Provable Security for PKI Schemes

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

Amir Herzberg
Dept. of Computer Science and Engineering, University of Connecticut
Storrs, CT

Ewa Syta†
Dept. of Computer Science, Trinity College
Hartford, CT

ABSTRACT

PKI provides a critical foundation to applied cryptographic protocols. However, there are no rigorous security specifications for PKI, and therefore, no PKI schemes were proven secure. This is problematic considering the extensive reliance on PKI, the multiple failures of PKI systems, and the fact that some proposed and deployed PKI schemes have complex design and advanced goals. The lack of specifications and proofs for PKI schemes means that applied cryptographic systems that use PKI are analyzed by adopting overly simplified models of the PKI, often, simply assuming secure public keys.

We present game-based security specifications for PKI schemes, and prove the security of the two most important and widely deployed schemes: PKIX and Certificate Transparency (CT), both based on version 3 of the X.509 standard, and using the (standard) CRL revocation mechanism. The proof shows a reduction from an adversary that 'wins' the PKI-specifications game to an adversary that 'wins' against the underlying signature scheme or hash function. This is the first reduction-based definition and proof of security for a realistic PKI scheme.

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 [21], and the first version of the X.509 standard [8] 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 [35]. *Certificate Transparency (CT)* [34, 24] is a recent, widely-deployed extension to PKIX, motivated by multiple PKI failures, mainly, rogue certificates issued by corrupt or negligent CAs. A significant number of other PKI schemes were proposed recently, with different goals and properties, and different, non-trivial designs, including [38, 10, 30, 20, 42, 3, 45, 40, 41, 28, 12, 2, 43, 22].

Considering the importance, variety and complexity of (some) PKI schemes, it is essential to ensure their security. However, this work is the first to define the security of a (non-trivial[1]) PKI scheme. 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 [14] and secure signature schemes [15].

The lack of rigorous specifications and analysis for PKI schemes is especially alarming, considering that PKI provides a critical infrastructure to applied cryptography, i.e., security of many applied cryptographic systems depend on the security of the underlying PKI. The extensive efforts to prove the security of 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 are a major concern in practice. To address these concerns, we present the *rigorous (game-based) definitions of security requirements for PKI schemes*, and then the *first reduction-based proofs of security* for practical PKI schemes; see Table 1.

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)*. Existing works often use informal security requirements and models for PKI schemes, often tailored to a specific construction. In fact, as we now discuss, even intuitively straightforward definitions have many subtleties, which may be overlooked or ignored.

Consider *accountability*, the most basic security property of PKI schemes. 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 CA identified as the *issuer* of ψ, denoted as $\psi.issuer$. That said, surely $\psi.issuer$ should not be held accountable for ψ, if there is no guarantee that $\psi.issuer$ indeed issued ψ. Namely, if the public verification key pk used to validate ψ is *not* a correct public-key of $\psi.issuer$. Even more clearly, $\psi.issuer$ can only be considered accountable if it is a real, supposedly trustworthy entity. To illustrate, consider a scenario where a rogue CA issues a certificate ψ' which fraudulently specifies pk as a public verification key of $\psi.issuer$. According to the aforementioned instinct, ψ' would allow to 'frame' $\psi.issuer$ to be held accountable for ψ, even though $\psi.issuer$ never issued ψ. Instead, the CA responsible (accountable) in this case for ψ should be the CA responsible for the fraudulent certificate ψ'. The definition of accountability, which we present later, supports a wide range of PKI schemes, and, in particular, the widely-used *certificate chains* mechanism, introduced in X.509 version 3, which allows intermediate CAs to issue certificates.

Revocation, supported by most applied PKIs, is also non-trivial to define. In fact, we found it necessary to define two variants: Full and

*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] A naive PKI ideal-functionality is presented in [7, 4, 13]; this functionality does not include basic functionalities such as *certificate chains, revocation* and *transparency*.

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

	Requirements				
			Δ-Revocation (Definition 8)		Δ-Transparency
PKI scheme	**Existential Unforgeability (Definition 6)**	**Accountability (Definition 7)**	**Weak**	**Full**	**(Definition 9)**
PKIX (X.509 version 3 with CRL)	✓ (Lemma 2)	✓ (Lemma 10)	✓ (Lemma 12)	✗ (Claim 3)	✗ (n/a)
PKIX with CT	✓ (Lemma 2)	✓ (Theorem 5)	✓ (Theorem 5)	✗ (Claim 3)	✗ (Claim 7)
PKIX with CT assuming honest loggers					✓ (Theorem 5)

Weak. The Full variant is more intuitive; basically, a PKI scheme ensures *Full 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, we realized that PKIX fails to ensure Full Δ-Revocation (Claim 3), and only ensures a weaker notion, which we call *Weak Δ-Revocation*.

