow \{(\text{'type'}, \text{'CRL'})\}$ ▷ *Certificate of type CRL*
3: $tbc.issuer \leftarrow \mathsf{st}.\iota$ ▷ *The issuer of the CRL*
4: $tbc.from \leftarrow clk$ ▷ *When was issued*
5: $tbc.to \leftarrow clk + \mathsf{st}.\Delta_r$ ▷ *Valid until*
6: $tbc.CRL \leftarrow \mathsf{st}.CRL$ ▷ *CRL information*
7: $\psi_{CRL} \leftarrow C.\mathsf{Certify}_{\mathsf{st}.sk}(tbc)$ ▷ *Issue certificate*
8: **return** (st, ψ_{CRL})
end procedure

Figure 1: PKIX (X.509 version 3 with CRL) implementation

A PKIX$_C$ IMPLEMENTATION

Figure 1 presents the PKIX$_C$ PKI, as a simplified pseudocode based on the specifications [45].

B ADDITIONAL PREDICATE PROCEDURES

This appendix provides specifications of procedures used in the Δ_{Rev}-Revocation function (Algorithm 4) and the Tra-Transparency function (Algorithm 5) predicates.

Algorithm 13 Ensure $\mathcal{P}$.Certify and $\mathcal{P}$.Revoke requested predicate

procedure CertifyAndRevokeRequested($T, \psi, t, \Delta_{\text{Rev}}$)
  return $\exists e, e_R \; s.t. \; e < e_R \wedge$ ▷ ψ issued, then revoked
    $T.ent[e] = T.ent[e_R] = \psi.issuer \wedge$ ▷ by $\psi.issuer$
    $T.opr[e] =$ 'Certify' $\wedge$ ▷ e was a Certify operation
    $T.opr[e_R] =$ 'Revoke' $\wedge$ ▷ e_R was a Revoke operation
    $T.out[e] = \psi \wedge$ ▷ ψ issued by $\psi.issuer$
    $T.inp[e_R] = \psi \wedge$ ▷ ψ revoked by $\psi.issuer$
    $T.\tau[e] \leq T.\tau[e_R] \leq t - \Delta_{\text{Rev}}$ ▷ both at least Δ_{Rev} before t
end procedure

Algorithm 14 Ensure fraudulent certificate can be found predicate

procedure CanFindFraudulentCert($T, \psi, t, store, aux$)
  $vcerts \leftarrow \mathcal{P}$.VCerts($\psi, t, store, aux$) ▷ Certificates used for validating ψ
  return $\exists \psi' \in vcerts$ **s.t.** ▷ Contain a certificate ψ' such that
    $\psi' \neq \psi \wedge$ ▷ ψ' is different than ψ
    $\left[\begin{array}{l} \mathcal{P}.\text{Valid}(\psi', t, store, aux) \vee \\ \left[\begin{array}{l} \psi' \in store \wedge \\ \psi'.subject = \psi'.issuer \end{array}\right] \end{array}\right] \wedge$ ▷ ψ' is a valid certificate, or ▷ ψ' is in store and ▷ has same subject and issuer
    $\psi'.subject \in T.\text{N} - T.\text{F} \wedge$ ▷ The subject of ψ_1 is benign
    $\neg$SelfCert($T, \psi'.subject, \psi'.pk$) ▷ ψ' is fraudulent (Alg. 2)
end procedure

Algorithm 15 There is no subset of *aux.logs* which includes an honest log which is needed to validate ψ using $\mathcal{P}$.Valid

procedure NoHonestLog($\psi, t, store, aux$)
  return $\nexists logs_\psi \subseteq aux.logs, id_{\text{HL}} \in logs_\psi$ **s.t.** ▷ There is a subset $logs_\psi$ of aux.logs and a log id id_{HL} in this subset
    $\left[\begin{array}{l} \mathcal{P}.\text{Valid}(\psi, t, store, aux') \wedge \\ \quad \text{where} \left\{\begin{array}{l} aux' \leftarrow aux; \\ aux'.logs \leftarrow logs_\psi \end{array}\right\} \end{array}\right] \wedge$ ▷ ψ is valid w.r.t. logs $logs_\psi$
    $\left[\begin{array}{l} \neg\mathcal{P}.\text{Valid}(\psi, t, store, aux'') \\ \quad \text{where} \left\{\begin{array}{l} aux'' \leftarrow aux; \\ aux''.logs \leftarrow logs_\psi - id_{\text{HL}} \end{array}\right\} \end{array}\right] \wedge$ ▷ ψ is not valid w.r.t. logs $logs_\psi - id_{\text{HL}}$
    $\left[\begin{array}{l} \exists e \text{ s.t. } T.ent[e] \in T.\text{N} - T.\text{F} \wedge \\ \quad (\text{'LogCert'}, store.logs[id_{\text{HL}}]) \in T.out[e] \end{array}\right]$ ▷ The log cert from store.logs for log id_{HL} is a benign entity's log cert
end procedure

Algorithm 16 Ensure certificate logs are monitored by entity

procedure MonitoringCertLogs($T, \psi, t', store, aux, \iota_M$)
  return $\forall log\_id \in aux.logs:$ ▷ for each log in aux.logs
    $\exists e$ s.t.$T.opr[e] =$ 'Monitor' $\wedge$ ▷ 'Monitor' operation was invoked
    $T.ent[e] = \iota_M \wedge$ ▷ at entity ι_M
    $T.inp[e] = store.logs[log\_id] \wedge$ ▷ to monitor this log
    $T.\tau[e] \leq t'$ ▷ at real time t' or earlier
end procedure

Algorithm 17 Monitor is unaware of logged fields of certificate

procedure MonitorIsUnaware(T, ψ, t, ι_M)
  return $\exists e$ **s.t.** $T.opr[e] =$ *'Lookup'* $\wedge$ ▷ *'Lookup' operation was invoked*
    $T.ent[e] = \iota_M \wedge$ ▷ *at entity ι_M*
    $T.inp[e] = \psi.subject \wedge$ ▷ *w.r.t $\psi.subject$*
    $T.\tau[e] > t \wedge$ ▷ *sometime after t*
    $\nexists \psi' \in T.out[e]$ **s.t.** ▷ *but ι_M was unaware of any ψ' s.t.*
      NonAdded(ψ') = NonAdded(ψ) ▷ *$\psi'.tbc$ and $\psi.tbc$ have the same non-PKIadded fields and values*
end procedure
procedure NonAdded(ψ)
  return $\left\{ (field, x) \;\middle|\; \begin{array}{l} (field, x) \in \psi.tbc \wedge \\ field \notin \psi.PKIadded \end{array} \right\}$ ▷ *Return set of all non-PKIadded fields and values in $\psi.tbc$*
end procedure

Algorithm 18 Monitor is audited

procedure Audited(T, ψ, t, aux, ι_M)
  return $\exists e$ **s.t.**
    $\left[\begin{array}{ll} T.opr[e] = \text{'Audit'} \wedge & \triangleright \text{ 'Audit' invoked} \\ T.ent[e] = \iota_M \wedge & \triangleright \text{ at entity } \iota_M \\ T.inp[e] = (\psi, aux) \wedge & \triangleright \text{ w.r.t } \psi \text{ and } aux \\ T.\tau[e] > t & \triangleright \text{ sometime after } t \end{array} \right]$
end procedure

Algorithm 19 Monitor fails to correctly identify corrupted log cert

procedure FailedIdentifyCorrupted($T, \psi, t, store, aux, \iota_M$)
  return $\exists e$ **s.t.**
    $\left[\begin{array}{ll} T.opr[e] = \text{'Audit'} \wedge & \triangleright \text{ 'Audit' invoked} \\ T.ent[e] = \iota_M \wedge & \triangleright \text{ at entity } \iota_M \\ T.inp[e] = (\psi, aux) \wedge & \triangleright \text{ w.r.t } \psi \text{ and } aux \\ T.\tau[e] > t & \triangleright \text{ sometime after } t \\ \left[\begin{array}{ll} \left[\nexists(\text{'Corrupt'}, \psi_L) \in T.out[e] \text{ s.t.} \right. & \triangleright \text{ did not identify} \\ \quad \left. \text{CorruptedLog}(T, \psi_L, store, aux) \right] \vee & \triangleright \text{ corrupt log cert} \\ \left[\exists(\text{'Corrupt'}, \psi_L) \in T.out[e] \text{ s.t.} \right. & \triangleright \text{ or identified} \\ \quad \left. \neg\text{CorruptedLog}(T, \psi_L, store, aux) \right] & \triangleright \text{ incorrect log cert} \end{array} \right] \end{array} \right]$
end procedure
procedure CorruptedLog($T, \psi_L, store, aux$)
  return $\exists log\_id \in aux.logs$ **s.t.** $\psi_L = store.logs[log\_id] \wedge$ ▷ *ψ_L is a log cert from aux.logs and store.logs*
    $\nexists e'$ **s.t.** $\left[\begin{array}{ll} T.ent[e'] \in T.\mathsf{N} - T.\mathsf{F} \wedge & \triangleright \text{ no benign entity} \\ (\text{'LogCert'}, \psi_L) \in T.out[e'] & \triangleright \text{ outputted } \psi_L \text{ as its log cert} \end{array} \right]$
end procedure

Algorithm 20 $\pi^{\text{WakeAt}}_{\Delta_{clk}, \Delta_w}$ model predicate

return $\forall \hat{e} \in \{1, \ldots, T.e\}$: ▷ *For each event $\hat{e}$*
  if $\left[\begin{array}{ll} (\text{'WakeAt'}, t, context) \in T.out[\hat{e}] \wedge & \triangleright \text{ If the output includes a 'WakeAt' triplet, and} \\ t \geq T.clk[\hat{e}] \wedge & \triangleright \text{ the requested time is the local time or later, and} \\ T.\tau[T.e] \geq t + \Delta_{clk} + \Delta_w & \triangleright \text{ execution did not end before last allowed wakeup time} \end{array} \right]$
  then $\exists \hat{e}' \in \{\hat{e}+1, \ldots, T.e\}$ **s.t.** ▷ *There is a later event*
    $\left[\begin{array}{ll} t \leq T.clk[\hat{e}'] \leq t + \Delta_w & \triangleright \text{ At the correct local time} \\ T.ent[\hat{e}'] = T.ent[\hat{e}] \wedge & \triangleright \text{ At the same entity} \\ T.opr[\hat{e}'] = \text{'Wakeup'} \wedge & \triangleright \text{ With operation 'Wakeup'} \\ T.inp[\hat{e}'] = context & \triangleright \text{ With input } context \end{array} \right]$
  end if

Algorithm 21 The π^{1LogCert} model predicate

return $\nexists \hat{e}, \hat{e}' \in \{1, \ldots, T.e\}$ **s.t.** ▷ *There are not two events*
    $\hat{e} \neq \hat{e}' \wedge$ ▷ *Which are different*
    $T.ent[\hat{e}] = T.ent[\hat{e}'] \in T.\mathsf{N} - T.\mathsf{F} \wedge$ ▷ *Both at the same benign entity*
    $T.opr[\hat{e}] = T.opr[\hat{e}'] =$ 'Monitor' $\wedge$ ▷ *Asking to monitor a log*
    $T.inp[\hat{e}].log\_id = T.inp[\hat{e}'].log\_id$ ▷ *With input log certificates with the same log identifier*
    $T.inp[\hat{e}] \neq T.inp[\hat{e}']$ ▷ *But the log certificates are different*

Algorithm 22 The π^{Init} model predicate

return $\left[\begin{array}{l} \forall \hat{e} \in \{1, \ldots, T.e\} : \exists \hat{e}' \leq \hat{e} \text{ s.t.} \\ \quad T.ent[\hat{e}'] = T.ent[\hat{e}] \wedge \\ \quad T.opr[\hat{e}'] = \text{'Init'} \end{array} \right] \wedge$ ▷ *There must be an 'Init' event before any other event at the entity*
  $\forall \hat{e}$ **s.t.** $T.opr[\hat{e}] =$ 'Init' : ▷ *In every 'Init' event*
    $|T.inp[\hat{e}]| \geq |T.params| \wedge$ ▷ *Input size is at least $|T.params|$*
    $T.inp[\hat{e}].params.\iota = T.ent[\hat{e}] \wedge$ ▷ *Correct entity identifier parameter is given*
    $T.inp[\hat{e}].params.global = T.params.global$ ▷ *Global parameters are $T.params.global$*

C MODEL PREDICATES

This appendix provides model predicates used in the security analysis.

D CERTIFICATE TRANSPARENCY IMPLEMENTATIONS

This appendix provides our implementation of two versions of Certificate Transparency. The first version, CT, presented in Appendix D.1, follows the CT 2.0 specification [50]. The second version CTwAudit, presented in Appendix D.2, is the CT augmented by an auditing mechanism [27] and available as an option in Google Chrome.

D.1 Certificate Transparency

Certificate Transparency (CT) [47, 50] extends PKIX to ensure transparency (Definition 2 and Algorithm 5). Algorithm 23 presents the $\mathsf{CT}_{C,h}$ algorithms, utilizing, as subroutines, the PKIX_C algorithms (Algorithms 7-12). Note the design uses a hash function h (used for Merkle-tree integrity protection). For convenience, we use a few timing parameters (Δ_r, Δ_{MMD}, Δ_{clk} and Δ_{com}) as constants in the algorithms.

Entities and state. The set N of stateful entities in $\mathsf{CT}_{C,h}$ has two additional types of entities in addition to CAs: *loggers* and *monitors*. Each logger maintains a log of certificates (st.*log*) and an identifier for the log (st.*log_id*). Each monitor keeps a dictionary of log public key certificates (in st.*logcerts*), which is indexed by log identifiers *log_id*. For each log *log_id* which the monitor oversees, the monitor keeps the latest version of the log received (st.*logs*[*log_id*]).

SCTs and pre-certificates. CT certificates use the same certification schemes and certificate fields as in PKIX, but add additional certificate types and encodings. One of these new certificate types is the *signed certificate timestamp (SCT)*. In CT, a certificate is valid only together with two valid SCTs from different loggers[16]. A CA wishing to issue a CT certificate, typically first certifies and submits a *pre-certificate* to loggers, using the $\mathsf{CT}_{C,h}$.AddPreChain-Req

[16] We simplified; the RFC [50] actually allows the use of different criteria for a sufficient set of SCTs.

operation at the loggers. The *pre-certificate* is a PKIX-invalid[17] certificate. A logger may add a pre-certificate to a log, then certify and return the corresponding SCT.[18]

Implementation. Algorithm 23 presents the implementation of $\mathsf{CT}_{C,h}$. The operations defined in Algorithm 23 include these required from any PKI (Definition 1), operations required from any transparent PKI (Definition 2), and operations[19] defined in CT specifications [47]: GetSTH-Req, GetSTH-Resp, AddPreChain-Req, GetEntries-Req and GetEntries-Resp. One more operation, not explicitly defined in the CT specifications [47], is $\mathsf{CT}_{C,h}$.Wakeup. The Wakeup operation is required to perform scheduled operations defined in [47], such as updating the log's *Signed Tree Head (STH)*, and periodically retrieving the updated logs by monitors.