Transparency is a relatively recently identified goal for PKIs, which aims to ensure that certificates are available for scrutiny, in the form of a *public log* where certificates must appear (within bounded time after being issued). This requirement was identified when, after PKI failures, it was realized that the accountability requirement may not suffice, if the attacker's use of a rogue certificate may go undetected. We show that PKIX, complemented by the *Certificate Transparency (CT)* specification [23, 25], ensures transparency if (and only if) we can trust the integrity of the CT *Logger* entities.

We also define a PKI *existential unforgeability* requirement, which ensures that if ψ is a valid certificate, as validated by the public key generated by a benign $\psi.issuer$, then $\psi.issuer$ has indeed certified ψ. For X.509-based PKIs, unforgeability follows easily from the corresponding property of the underlying signature scheme; the property may not follow so simply for some other PKIs.

A challenge in defining security requirements for PKI schemes is that a scheme may satisfy a requirement only under specific models (assumptions), including *adversary, communication, synchronization* and other models. In particular, both PKIX and CT require some level of clock synchronization, to ensure consistent interpretation of validity periods. To ensure transparency, CT also requires *benign loggers* as well as reliable communication with bounded delays[2].

To meet this challenge, we use the *Modular Security Specifications (MoSS) framework* [16], which allows the PKI security requirements to be applied under diverse, rigorously-defined adversary, synchronization, network and other models (assumptions).

We present pseudocode to rigorously define two PKI schemes, which are (minimally simplified[3] versions of) the most well-known and widely-deployed PKI schemes: PKIX (X.509 version 3 with CRL) and CT (PKIX extended to support certificate transparency). We then rigorously prove security requirements satisfied by these schemes (see Table 1).

Out of scope: selection of trust anchors and certificate-chain constraints. 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 Mauer [29], subsequently extended by [27, 6], and others [17, 26, 46, 39, 5, 19]. These solutions are complementary and orthogonal to our results.

Contributions. This work:

(1) Presents the first definition of a (non-trivial) PKI scheme, e.g., supporting certificate chains.
(2) Presents the first rigorous security requirements for PKI schemes, including *accountability*, *revocation*, *transparency* and *unforgeability*.
(3) Presents the first precise and complete specification, analysis and proof of security for the two most widely-deployed PKI standards, PKIX and CT.
(4) Introduces and constructs a *certificate scheme* (Section 2), an abstraction for applying signatures to structured information. Certificate schemes simplify definition and analysis of PKI schemes, and may have additional applications.

Organization. In Section 2 we define *Certificate schemes* and their security. In Section 3 we define a PKI scheme and in Section 4 we define security requirements for PKI schemes. In Section 5 we present the specifications of PKIX and CT, and in Section 6 we analyze their security. Finally, Section 7 concludes and discusses future work.

2 CERTIFICATE 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*. Different PKIs may certify different information (fields), use different encodings and 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'.

In this section, we define *certificate schemes*, which provide an abstract definition of a certificate. Our definition allows a uniform treatment of different types of certificates and certified information.

[2] CT includes some, but insufficient defenses against rogue loggers. Version 1 of the CT specifications [34] incorrectly states that the public logs can be *untrusted*; the CT 2.0 specifications [36] states that loggers should be *trusted* third-parties.

[3] Obviously, we could not include all aspects of the PKIX and CT specifications, but we believe that we included all or most of the significant aspects. Our most significant simplification is that we focus on the standardized CRL revocation mechanism, while current browsers mostly use proprietary revocation mechanisms. It appears feasible to extend our analysis to cover these and other aspects which we simplified, although, realistically, this may require automation. Note also that PKI standards are often ambiguous and not fully defined; for example, both CT specifications [34, 36] mention a gossip mechanism, but do not specify it. Hence, specifications can be implemented in different ways, and possibly some implementations will be insecure while others may be secure.

This abstraction allows us to define *PKI schemes* and their security requirements by having each PKI scheme use a specific certificate scheme.

2.1 Certificate Scheme

Definition 1 (Certificate scheme). *A* certificate scheme C *is a set of four PPT algorithms:*

$$C = (\mathsf{KG}, \mathsf{Certify}, \mathsf{Verify}, \mathsf{Extract})$$

where:

- $\mathsf{KG}(1^*) \to (\{0,1\}^*, \{0,1\}^*)$*:* key generation algorithm. *Returns a pair* (sk, pk) *of keys, where* sk *is a private signing (certification) key and* pk *is a public verification key.*
- $\mathsf{Certify}_{sk}(tbc) \to \psi$*:* certification (signing) algorithm. *Returns a binary string* ψ *which is called the* certificate *of the input,* tbc *(*to be certified*), signed using the private certifying (signing) key* sk*. The input* tbc *is a set of pairs,* $tbc = \{(f_1, v_1), (f_2, v_2), \ldots\}$*, where the first element of each pair* (f_i) *is called the* field name *and the second element* (v_i) *is called the* field value.
- $\mathsf{Verify}_{pk}(\psi) \to \{\top, \bot\}$*:* certificate verification algorithm. *Returns* $\top$ *if* ψ *was correctly signed according to the public verification key* pk*, otherwise, outputs* $\bot$*.*
- $\mathsf{Extract}(\psi, \textsc{field}) \to \{0,1\}^* \cup \{\bot\}$*:* field value extraction algorithm. *Returns the value of the inputted field name* FIELD *in* ψ*.*

We say that C *ensures* correctness *if for every* $(sk, pk) \leftarrow \mathsf{KG}(1^n)$ *and for every set of name-value pairs* tbc*, the following two correctness properties hold:*

(1) Verification correctness:

$$\mathsf{Verify}_{pk}(\mathsf{Certify}_{sk}(tbc)) = \top \tag{1}$$

(2) Extraction correctness: *Let* $\psi \leftarrow \mathsf{Certify}_{sk}(tbc)$*. Then:*

$$\mathsf{Extract}(\psi, field) = \begin{cases} val & if\ (field, val) \in tbc \\ \bot & otherwise \end{cases} \tag{2}$$

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,* $\Pr[ECF(n)]$*, is negligible, where*

$$ECF(n) \equiv \left(\begin{array}{c} \mathsf{Verify}_{pk}(\psi) = \top, \textit{ where:} \\ (sk, pk) \leftarrow \mathsf{KG}(1^n), \\ \psi \leftarrow \mathcal{A}^{\mathsf{Certify}_{sk}(\cdot)}(pk), \textit{ and} \\ \mathcal{A} \textit{ didn't receive } \psi \textit{ from } \mathsf{Certify}_{sk}(\cdot) \end{array} \right) \tag{3}$$

We say that C *is* secure *if it ensures both correctness and existential unforgeability.*

2.2 Security of Simple Certificate Schemes

All existing PKIs use a simple certificate scheme which applies a public-key signature to an encoding of the input sequence of name-value pairs (tbc). Let us define such schemes and prove their security.

Definition 2. *Let* Θ *be an invertible function from sequences of name-value pairs to binary strings, and let* $\mathcal{S} = (\mathcal{S}.\mathsf{KG}, \mathcal{S}.\mathsf{Sign}, \mathcal{S}.\mathsf{Verify})$ *be an (existentially-unforgeable) signature scheme. The* $C^{\mathcal{S},\Theta}$ *certificate scheme is defined as:*

$$\begin{aligned} C^{\mathcal{S},\Theta}.\mathsf{KG}(1^n) &\equiv \mathcal{S}.\mathsf{KG}(1^n) \\ C^{\mathcal{S},\Theta}.\mathsf{Certify}_{sk}(tbc) &\equiv \Theta(\{(\text{`}tbc\text{'}, tbc), (\text{`}\sigma\text{'}, \mathcal{S}.Sign_{sk}(\Theta(tbc)))\}) \\ C^{\mathcal{S},\Theta}.\mathsf{Verify}_{pk}(\lambda) &\equiv \mathcal{S}.\mathsf{Verify}_{pk}(\Theta^{-1}(\lambda)[\text{`}tbc\text{'}], \Theta^{-1}(\lambda)[\text{`}\sigma\text{'}]) \\ C^{\mathcal{S},\Theta}.\mathsf{Extract}(\lambda, f) &\equiv \begin{cases} val & if\ (f, val) \in \Theta^{-1}(\lambda)[\text{`}tbc\text{'}] \\ \bot & otherwise \end{cases} \end{aligned}$$

Lemma 1. *If* $\mathcal{S}$ *is an existentially-unforgeable signature scheme, and* Θ *is an invertible function, then* $C^{\mathcal{S},\Theta}$*, defined in Definition 2, is a secure certificate scheme.*

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

Dot notation for schemes and certificates. We use dot notation for the functions of a scheme, above and elsewhere, to avoid ambiguity when referring to functions of different schemes. We also use dot notation for fields in a certificate, i.e., instead of referring to the value val of field f in certificate ψ by the cumbersome notation $\mathsf{Extract}(\psi, f)$, we use the compact notation $\psi.f$. Namely, $\psi.f = \mathsf{Extract}(\psi, f)$, and specifically, $\psi.f = \bot$ if the certificate ψ does not include a name-value pair for a field name f.