The CT implementation uses several time-related bounds: Δ_{com}, Δ_{clk}, and Δ_w. These represent the maximum communication delay, maximum clock drift, and maximum wake-up imprecision, respectively. We assume these to be known to entities, and we use them also in the model predicates (Section 5.1) and for security analysis.

To schedule a Wakeup operation at time t, the protocol outputs ('WakeAt', t). The $\pi^{\text{WakeAt}}_{\Delta_{clk},\Delta_w}$ model predicate, described in Section 5.1, assumes that if ('WakeAt', t) is output at real time τ and $t \geq \tau$, then the Wakeup operation is invoked, at the entity requesting the wake-up, at real time within a window of Δ_w before and after time t, i.e., in $[t - \Delta_w, t + \Delta_w]$. If $t < \tau$, then the Wakeup operation is invoked at real time within a window of Δ_w before and after time τ.

We describe the implementation, focusing on the differences from PKIX. The revocation mechanism used in PKIX and CT is identical, hence, the $\mathsf{CT}_{C,h}$.Revoke algorithm simply invokes its PKIX counterpart. The $\mathsf{CT}_{C,h}$.Aux algorithm also simply invokes its PKIX counterpart to update the CRL in the state and to output a CRL. Different CT operations are intended to be invoked at different types of entities. All entities (CAs, loggers, and monitors) use $\mathsf{CT}_{C,h}$.Init for initialization. CAs use $\mathsf{CT}_{C,h}$.Certify, $\mathsf{CT}_{C,h}$.Revoke, and $\mathsf{CT}_{C,h}$.Aux for certifying and revoking certificates. All entities may use $\mathsf{CT}_{C,h}$.Valid, which is a stateless functions. The $\mathsf{CT}_{C,h}$.Wakeup operation is invoked periodically at loggers (to compute new STHs) and monitors (to ask for new STHs). In addition, $\mathsf{CT}_{C,h}$ has four monitor-specific operations: Monitor, Lookup, GetSTH-Resp and GetEntries-Resp. Finally, $\mathsf{CT}_{C,h}$ includes three logger-specific operations: AddPreChain-Req, GetSTH-Req, and GetEntries-Req.

$\mathsf{CT}_{C,h}$.Init. The algorithm adds the initialization details for loggers and monitors. First, the algorithm calls PKIX.Init to handle any of PKIX related details (Line 4). Then, the algorithm performs additional operations for loggers and monitors. First, the algorithm requests a wake-up in Δ_{MMD} from the current local time (Line 9).[20] Finally, for monitors, the algorithm initializes an empty dictionary of log certificates (indexed by log identifiers) for public keys of monitored logs, and an empty dictionary of monitored logs (also indexed by log identifiers), while for loggers, the algorithm sets its log identifier to its entity identifier and initializes an empty maintained log[21] (Lines 10-16).

$\mathsf{CT}_{C,h}$.Certify. The algorithm checks if the inputted data to be certified (*tbc*) contains a field *tbc.SCTset* which is non-empty (Line 1), and if so, sets the *tbc.type* field to 'pre-certificate' and issues a *pre-certificate*. If *tbc* includes a non-empty *tbc.SCTset* field, then $\mathsf{CT}_{C,h}$.Certify adds 'SCTset' to *tbc.PKIadded* and issues a final certificate. In either case, the certificate is generated using the PKIX.Certify algorithm (Lines 3 and 10).

$\mathsf{CT}_{C,h}$.Valid. First, use PKIX.Valid to validate that ψ is a PKIX-valid certificate. From Line 5, we verify that ψ is submitted with two valid SCTs, issued for $\psi.tbc - \psi.SCTset$, from different loggers, from logs in *aux.logs* which are trusted (i.e., verified with certificates from *store.logs*).

$\mathsf{CT}_{C,h}$.Monitor. $\mathsf{CT}_{C,h}$.Monitor initiates the monitoring process for a log. The algorithm receives a log certificate ψ_L and adds it to the dictionary of public key certificates for monitored logs (Line 1); . As a result, starting from the next time the $\mathsf{CT}_{C,h}$.Wakeup algorithm is invoked, the log maintained by $\psi_L.issuer$ will be monitored along with the other logs.

$\mathsf{CT}_{C,h}$.Wakeup. First, the algorithm asks for another Wake-up in Δ_{MMD} from the previous wake-up time it asked for (Line 1). In a logger, the algorithm updates the *signed tree hash (STH)* over the local log maintained by the logger (Lines 4-10), and stores the STH locally (Line 11) so it can be retrieved by monitors in $\mathsf{CT}_{C,h}$.GetSTH-Req. In a monitor, $\mathsf{CT}_{C,h}$.Wakeup outputs a request to get the latest STH for each of the loggers overseen by this monitor (Line 13).

$\mathsf{CT}_{C,h}$.Lookup. The algorithm returns the set of certificates that were issued to the given *subject*, based on the local copies of the logs that the monitor maintains (Lines 2-4).

$\mathsf{CT}_{C,h}$.AddPreChain-Req. Add a pre-certificate to the log. The algorithm ensures that the pre-certificate is a valid PKIX certificate, not already logged, and with validity period starting the maximal communication delay (Δ_{com}) and clock drift (Δ_{clk}) before the current local time or later (Line 1). It adds the certificate to the local log (Line 2), generates a Signed Certificate Timestamp (SCT; Lines 3-8) and outputs the SCT (Line 9).

$\mathsf{CT}_{C,h}$.GetSTH-Req. The algorithm outputs the latest STH certificate (Line 1) that was generated in $\mathsf{CT}_{C,h}$.Wakeup.

$\mathsf{CT}_{C,h}$.GetSTH-Resp. The algorithm is used to receive a new STH. It uses the logger's certificate, which is stored in the monitor's local state (Line 3), to check that the given STH was indeed signed by the logger, and that new certificates, in the STH, were added to the log (Line 4). If so, the monitor stores the new STH in a temporary location in its state (Line 5), until the newly logged certificates will be retrieved and the new STH root can be verified as a valid root (tree hash) that reflects the addition of the newly logged certificates. To that end, the algorithm outputs a 'GetEntries' request, to be sent

[17] The method of making pre-certificates invalid changed between CT 1.0 [47], where pre-certificates contain a 'poison' extension, and CT 2.0 [50], which we follow, where pre-certificates are encoded as CMS objects which are not PKIX-valid certificates.

[18] The *SCTset* field in CT certificates is a PKI-added field.

[19] We kept the operation names from RFC 6962 [47], although some may be sub-optimal.

[20] Δ_{MMD} is the 'maximal merge delay' [50].

[21] We simplify to one log per logger and using the entity identifier as the log identifier. It seems easy to extend support for multiple logs per logger, with distinct log identifiers.

to the logger, to which the logger should respond by sending the entries (of the newly logged certificates) (Line 6).

$\mathsf{CT}_{C,h}$.GetEntries-Req. The algorithm outputs a subset of the logged certificates, starting from index *start* until index *end*. The algorithm takes the relevant subset from the log which is saved in the entity's state (Line 2)and outputs it (Line 3).

$\mathsf{CT}_{C,h}$.GetEntries-Resp. The algorithm calculates the updated tree hash (*root*) by adding the newly logged certificates to the local copy of the log and computing the root ('tree hash') of the Merkle tree [13] (Lines 2-3). If the root in the STH that was stored temporarily in $\mathsf{CT}_{C,h}$.GetSTH-Resp indeed matches *root* (calculated in Line 3), the monitor saves the new certificates and the STH (Lines 7-8), and may inspect the newly logged certificates (Line 10).