2.3 Common Certificate Fields

PKI standards, most notably X.509 and PKIX, define and refer to specific named values in certificates. We refer to all of these named values as 'fields'. We list below some of the important fields[4], to which we also refer in our security requirements.

- $\psi.issuer$: the entity that issued the certificate. In X.509, this is encoded as a *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. In X.509, this is encoded in the *notBefore* entry of the *Validity* certificate field.
- $\psi.to$: the date and time at which the certificate should expire (become invalid). In X.509, this is encoded in the *notAfter* entry of the *Validity* certificate field.
- $\psi.subject$: an identifier for the subject of ψ. In X.509, this is encoded as a *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. In X.509, the subject public key is one component of the *Subject Public Key Info* certificate-field.
- $\psi.is\_CA$: a Boolean which is true if the issuer authorizes the subject to be a CA, i.e., to issue public key certificates. In X.509, this is a part of the *Basic Constraints* extension.
- $\psi.type$: the *certificate type*. This field is not explicitly defined in X.509; see discussion below.

[4] We use the term 'field' for all of these named values, and have abstracted away the encoding. However, specific PKIs, in particular, X.509, may use different terms in addition or instead of 'field', and may use different encodings for different fields. For example, X.509 uses the term 'field' for the *subject* and *issuer* fields (of the certificate) as well as for the *pathLenConstraint* field (which is a part of the Basic Constraints certificate extension). X.509 also uses other terms for named values, e.g., it refers to the *dnsName component* (in the *subjectAltName* extension) and to the *cA boolean* (in the BasicConstraints extension).

2.4 Types of Certificates

The aforementioned common certificate fields mostly correspond to certificate fields defined by X.509. One important exception is the *type* field, which is not one of the X.509 fields. We use the *type* field to distinguish between *different types of certificates*. X.509 explicitly refers to two types of certificates: *public key certificates* and *attribute certificates*, which we identify by $\psi.type = PubKey$ (for public key certificates) or $\psi.type = ATTR$ (for attribute certificates). Public key certificates are signed by a certificate authority (CA), and 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 $\psi.type = ATTR$, then $\psi.pk = \perp$).

Note that our definition of a certificate can be applied also to other types of signed information consisting of multiple name-value pairs. Important, standardized examples include:

- *Certificate Revocation Lists (CRLs),* used to identify revoked and non-revoked certificates. CRLs are certificates since they are signed (certified) objects with multiple fields (e.g., validity period and issuer).
- *Signed Tree Head (STH).* Certificate Transparency (CT) uses *Signed Tree Head (STH)* certificates, signed (certified) by the CT loggers.
- *Route Origin Authorizations (ROAs)*, defined in the Resource Public Key Infrastructure (RPKI) [32]. ROAs are signed (certified) by the (certified) owner of a routing resource such as IP prefix or Autonomous System Number (ASN), and include multiple fields, e.g., validity period and issuer.

3 PKI SCHEMES AND TRANSPARENT PKI SCHEMES

We define *PKI schemes* and *transparent PKI schemes*, which are PKI schemes with additional entities and functions used to ensure transparency (§3.5). These sections provide necessary background: *basic PKI entities* (§3.1) and basic PKI functions, for evaluating the *validity* of a certificate (§3.2), for handling *certificate chains* (§3.3) and for certifying (issuing) and revoking certificates (§3.4).

3.1 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 can be initialized using a dedicated *initialization* operation, denoted as $\mathcal{P}.\mathsf{Init}$. Typically, $\mathcal{P}.\mathsf{Init}$ 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 key $\psi.pk$.

3.2 Certificate Validity

We determine the validity of a certificate ψ using the stateless *certificate validation predicate* $\mathcal{P}.\mathsf{Valid}(clk, \psi, store, \omega)$ (Definition 4). An entity evaluates validity with respect to its current clock value clk, the set $store$ of self-certified *trust-anchor* certificates and auxiliary information ω.

In existing PKIs, validation uses the $C.\mathsf{Verify}$ operation of an underlying certificate scheme, typically, by establishing trust in a public validation key pk of $\psi.issuer$. This usually uses *certificate chains* from ω, introduced next. We use chains to establish trust in pk (from the trust-anchor certificates in $store$), and validate that ψ and certificates in the chains were not revoked, e.g, using a CRL (§3.4).

3.3 Certificate Chains

We now define *certificate chains*[5], which allow validation of certificates signed by a non-root CA, i.e., a CA which does not have a self-signed certificate in $store$.

Definition 3 (Certificate chain). *Let $\xi = \{\psi_0, \psi_1, \ldots\}$ be a sequence of certificates. We say that ξ is a* certificate chain *for certificate scheme C if for every issued certificate ψ_{i-1} and its issuer certificate ψ_i, for $0 < i < |\xi|$, holds:*

1. *Correct certification: $C.\mathsf{Verify}_{\psi_i.pk}(\psi_{i-1}) = \top$.*
2. *The issuer of ψ_{i-1} is the subject of the issuer certificate ψ_i, i.e., $\psi_{i-1}.issuer = \psi_i.subject$.*
3. *The issuer certificate is a public-key certificate for a CA, i.e.: $\psi_i.is\_CA = \top$ and $\psi_i.type = PubKey$.*

See pseudocode in Algorithm 1.

Algorithm 1 Certificate chain predicate

Input: certificate chain ξ.
Output: $\top$ if ξ is a well structured chain or $\perp$ otherwise.

```
1: procedure IsChain^C(ξ)
2:     return ∀i : 0 < i < |ξ| :                          ▷ All non-first certs in the chain
3:                 C.Verify_{ξ[i].pk}(ξ[i − 1]) ∧          ▷ Are properly certified
4:                  ξ[i − 1].issuer = ξ[i].subject ∧       ▷ Are for issuer of prev cert
5:                  ξ[i].type = PubKey ∧                   ▷ Are PubKey certs
6:                  ξ[i].is_CA                             ▷ Are for a CA
7: end procedure
```

The stateless operation $\mathcal{P}.\mathsf{Chains}(clk, \psi, store, \omega) \rightarrow \{\xi_i\}$ identifies one or more chains used to validate a given certificate ψ. The certificate chains returned by $\mathcal{P}.\mathsf{Chains}$ are used to ensure *accountability*, as defined in §4.3.

3.4 Certifying and Revoking Certificates

A certificate ψ is issued using the private certification key of a CA, using the PKI's $\mathcal{P}.\mathsf{Certify}$ 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 data to be certified tbc, and outputs an updated state st, and a signed certificate ψ if successful or $\perp$ otherwise. Typically, $\mathcal{P}.\mathsf{Certify}$ uses the certify function $C.\mathsf{Certify}_{sk}(tbc)$ of an underlying certificate scheme C (see Definition 2) where tbc is

[5]The term 'chain of trust' is also sometimes used.

the information to be certified using the entity's private key sk extracted from its local state st.

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.: $\mathcal{P}.\text{Revoke}(\text{st}, clk, \psi) \rightarrow (\text{st}, \top/\bot)$. For example, $\mathcal{P}$.Revoke would return $\bot$ if attempting to revoke an already expired or revoked certificate, or a certificate not issued by this issuer.

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

3.5 Definitions of PKI Schemes

Definition 4 (PKI scheme). *A PKI scheme $\mathcal{P}$ is a set containing (at least) the following PPT algorithms:*

- $\mathcal{P}.\text{Init}(\text{st}, clk, params) \rightarrow (\text{st}, \psi)$*: Takes as input the state*[6] st*, local clock* clk*, and parameters* $params$*, and returns the initialized local state* st *after performing initialization based on the input parameters* $params$ *on time* clk *and a self-certified certificate* ψ*.*
- $\mathcal{P}.\text{Certify}(\text{st}, clk, tbc) \rightarrow (\text{st}, \psi/\bot)$*: Takes as input the state* st*, local clock* clk *and information to be certified* tbc*, and outputs an updated state* st *and either a certificate* ψ *or a failure indicator* $\bot$*.*
- $\mathcal{P}.\text{Revoke}(\text{st}, clk, \psi) \rightarrow (\text{st}, \top/\bot)$*: Takes as input the state* st*, local clock* clk*, and a certificate* ψ*, and outputs an updated state* st *and* $\top$ *if* ψ *was revoked successfully or* $\bot$ *if the revocation failed.*
- $\mathcal{P}.\text{Valid}(clk, \psi, store, \omega) \rightarrow (\top/\bot)$*: This (stateless) algorithm takes as input local clock* clk*, a certificate* ψ*, a root store* $store$*, and auxiliary information* ω*, and outputs either* $\top$ *or* $\bot$*.*
- $\mathcal{P}.\text{Chains}(clk, \psi, store, \omega) \rightarrow \{\xi_i\}$*: This (stateless) algorithm receives the same inputs as* $\mathcal{P}$.Valid*, and, if* ψ *is valid, returns one or more certificate chains* ξ_i*.*

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 will be publicly available (i.e., "transparent"), allowing to detect discrepancies and suspect certificates.