D.2 Certificate Transparency with Audit

$\mathsf{CT}_{C,h}$.Audit *(Algorithm 25).* Return a set of log certificates with a certificate for each log for which there is an identifier in *aux.logs*, the log certificate is in *store.logs*, and ψ.*SCTset* includes a correctly-verifying SCT for the log, yet ψ is missing from the local log copy of the log, or return $\perp$ if no such logs were identified.

In Appendix E, we show that $\mathsf{CTwAudit}_{C,h}$ ensures Audited-Transparency under the same model predicate under which we show that $\mathsf{CT}_{C,h}$ ensures HL-Transparency.

Privacy concerns. The $\mathsf{CTwAudit}_{C,h}$ operation may expose the certificates used by Chrome to the monitor, which can be a privacy exposure. The definition of the related privacy property, and analysis of the exposure and of alternatives, are beyond our scope.

Algorithm 25 $\mathsf{CT}_C^{\mathcal{MT}^h}$.Audit (for $\mathsf{CTwAudit}_{C,h}$)

procedure CT.Audit(st, ψ, *aux*)

1: *CorruptLogCerts* $\leftarrow \emptyset$

2: $\forall log\_id \in aux.logs$: ▷ *For each log identifier in aux.logs*

3: **if** [st.$logcerts[log\_id] \neq \perp \wedge$ ▷ st.$logcerts$ *includes a corresponding log cert*; $\psi \notin$ st.$logs[log\_id] \wedge$ ▷ *But ψ is not in the corresponding local log copy*; $\exists \psi_{SCT} \in \psi.SCTset$ **s.t.:** ▷ *ψ includes a cert ψ_{SCT}*; $\psi_{SCT}.type$ = 'SCT' $\wedge$ ▷ *ψ_{SCT} is an SCT*; $\psi_{SCT}.log\_id = log\_id \wedge$ ▷ *ψ_{SCT} has this log identifier*; $C.\mathsf{Verify}_{\text{st}.logcerts[log\_id].pk}(\psi_{SCT}) \wedge$ ▷ *Verify ψ_{SCT}*; $\psi_{SCT}.tbc.cert = \psi.tbc - \psi.SCTset$] **then** ▷ *ψ_{SCT} signs ψ*

4: *CorruptLogCerts* += ('Corrupt', st.$logcerts[log\_id]$) ▷ *Add corrupt log cert*

5: **end if**

6: **if** *CorruptLogCerts* = $\emptyset$ **then**

7: **return** $\perp$ ▷ *No corrupt log certs found*

8: **else**

9: **return** *CorruptLogCerts* ▷ *Output corrupt log certs*

10: **end if**

end procedure

E ANALYSIS OF TRANSPARENCY IN CT AND CTwAudit

This appendix provides an analysis of transparency properties for the two versions of CT we define, $\mathsf{CT}_{C,h}$ and $\mathsf{CTwAudit}_{C,h}$. We show that $\mathsf{CT}_{C,h}$ ensures the HL Δ_{Tra}-Transparency requirement, under an appropriate model predicate; see Theorem 4. Then, we show that $\mathsf{CT}_{C,h}$ does not ensure the Guaranteed Δ-Transparency requirement or the Audited Δ-Transparency requirement under the same model predicate; see Theorem 5. Namely, CT only ensures transparency for certificates which are logged by at least one benign logger and include an SCT from the benign logger. Lastly, we show that $\mathsf{CTwAudit}_{C,h}$ ensures the Audited $\Delta^{\text{AUD}}_{\mathsf{Tra}}$-Transparency requirement under the same model predicate; see Theorem 6.

E.1 $\mathsf{CT}_{C,h}$ ensures HL Δ_{Tra}-Transparency

Theorem 4. *Let C be an existentially-unforgeable certificate scheme and h be a collision-resistant hash. Denote $\Delta_{\mathsf{Tra}} \equiv 6 \cdot \Delta_{clk} + 2 \cdot \Delta_{MMD} + 2 \cdot \Delta_w + 5 \cdot \Delta_{com}$. Then, $\mathsf{CT}_{C,h}$ satisfies the HL Δ_{Tra}-Transparency requirement under model predicate:*

$$\pi^{Init} \wedge \pi^{\mathsf{F}} \wedge \pi^{1LogCert} \wedge \pi^{Drift}_{\Delta_{clk}} \wedge \pi^{Com}_{\Delta_{com}} \wedge \pi^{WakeAt}_{\Delta_{clk},\Delta_w} \quad (4)$$

Note 1: to simplify, we ignore the fact that h is a keyless hash, which limits the analysis to the Random Oracle Model (ROM). This could be avoided by using a keyed hash.

Note 2: Δ_{MMD} is called the *maximal merge delay*. The model predicate π^{Init} ensures that the value of Δ_{MMD} given to each entity at initialization is the same as $params.\Delta_{MMD}$.

Proof. (Sketch) Suppose $\mathsf{CT}_{C,h}$ *does not* satisfy the HL Δ_{Tra}-Transparency requirement under the model predicate of Equation 4. By definition, there exists a PPT adversary $\mathcal{A}_{\mathsf{Tra}}$ that satisfies:

$$\Pr\left[\begin{array}{c} \pi^{HLTra}_{\Delta_{\mathsf{Tra}}}(T) = \perp, \text{ where} \\ T \leftarrow \mathbf{Exec}_{\mathcal{A}_{\mathsf{Tra}},\mathsf{CT}_{C,h}}(params) \end{array}\right] \notin Negl(|params|) \quad (5)$$

Following Equation (5) and the π^{HLTra}_{Δ} predicate (in Definition 4), with non-negligible probability over the transcripts T of executions of $\mathsf{CT}_{C,h}$ with $\mathcal{A}_{\mathsf{Tra}}$, we have $f^{\mathsf{Tra}}_{\Delta_{\mathsf{Tra}}}(T, \top) \neq$ 'G', where the $f^{\mathsf{Tra}}_{\Delta}$ function is defined in Algorithm 5.

Following the $f^{\mathsf{Tra}}_{\Delta}$ function, which in Line 3 returns ('G') if the HL flag is set but the NoHonestLog function (Algorithm 15) returns $\top$, and the implementation of $\mathsf{CT}_C^{\mathcal{MT}^h}$.Valid (in Algorithm 23), none of the loggers providing SCTs in ψ are honest. Therefore we can safely assume that in all these executions, the adversary will ensure that the NoHonestLog function (Algorithm 15) will return $\perp$.

Namely, the adversary will provide in *aux.logs* a set of log identifiers $logs_\psi$, such that ψ remains valid even if we validate using only these log identifiers, i.e., with $aux.logs = logs_\psi$. Furthermore, there is at least one of them, $id_{HL} \in logs_\psi$, which we call the 'honest log identifier', which is 'essential', i.e., ψ is invalid with $aux.logs = logs_\psi - id_{HL}$, such that the certificate of log id_{HL} in *store.logs*, i.e., $store.logs[id_{HL}]$, was generated by a benign entity ι_{HL}. Since id_{HL} is 'essential' and following the implementation of $\mathsf{CT}_C^{\mathcal{MT}^h}$.Valid, this implies that $\psi.SCTset$ must contain a valid SCT of the honest log id_{HL}, and the SCT certifies the logged fields of ψ.