Definition 5 (Transparent PKI scheme). *A transparent PKI scheme $\mathcal{P}$ is a PKI scheme with the following additional PPT algorithms:*

- $\mathcal{P}.\text{Monitor}(\text{st}, clk, \iota) \rightarrow \text{st}$*: Takes as input state* st*, local clock* clk *and entity identifier* $\iota \in \mathsf{N}$*, and outputs an updated state* st*, and it starts to monitor (certificates* logged *by)* ι*.*
- $\mathcal{P}.\text{Lookup}(\text{st}, clk, subject) \rightarrow (\text{st}, \Psi)$*: Takes as input state* st*, local clock* clk*, and an identifier* $subject$*, and outputs an updated state* st *and a set* $\Psi = \{(\psi_1, \omega_1), (\psi_2, \omega_2), ...\}$ *of all pairs* (ψ_i, ω_i) *known to the entity s.t.* ψ_i *is the certificate of the given* $subject$*, i.e.,* $\psi_i.subject = subject$*, and* ω_i *is the corresponding auxiliary information for* ψ_i*.*

[6] The state st is given in the input for a technical reason: the execution process treats Init like other operations, and therefore, also passes the state st to it. Typically, the input state is ignored.

4 PKI REQUIREMENTS

In this section we define the requirements that a PKI scheme should satisfy. Formally-defined requirements are necessary to (formally) prove whether or not a given PKI implementation satisfies specific requirements, and if so, under what assumptions.

4.1 Modular Security Specifications

PKI schemes, both 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* [16] for our PKI specifications. MoSS separates the definition of *requirements* from the *models*, which simplifies the definition of requirements and allows evaluating the requirements satisfied by PKI schemes under different models. For example, we show that CT provides transparency under a model that assumes honest loggers, but not under a model that allows corrupt loggers (Table 1).

To illustrate the importance of separating models from requirements, consider the following *simplified* Δ-Revocation requirement: a certificate revoked at time clk by an honest CA would be considered invalid at any time after $clk + \Delta$. 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 delay models.

For completeness, we recall here and in Appendix A.1 concepts of MoSS [16] used in this work.

MoSS defines an execution process $\mathbf{Exec}_{\mathcal{A},\mathcal{P}}(params)$ in which an adversary $\mathcal{A}$ has complete control over the environment, the execution of a protocol $\mathcal{P}$ and all external inputs (parameters) of the protocol ($params$). The output of the execution of protocol $\mathcal{P}$ with parameters $params$ is called a *transcript*, denoted as $T \leftarrow \mathbf{Exec}_{\mathcal{A},\mathcal{P}}(params)$. A transcript contains the set of entities $T.\mathsf{N}$ and the subset of faulty entities $T.\mathsf{F} \subseteq T.\mathsf{N}$. It also contains, for each step in the execution, the operation invoked, the inputs and the outputs of the protocol, the output of the adversary $T.out_{\mathcal{A}}$, and other relevant values of the execution. To illustrate, given a transcript łogm and certificate ψ, we can check if there is an event e in T in which the adversary invoked the $\mathcal{P}$.Certify operation on entity $\psi.issuer$ with input $\psi.tbc$ and received ψ as output by using the following predicate:

$$\exists e \textbf{ s.t. } T.opr[e] = \text{'Certify'} \wedge T.ent[e] = \psi.issuer \wedge T.inp[e] = \psi.tbc \wedge T.out[e] = \psi$$

A MoSS model $\mathcal{M}$ is defined using an associated predicate $\pi_{\mathcal{M}}$, and similarly, a MoSS requirement $\mathcal{R}$ is defined using an associated predicate $\pi_{\mathcal{R}}$. The predicates are applied to the transcript of an execution of a protocol and define if the execution satisfied the requirement or model. For each model and predicate, we can define

a base function $\beta(params)$, applied to the parameters $params$, to allow some probability for an execution to not satisfy the model or requirement. In this work we omit β and write (π) as a shorthand for (π, β), since our models and requirements have $\beta(params) = 0$.

A PPT adversary $\mathcal{A}$ *satisfies model* $\mathcal{M} = (\pi_{\mathcal{M}})$, if the probability that $\pi_{\mathcal{M}}(T) = \perp$ is negligible, for a random transcript of any protocol interacting with $\mathcal{A}$. A protocol $\mathcal{P}$ *satisfies requirement* $\mathcal{R} = (\pi_{\mathcal{R}})$ *given model* $\mathcal{M}$, if the probability that $\pi_{\mathcal{R}}(T) = \perp$ is negligible for a random transcript of $\mathcal{P}$ interacting with any PPT adversary $\mathcal{A}$ that satisfies model $\mathcal{M}$.

The rest of this section defines several PKI requirements: the existential unforgeability requirement in §4.2, the accountability requirement in §4.3, two versions (Weak and Full) of the Δ-Revocation requirement in §4.4, and the Δ-Transparency requirement in §4.5.

4.2 The Existential Unforgeability Requirement

The existential unforgeability requirement of PKI schemes is similar to the corresponding property for signature and certificate schemes (Definition 1). It ensures that every certificate ψ issued by an honest entity ι and successfully validated using ι's public key, was issued following a request to do so, i.e., ι invoked $\mathcal{P}$.Certify to produce ψ. The only exception are *self-signed certificates*, i.e., certificates whose *issuer* and *subject* fields are the same *and* certify the same public key used to validate them. CAs *self-certify* their public keys after generating them (typically, at initialization). The unforgeability requirement ensures that honest entities will not be considered as issuers for certificates that they did not issue. Unforgeability typically follows easily from the fact that certificates are issued by a secure certificate scheme; see Lemma 2.

Definition 6 (PKI Existential Unforgeability). *The PKI existential unforgeability requirement* $\mathcal{R}_{\mathsf{EUF}}$ *is defined as* $\mathcal{R}_{\mathsf{EUF}} = (\pi_{\mathsf{EUF}})$*, where* π_{EUF} *is defined in Algorithm 2.*

The following lemma shows that both PKIX and CT ensure existential unforgeability under the $\mathcal{M}^{\mathsf{F}} = (\pi^{\mathsf{F}})$ model. This model, defined in [16], ensures that all entities follow the protocol correctly, except for entities in T.F (the set of faulty entities outputted by the adversary).

Lemma 2. *The PKIX and CT PKI schemes, presented in §5, ensure existential unforgeability under the* $\mathcal{M}^{\mathsf{F}}$ *model, if they use a secure (existentially unforgeable) certificate scheme* C.

Proof: In both PKIX and CT, non-faulty entities (1) output a self-signed certificate only using a public key generated by the entity using C.KG (in their Init operations), (2) never output the corresponding private key, and (3) only apply the corresponding private key, using C.Certify, to certify $\psi.tbc$. The claim follows by reduction to the unforgeability of C. □

4.3 The Accountability Requirement

Intuitively, we[7] use the term *accountability* for the ability to *identify*, for every certificate ψ_0, an *accountable* (responsible) *trust anchor* for ψ_0. We identify the accountable trust anchor by a chain[8] of valid certificates leading to a self-signed certificate in $store$, which provides *non-repudiation*. Accountability is an important PKI requirement; it provides a (reactive) defense by identifying a root CA that is accountable for rogue yet valid certificates, regardless of whether the CA acted intentionally or negligently. A root CA which is found accountable for a rogue certificate, once or repeatedly, may be removed from root stores or otherwise penalized.

To identify a CA accountable for a valid ψ, we use the $\mathcal{P}$.Chains function to extract the set of auxiliary information ω used to validate ψ. In an accountable PKI, ω contains at least one *valid certificate chain* ξ that begins with ψ (i.e., $\xi[0] = \psi$). We say that a ξ is a valid certificate chain, if it contains a (self-signed) trust anchor $\xi[i] \in store$, and the certs from $\xi[0]$ to $\xi[i-1]$ are valid and form a chain (Definition 3); see the IsValidChain function in Algorithm 3. The issuer (which is also the subject) of the trust anchor[9] $\xi[i]$ is *accountable* for ψ.

We express accountability as a predicate π_{ACC} (Algorithm 3) and use it to define the accountability requirement in Definition 7.

Definition 7 (Accountability). *The accountability requirement* $\mathcal{R}_{\mathsf{ACC}}$ *is defined as* $\mathcal{R}_{\mathsf{ACC}} = (\pi_{\mathsf{ACC}})$*; see Alg. 3.*[10]

4.4 The Weak and Full Δ-Revocation Requirements

Most PKIs allow a certificate to be revoked by its issuer, i.e., invalidated prior to its specified expiration date. Revocation may be used

Algorithm 2 π_{EUF}: Existential Unforgeability Requirement

Input: transcript T.
Output: $\top$ if for the certificate ψ outputted by the adversary in the execution transcript T, where $\psi.issuer$ is an identifier of a benign entity and ψ is valid with respect to a public key self-signed by $\psi.issuer$, holds that $\psi.issuer$ was requested to certify $\psi.tbc$. Otherwise, $\perp$.