Also, since $f^{\mathsf{Tra}}_{\Delta_{\mathsf{Tra}}}$ does not return 'G' in these executions, we know that all of the following must hold:

(1) ψ is valid at time $t \geq \psi.from + \Delta_{\mathsf{Tra}}$ w.r.t. *store* and *aux*
(2) The logs of ψ are monitored by a benign monitor ι_M since $t - \Delta_{\mathsf{Tra}}$; this includes the honest log, i.e., log id_{HL}.
(3) The same benign monitor ι_M is unaware of the logged fields of ψ after real time t.

Algorithm 23 $\mathsf{CT}_C^{\mathcal{MT}^h}$, i.e., CT implemented using certificate scheme C and Merkle Tree $\mathcal{MT}^h$; see [13] for $\mathcal{MT}^h$

procedure $\mathsf{CT}_C^{\mathcal{MT}^h}$.Init (st, $clk, params$)

if st $\neq \perp$ **then** ▷ *If state is non-empty*
  return (st, $\perp$) ▷ *Do not initialize*
end if
(st, out) $\leftarrow$ PKIX$_C$.Init(st, $clk, params$) ▷ *See Algorithm 7*
st.$role \leftarrow params.role$ ▷ *Set entity's role*
st.$\Delta_{MMD} \leftarrow params.global.\Delta_{MMD}$ ▷ *Set Δ_{MMD} delay*
st.$t_{wake} \leftarrow clk +$ st.Δ_{MMD} ▷ *Save next wake-up time*
if st.$role \in$ {'Logger', 'Monitor'} **then** ▷ *Only monitors and loggers*
  out += ('WakeAt', st.t_{wake}) ▷ *Add next wakeup time*
  if st.$role$ = 'Monitor' **then** ▷ *If monitor*
    st.$logcerts \leftarrow \perp$ ▷ *Init empty log certs dictionary*
    st.$logs \leftarrow \perp$ ▷ *Init empty logs dictionary*
  else ▷ *If logger*
    out += ('LogCert', out['SelfCert']) ▷ *Add entity's self cert as log cert*
    st.$log \leftarrow \perp$ ▷ *Initialize log*
  end if
end if
return (st, out) ▷ *Output*

end procedure

procedure $\mathsf{CT}_C^{\mathcal{MT}^h}$.Certify (st, clk, tbc)

if $tbc.SCTset = \perp$ **then** ▷ *If SCTset is empty*
  $tbc.type \leftarrow$ 'pre-certificate' ▷ *Set type to 'pre-certificate'*
  (st, ψ_{pre}) $\leftarrow$ PKIX$_C$.Certify(st, clk, tbc) ▷ *See Algorithm 8*
  return (st, ψ_{pre}) ▷ *Output pre-certificate*
else ▷ *If SCTset is not empty*
  $tbc.PKIadded$ += {'SCTset'} ▷ *Set 'SCTset' as a 'PKIadded' field*
  if $tbc.type \in$ {'SCT', 'STH'} **then** ▷ *Type which is not allowed!*
    return (st, $\perp$)
  end if
  (st, ψ) $\leftarrow$ PKIX$_C$.Certify(st, clk, tbc) ▷ *See Algorithm 8*
  return (st, ψ) ▷ *Output final certificate*
end if

end procedure

procedure $\mathsf{CT}_C^{\mathcal{MT}^h}$.Revoke (st, clk, ψ)

return PKIX$_C$.Revoke(ψ) ▷ *See Algorithm 9*

end procedure

procedure $\mathsf{CT}_C^{\mathcal{MT}^h}$.Aux (st, clk)

return PKIX$_C$.Aux(st, clk) ▷ *See Algorithm 12*

end procedure

procedure $\mathsf{CT}_C^{\mathcal{MT}^h}$.Valid ($\psi, t$, *store, aux*)

return PKIX$_C$.Valid(ψ, t, *store.CAs, aux.chains*) $\wedge$ ▷ *See Algorithm 11*
  $(\exists \psi_1, \psi_2 \in \psi.SCTset)$ **s.t.** ▷ *Two certificates*
    $\psi_1.issuer \neq \psi_2.issuer \wedge$ ▷ *From different loggers*
    $(\forall \psi_{SCT} \in \{\psi_1, \psi_2\})$ **:**
      $\psi_{SCT}.type =$ 'SCT' $\wedge$ ▷ *ψ_{SCT} is an SCT*
      $\psi_{SCT}.log\_id \in aux.logs \wedge$ ▷ *Log identifier is in aux.logs*
      C.Verify$_{store.logs[\psi_{SCT}.log_id].pk}(\psi_{SCT}) \wedge$ ▷ *Verify ψ_{SCT}*
      $\psi_{SCT}.tbc.cert = \psi.tbc - \psi.SCTset$ ▷ *ψ_{SCT} signs ψ*

end procedure

procedure $\mathsf{CT}_C^{\mathcal{MT}^h}$.VCerts ($\psi, t$, *store, aux*)

return PKIX$_C$.VCerts(ψ, t, *store, aux*) $\cup \psi.SCTset$ ▷ *See Algorithm 10*

end procedure

procedure $\mathsf{CT}_C^{\mathcal{MT}^h}$.Monitor (st, clk, ψ_L)

st.$logcerts[\psi_L.subject] \leftarrow \psi_L$ ▷ *Save log certificate ψ_L*
return st ▷ *Output*

end procedure

procedure $\mathsf{CT}_C^{\mathcal{MT}^h}$.AddPreChain-Req (st, clk, ψ, *store, aux*)

if $\psi \notin$ st.$log \wedge clk \leq \psi.from + \Delta_{clk} + \Delta_{com}$ **then**
  st.log += ψ ▷ *Store locally*
  $tbc \leftarrow$ {('type', 'SCT')} ▷ *Certificate of type SCT*
  $tbc.issuer \leftarrow$ st.$\imath$ ▷ *The issuer of the SCT*
  $tbc.log\_id \leftarrow$ st.$\imath$ ▷ *The log to which ψ was added*
  $tbc.timestamp \leftarrow clk$ ▷ *Timestamp (when issued)*
  $tbc.cert \leftarrow \psi.tbc$ ▷ *Certificate information*
  $\psi_{SCT} \leftarrow C$.Certify$_{\text{st.}sk}(tbc)$ ▷ *Issue certificate*
  return (st, ('AddPreChain-Resp', ψ_{SCT})) ▷ *Respond with SCT*
end if
return (st, $\perp$) ▷ *Invalid request*

end procedure

procedure $\mathsf{CT}_C^{\mathcal{MT}^h}$.GetSTH-Req (st, clk)

return (st, ('GetSTH-Resp', st.ψ_{STH})) ▷ *Output*

end procedure

procedure $\mathsf{CT}_C^{\mathcal{MT}^h}$.GetSTH-Resp (st, $clk, store, \psi$)

$out \leftarrow \perp$
$log \leftarrow$ st.$logs[\psi.log\_id]$ ▷ *Monitor's local log data*
$\psi_L \leftarrow store.logs[\psi.log\_id]$ ▷ *Log certificate*
if C.Verify$_{\psi_L.pk}(\psi) \wedge |log.entries| < \psi.size$ **then** ▷ *Valid STH*
  st.$newSTH \leftarrow \psi$ ▷ *Store new STH*
  $out \leftarrow$ ('GetEntries-Req', $\psi.issuer$,
    $start = |log.entries|, end = \psi.size$) ▷ *Ask for new certs*
end if
return (st, out) ▷ *Output*

end procedure

procedure $\mathsf{CT}_C^{\mathcal{MT}^h}$.GetEntries-Req (st, $clk, start, end$)

$entries.log\_id \leftarrow$ st.$\imath$ ▷ *The log identifier*
$entries.certs \leftarrow$ st.$log[start : end]$ ▷ *Desired certificates*
return (st, ('GetEntries-Resp', $entries$)) ▷ *Output*

end procedure

procedure $\mathsf{CT}_C^{\mathcal{MT}^h}$.GetEntries-Resp (st, $clk, store, entries$)

$\psi_L \leftarrow store.logs[entries.log\_id]$ ▷ *Log certificate*
$log \leftarrow$ st.$logs[entries.log\_id]$ ▷ *Monitor's local data*
$root \leftarrow \mathcal{MT}^h(log.entries + entries.certs)$ ▷ *Calc new root*
if $root =$ st.$newSTH.root$ **then** ▷ *Verify Merkle root*

if $\left[\begin{array}{l} \forall \psi \in entries.certs: \\ \quad clk \leq \psi.from + 6 \cdot \Delta_{clk} + 2 \cdot \Delta_{MMD} + 2 \cdot \Delta_w + 5 \cdot \Delta_{com} \end{array}\right]$ **then**
  $log.entries$ += $entries.certs$ ▷ *Update log certs*
  $log.STH \leftarrow$ st.$newSTH$ ▷ *Update local log STH*
  st.$logs[entries.log\_id] \leftarrow log$ ▷ *Update local log data*
  Optionally: examine new certificates for problems, e.g., potential phishing
end if
end if
return st ▷ *Output*

end procedure

Algorithm 24 $\mathsf{CT}_C^{\mathcal{MT}^h}$ continued

procedure $\mathsf{CT}_C^{\mathcal{MT}^h}$.Wakeup (st, $clk, data$)

1: $out \leftarrow$ ('WakeAt', st.t_{wake}) ▷ *Set next wake-up*
2: st.$t_{wake} \leftarrow$ st.t_{wake} + st.Δ_{MMD} ▷ *Save next wake-up time*
3: **if** st.$role$ = 'Logger' **then**
4: $tbc \leftarrow$ {('type', 'STH')} ▷ *Certificate of type STH*
5: $tbc.issuer \leftarrow$ st.ι ▷ *The issuer of the STH*
6: $tbc.log\_id \leftarrow$ st.ι ▷ *The log identifier of the STH*
7: $tbc.from \leftarrow clk$ ▷ *When was issued*
8: $tbc.to \leftarrow clk$ + st.Δ_{MMD} ▷ *Time for next STH*
9: $tbc.root \leftarrow \mathcal{MT}^h$(st.$log$)
10: $tbc.size \leftarrow$ |st.log| ▷ *Log size*
11: st.$\psi_{STH} \leftarrow C$.Certify$_{\text{st.}sk}(tbc)$ ▷ *Issue certificate*
12: **else if** st.$role$ = 'Monitor' **then** ▷ *Ask for logs' latest STH*
13: out += {('GetSTH-Req', $\psi_L.issuer$) | $\forall \psi_L \in$ st.$logcerts$}
14: **end if**
15: **return** (st, out) ▷ *Output*

end procedure

procedure $\mathsf{CT}_C^{\mathcal{MT}^h}$.Lookup (st, $clk, subject$)

1: $certs \leftarrow \perp$ ▷ *Initialize set to* $\perp$
2: **if** st.$role$ = 'Monitor' **then**
3: $certs \leftarrow \left\{ \psi \;\middle|\; \begin{array}{c} \forall log \in \text{st.}logs, \psi \in log \text{ s.t.} \\ \psi.subject = subject \end{array} \right\}$ ▷ $subject$'s certificates
4: **end if**
5: **return** (st, $certs$) ▷ *Output*

end procedure

Since ι_L is benign, and ψ_{SCT} is a valid SCT, then with overwhelming probability, ψ_{SCT} was properly generated and outputted by ι_L. $\mathsf{CT}_{C,h}$.AddPreChain-Req is the only operation where ι_L issues an SCT certificate. Let τ_{SCT} denote the real time when ι_L issued the SCT. Since the SCT certifies the logged fields of ψ, then the $\mathsf{CT}_{C,h}$.AddPreChain-Req operation must have received a certificate ψ' which has the same logged fields as ψ. From Line 1 of $\mathsf{CT}_{C,h}$.AddPreChain-Req and since the $from$ field is a logged field in $\mathsf{CT}_{C,h}$, then ι_L would issue the SCT only if its clock upon receiving ψ' is at most $\psi.from + \Delta_{clk} + \Delta_{com}$, so the real time τ_{SCT} is at most $\psi.from + 2 \cdot \Delta_{clk} + \Delta_{com}$ (due to the $\pi^{\text{Drift}}_{\Delta_{clk}}$ model predicate).

Since ι_L is benign, it asks for wake-up at time values which are Δ_{MMD} apart (see the $\mathsf{CT}_{C,h}$.Init and $\mathsf{CT}_{C,h}$.Wakeup operations). From the π^{Init} model predicate (Algorithm 22), the Init operation is called before any other operation, including AddPreChain-Req. Due to the $\pi^{\text{WakeAt}}_{\Delta_{clk},\Delta_w}$ and $\pi^{\text{Drift}}_{\Delta_{clk}}$ model predicates, the maximum time between an $\mathsf{CT}_{C,h}$.Init event and the first following $\mathsf{CT}_{C,h}$.Wakeup event at the same entity is $2 \cdot \Delta_{clk} + \Delta_{MMD} + \Delta_w$, and similarly, the maximum time between two consecutive $\mathsf{CT}_{C,h}$.Wakeup events at the same entity is $2 \cdot \Delta_{clk} + \Delta_{MMD} + \Delta_w$. Thus, the $\mathsf{CT}_{C,h}$.Wakeup event will occur after at most $2 \cdot \Delta_{clk} + \Delta_{MMD} + \Delta_w$, i.e., at or before $\tau_{SCT} + 2 \cdot \Delta_{clk} + \Delta_{MMD} + \Delta_w$. At this time, ι_L updates its STH to reflect the logging of ψ' (and possibly other certificates). The STH is signed by ι_L, and includes the Merkle-tree digest of ι_L's log, which contains all certificates logged by ι_L in its log until this time.

The benign monitor ι_M which monitors ι_L's log also asks for wake-up at time values which are Δ_{MMD} apart. Again, by the $\pi^{\text{WakeAt}}_{\Delta_{clk},\Delta_w}$ and $\pi^{\text{Drift}}_{\Delta_{clk}}$ model predicates, the operation will occur after at most $2 \cdot \Delta_{clk} + \Delta_{MMD} + \Delta_w$ after ι_L updates its STH, i.e., at or before $\tau_{SCT} + 4 \cdot \Delta_{clk} + 2 \cdot \Delta_{MMD} + 2 \cdot \Delta_w$. In every invocation of $\mathsf{CT}_{C,h}$.Wakeup, ι_M requests a $\mathsf{CT}_{C,h}$.GetSTH-Req operation from the logger of every log that ι_M is monitoring, which includes ι_L. From the $\pi^{\text{Com}}_{\Delta_{com}}$ model predicate, the $\mathsf{CT}_{C,h}$.GetSTH-Req operation in ι_L occurs at most Δ_{com} afterwards, i.e., at or before $\tau_{SCT} + 4 \cdot \Delta_{clk} + 2 \cdot \Delta_{MMD} + 2 \cdot \Delta_w + \Delta_{com}$ (including the possible impacts of the local clock bias, the wake-up imprecision, and the network delay).

From the $\pi^{\text{Com}}_{\Delta_{com}}$ model predicate, the corresponding event with operation $\mathsf{CT}_{C,h}$.GetSTH-Resp in ι_M occurs within Δ_{com}. Suppose that no malicious entries are received by ι_M for the benign log. This means that the size of ι_M's local copy of the benign log is less than the size specified in the STH. Since ι_L is benign, then the STH is correctly signed. Since ι_M is benign and due to the π^{1LogCert} model predicate, ι_M only has the correct log certificate and should accept benign STHs and benign entries for the benign log. As shown in $\mathsf{CT}_{C,h}$.GetSTH-Resp, this means that the monitor ι_M identifies that new certificates were logged and asks for them, i.e., invokes $\mathsf{CT}_{C,h}$.GetEntries-Req at ι_L; and the logger ι_L immediately responds by sending the certificates, including ψ'. Then, ι_M receives ψ' in the corresponding $\mathsf{CT}_{C,h}$.GetEntries-Resp event. This round-trip of communication will take at most $2 \cdot \Delta_{com}$; hence, ι_M receives ψ' at or before time τ_ψ, defined as:

$$\begin{aligned} \tau_\psi &\equiv \tau_{SCT} + 4 \cdot \Delta_{clk} + 2 \cdot \Delta_{MMD} + 2 \cdot \Delta_w + 4 \cdot \Delta_{com} \\ &\leq \psi.from + 2 \cdot \Delta_{clk} + \Delta_{com} \\ &\quad + 4 \cdot \Delta_{clk} + 2 \cdot \Delta_{MMD} + 2 \cdot \Delta_w + 4 \cdot \Delta_{com} \\ &= \psi.from + 6 \cdot \Delta_{clk} + 2 \cdot \Delta_{MMD} + 2 \cdot \Delta_w + 5 \cdot \Delta_{com} \\ &= \psi.from + \Delta_{\mathsf{Tra}} \end{aligned} \tag{6}$$

Since ι_M is a benign monitor and due to the π^{1LogCert} model predicate, ι_M should accept the benign entries for the benign log, so in the $\mathsf{CT}_{C,h}$.GetEntries-Resp event, ι_M stores ψ' in its local copy of ι_L's log. After this time, when the $\mathsf{CT}_{C,h}$.Lookup operation is invoked at ι_M with input $\psi.subject$, then ι_L's output includes ψ', which has the same logged fields as ψ. This contradicts the assumption that $\mathcal{A}_{\mathsf{Tra}}$ produces, with non-negligible probability, a certificate ψ that is valid at time $t \geq \psi.from + \Delta_{\mathsf{Tra}}$ but no certificate with the same logged fields as the logged fields of ψ is known to ι_M after time t. Therefore, either:

- ψ_{SCT} was not properly generated by ι_L. Since ι_L is benign, this means that ι_L's signature in ψ_{SCT} was forged, which contradicts the existential unforgeability of C.

- Using a malicious STH, malicious entries were accepted by ι_M for the benign log, which prevented ι_M from receiving ψ. Since ι_L is benign, this implies that ι_L's signature was forged, which contradicts the existential unforgeability of C.
- Using a correct Merkle tree root output in an STH by ι_L, malicious entries, which were not used to compute the root by ι_L, were accepted by ι_M for the benign log, which prevented ι_M from receiving ψ'. This contradicts the collision-resistance of the Merkle tree $\mathcal{MT}^h$, which is guaranteed when h is a collision-resistant hash function (see [13]).

□

E.2 $\mathsf{CT}_{C,h}$ does not ensure Guaranteed Δ-Transparency or Audited Δ-Transparency

We next show that CT *fails* to satisfy the Guaranteed Δ-Transparency requirement and the Audited Δ-Transparency requirement, for any finite delay Δ, assuming the same model predicate (of Equation 4).

Theorem 5. *Let C be a secure certificate scheme and h be a collision-resistant hash, and let Δ be any finite delay. Then,* $\mathsf{CT}_{C,h}$ does not *satisfy the Guaranteed Δ-Transparency requirement or the Audited Δ-Transparency requirement under the model predicate of Equation 4.*

Proof. Notice that the adversary may control all of the logs in *aux.logs*, i.e., each log $id_L \in$ *aux.logs* may be either controlled by a corrupt entity in the execution or may be a fake log of the adversary, not existing in the execution. We construct an adversary $\mathcal{A}^{\mathrm{G}}_{\mathsf{Tra}}$ that controls all of the logs in *aux.logs* through corrupt entities in the execution. That is, each $id_L \in$ *aux.logs* is controlled by a logger $\iota_L \in T.\mathsf{F}$ (where T is an execution transcript). For each such adversary-controlled logger ι_L, $\mathcal{A}^{\mathrm{G}}_{\mathsf{Tra}}$ instructs ι_L to execute $\mathsf{CT}_{C,h}$ as described in §D.1, with the single change of eliminating Line 2 from AddPreChain-Req. As a result, each such adversary-controlled logger ι_L generates an SCT to every certificate with an acceptable 'from' field time value, yet ι_L never logs these certificates in its log, and therefore, never informs monitors about them. This means that despite the fact that there is a valid SCT from ι_L for each of the certificates, none of these certificates will be known to monitors. Thus, even if ψ is valid and thus includes valid SCTs for at least two of the log identifiers in *aux.logs*, and monitor ι_M monitors all of the logs in *aux.logs*, no certificate with the same logged fields as the logged fields of ψ may be known to ι_M during the entire execution. Hence, the existence of such $\mathcal{A}^{\mathrm{G}}_{\mathsf{Tra}}$ means that the Guaranteed Δ-Transparency requirement is not satisfied under the model predicate of Equation 4.

$\mathsf{CT}_{C,h}$ also does not satisfy the Audited Δ-Transparency requirement under the model predicate of Equation 4, as we show next. We construct an adversary $\mathcal{A}^{\mathrm{A}}_{\mathsf{Tra}}$ which works like $\mathcal{A}^{\mathrm{G}}_{\mathsf{Tra}}$ and in addition, $\mathcal{A}^{\mathrm{A}}_{\mathsf{Tra}}$ does the following. After invoking AddPreChain-Req at a corrupt logger $\iota_L \in T.\mathsf{F}$ which is in *aux.logs* to correctly generate and output an SCT certificate ψ_{SCT} with $\psi_{SCT}.log\_id = \iota_L$, $\mathcal{A}^{\mathrm{A}}_{\mathsf{Tra}}$ certifies a corresponding valid certificate ψ which includes ψ_{SCT}, and then $\mathcal{A}^{\mathrm{A}}_{\mathsf{Tra}}$ waits until after $\psi.from + \Delta$. Then, $\mathcal{A}^{\mathrm{A}}_{\mathsf{Tra}}$ invokes the 'Audit' operation with input ψ, aux) at a monitor ι_M which monitors ι_L. As described above, ι_L never logs in its log the certificates for which it generates SCTs and never informs monitors about them. Consequently, ι_M will not be aware of ψ before $\mathcal{A}^{\mathrm{A}}_{\mathsf{Tra}}$ audits ι_M with input ψ, aux), where ψ includes the correct SCT certificate ψ_{SCT} for log ι_L. Since $\mathsf{CT}_{C,h}$ does not have an 'Audit' operation which would allow monitors to identify certificates of corrupted logs, then the monitor audited by $\mathcal{A}^{\Delta}_{\mathsf{Tra}}$ will not output the corrupt log certificate for log ι_L. Thus, ι_M will fail to correctly identify a corrupt log certificate. Hence, trivially, the existence of such $\mathcal{A}^{\mathrm{G}}_{\mathsf{Tra}}$ means that the Audited Δ-Transparency requirement is not satisfied under the model predicate of Equation 4. □

E.3 $\mathsf{CTwAudit}_{C,h}$ ensures Audited $\Delta^{\mathbf{AUD}}_{\mathsf{Tra}}$-Transparency

Finally, Theorem 6 proves that $\mathsf{CTwAudit}_{C,h}$ ensures the Audited $\Delta^{\mathrm{AUD}}_{\mathsf{Tra}}$-Transparency requirement under the model predicate of Equation 4.

Theorem 6. *Let C be an existentially-unforgeable certificate scheme and h be a collision-resistant hash. Denote $\Delta^{AUD}_{\mathsf{Tra}} \equiv 7 \cdot \Delta_{clk} + 2 \cdot \Delta_{MMD} + 2 \cdot \Delta_w + 5 \cdot \Delta_{com}$, where Δ_{MMD} is the maximal merge delay. Then,* $\mathsf{CTwAudit}_{C,h}$ *satisfies the Audited $\Delta^{AUD}_{\mathsf{Tra}}$-Transparency requirement under the model predicate of Equation 4.*

Proof. (Sketch) Suppose that $\mathsf{CTwAudit}_{C,h}$ *does not* satisfy the Audited $\Delta^{\mathrm{AUD}}_{\mathsf{Tra}}$-Transparency requirement under the model predicate of Equation 4. By definition, there exists a PPT adversary $\mathcal{A}^{\mathrm{AUD}}_{\mathsf{Tra}}$ that satisfies:

$$\Pr\left[\begin{array}{c} \pi^{AudTra}_{\Delta^{\mathrm{AUD}}_{\mathsf{Tra}}}(T) = \perp, \text{ where} \\ T \leftarrow \mathbf{Exec}_{\mathcal{A}^{\mathrm{AUD}}_{\mathsf{Tra}},\mathsf{CT}_{C,h}}(params) \end{array}\right] \notin Negl(|params|) \quad (7)$$

Following Equation (7) and the π^{AudTra}_{Δ} predicate (in Definition 4), with non-negligible probability over the transcripts T of executions of $\mathsf{CT}_{C,h}$ with $\mathcal{A}^{\mathrm{AUD}}_{\mathsf{Tra}}$, we have $f^{\mathsf{Tra}}_{\Delta^{\mathrm{AUD}}_{\mathsf{Tra}}}(T, \perp) \notin \{\text{'G'}, \text{'A'}\}$, where the $f^{\mathsf{Tra}}_{\Delta}$ function is defined in Algorithm 5.

That is, there exists an adversary $\mathcal{A}^{\mathrm{AUD}}_{\mathsf{Tra}}$ that produces, with non-negligible probability, a certificate ψ that is valid at time $t \geq \psi.from + \Delta^{\mathrm{AUD}}_{\mathsf{Tra}}$ w.r.t. *store* and *aux*, such that a benign monitor ι_M who is monitoring all the logs in *aux.logs* since $t - \Delta^{\mathrm{AUD}}_{\mathsf{Tra}}$, is unaware of ψ after time t, and when audited at some time after t, it does not identify any corrupt log certificate from *aux.logs* and *store.logs* or it identifies an incorrect log certificate as corrupt (either a certificate that is not in *aux.logs* and *store.logs* or a certificate which was outputted as a log certificate by a benign entity). We use t to denote some specific point in time to derive the bound for $\Delta^{\mathrm{AUD}}_{\mathsf{Tra}}$ based on the Δ_{MMD}, Δ_{clk}, Δ_{com}, and Δ_w parameters.

Since ψ is valid at time t, then Lines 2-8 of $\mathsf{CTwAudit}_{C,h}$.Valid (in Algorithm 23), ensure that ψ contains at least two SCTs which correctly verify using the public keys of the corresponding log certs in *store.logs*. Let us consider the set of all such SCTs and let us call the logs corresponding to these SCTs the 'SCT logs'. There are two cases: (1) at least one of the SCT logs is benign; (2) every one of the SCT logs is corrupt (i.e., either is controlled by a corrupt entity in the execution or is a fake log of the adversary, not existing in the execution).

Suppose that at least one of the SCT logs is benign. By Theorem 4, ι_M would be aware of a certificate with the same logged fields as the logged fields of ψ after time t, since the $\Delta^{\mathrm{AUD}}_{\mathsf{Tra}}$ in Theorem 6 is $\geq$ the Δ_{Tra} in Theorem 4. This is a contradiction. This implies that every one of the SCT logs is corrupt.

From $\mathsf{CT}_{C,h}$.Audit (Algorithm 25), when ι_M is audited, every log certificate in the output of ι_M has its log identifier in *aux.logs*, the log certificate is in *store.logs*, and the corresponding log is one of the SCT logs. This implies that if all of the SCT logs are corrupt, then the reason for FailedIdentifyCorrupted$(T, \psi, t, store, aux, \iota_M)$ (Algorithm 19) being true cannot be that ι_M identified an incorrect log cert (a log cert without its log identifier in *aux.logs* or a log cert not in *store.logs* or a log cert outputted by a benign entity) as corrupt, but rather, it must be that ι_M failed the audit due to not outputting any corrupt log cert when audited.

As stated before, since ψ is valid at time t, it must contain at least two correctly-verifying SCTs of logs in *aux.logs* and *store.logs*, so if ι_M is not aware of any certificate with the same logged fields as the logged fields of ψ when audited, then it should output at least two corrupt log certificates. If ι_M did not output any corrupt log certificate when audited, this implies that there must exist a certificate ψ' with the same logged fields as the logged fields of ψ such that ι_M was unaware of ψ' in the 'Lookup' event after time t but aware of ψ' in the 'Audit' event after time t. This implies that ι_M must have received ψ' in a $\mathsf{CT}_{C,h}$.GetEntries-Resp (in Algorithm 23) event after the 'Lookup' event and before the 'Audit' event.

Both the 'Lookup' and 'Audit' events occurred after time t, and $t \geq \psi.from + \Delta_{\mathsf{Tra}}^{\mathrm{AUD}} = \psi.from + 7 \cdot \Delta_{clk} + 2 \cdot \Delta_{MMD} + 2 \cdot \Delta_w + 5 \cdot \Delta_{com}$, so by the $\pi_{\Delta_{clk}}^{\mathrm{Drift}}$ model predicate, the $\mathsf{CT}_{C,h}$.GetEntries-Resp event occurred with local time $clk > \psi.from + 7 \cdot \Delta_{clk} + 2 \cdot \Delta_{MMD} + 2 \cdot \Delta_w + 5 \cdot \Delta_{com} - \Delta_{clk} = \psi.from + 6 \cdot \Delta_{clk} + 2 \cdot \Delta_{MMD} + 2 \cdot \Delta_w + 5 \cdot \Delta_{com}$. Since the $from$ field is a logged field in $\mathsf{CTwAudit}_{C,h}$, then $\psi'.from = \psi.from$, which implies that the $\mathsf{CT}_{C,h}$.GetEntries-Resp event occurred with local time $clk > \psi'.from + 6 \cdot \Delta_{clk} + 2 \cdot \Delta_{MMD} + 2 \cdot \Delta_w + 5 \cdot \Delta_{com}$. However, from $\mathsf{CT}_{C,h}$.GetEntries-Resp (in Algorithm 23), ι_M would only accept ψ' if $clk \leq \psi'.from + 6 \cdot \Delta_{clk} + 2 \cdot \Delta_{MMD} + 2 \cdot \Delta_w + 5 \cdot \Delta_{com}$. So ι_M would not accept ψ', and consequently, ι_M would not be aware of ψ' in the 'Audit' event after time t. This is a contradiction.

□