1: **procedure** $\pi_{\mathsf{EUF}}(T)$
2: $clk, \psi, store, \omega \leftarrow T.out_{\mathcal{A}}$ ▷ *Extract adversary's output*
3: **if** CertifyRequested(T, ψ) **then return** $\top$ ▷ *Requested, i.e., not forged!*
4: **If**
$\mathcal{P}.\mathsf{Valid}(clk, \psi, store, \omega) \wedge$ ▷ *ψ is valid*
$\psi.issuer \in T.\mathsf{N} - T.\mathsf{F} \wedge$ ▷ *ψ's issuer is a benign entity*
$(\forall \xi \in \omega$ **s.t.** $\xi[0] = \psi :$ ▷ *All chains by honest CAs in ω*
SelfCertified$(T, \psi.issuer, \xi[1].pk)) \wedge$ ▷ *to validate ψ are correct*
$(\forall \psi' \in store$ **s.t.** $\psi'.subject = \psi.issuer :$ ▷ *All honest root CAs' keys*
SelfCertified$(T, \psi'.subject, \psi'.pk))$ ▷ *in store are real*
5: **then return** $\perp$ ▷ *Forgery! Adversary wins*
6: **end if**
7: **return** $\top$ ▷ *Adversary failed*
8: **end procedure**
9: **procedure** CertifyRequested(T, ψ)
10: **return** $\exists e$ **s.t.** $T.opr[e] =$ *'Certify'* $\wedge$ ▷ *Certify operation invoked*
11: $T.ent[e] = \psi.issuer \wedge$ ▷ *on $\psi.issuer$*
12: $T.inp[e] = \psi.tbc$ ▷ *for certificate ψ*
13: **end procedure**
14: **procedure** SelfCertified(T, ι, pk)
15: **return** $\exists e, \psi'$ **s.t.** $T.ent[e] = \iota \wedge$ ▷ *Entity ι*
16: $T.out[e] = \psi' \wedge$ ▷ *outputted certificate ψ'*
17: $\psi'.pk = pk \wedge$ ▷ *certifying given pk,*
18: $C.\mathsf{Verify}_{pk}(\psi') \wedge$ ▷ *signed by same pk, and*
19: $\psi'.issuer = \psi'.subject = \iota$ ▷ *using the same identity*
20: **end procedure**

[7] We do not use the term accountability in a legal sense.

[8] Stronger accountability requirements are possible, e.g., requiring at least two 'independently-accountable' chains to different root CAs, or allowing only limited-length of certificate chains.

[9] Typically, but not necessarily, the trust anchor $\xi[i]$ is the last certificate in ξ.

[10] Line 5 of π_{ACC} checks $\mathcal{P}$.Valid after checking IsChain(ξ). Why? See App. D.

Algorithm 3 π_{ACC}: Accountability requirement

Input: transcript T.

Output: $\top$ if the adversary loses, i.e., does not output a valid certificate, or outputs a cert ψ for which the PKI identifies a valid certificate chain which contains an accountable CA (in $store$). Otherwise, i.e., if adversary wins, output $\perp$.

procedure $\pi_{\text{ACC}}(T)$
  $(clk, \psi, store, \omega) \leftarrow T.out_{\mathcal{A}}$ ▷ *Adversary's output*
  return $\neg \mathcal{P}.\text{Valid}(clk, \psi, store, \omega) \vee$ ▷ *Certificate is not valid*
    $(\exists \xi \in \mathcal{P}.\text{Chains}(clk, \psi, store, \omega),\ j < |\xi|)$ **s.t. :** ▷ *Or, for some chain ξ and $j < |\xi|$:*
    $\left(\begin{array}{c} \text{IsChain}(\xi) \ \wedge\ \xi[j] = \psi \ \wedge \\ \mathcal{P}.\text{Valid}(clk, \xi[j+1], store, \omega) \end{array} \right)$ ▷ *which contains ψ and whose next cert, $\xi[j+1]$, is valid*
end procedure

for a variety of reasons, including loss or compromise of the private key corresponding to a certified public key, or when the issuer determines that the certificate contains incorrect information. We focus on two variants of the basic requirement for revocation which we call the *Weak* and *Full* Δ-Revocation requirements. We first discuss the stronger and simpler Full Δ-Revocation requirement.

Full Δ-Revocation. The requirement ensures that a certificate ψ issued by a benign CA $\psi.issuer$, and revoked by $\psi.issuer$ at time clk, will not be considered as valid at any time following $clk + \Delta$. 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 the $\psi.issuer$ to the relying parties, and (3) the time for any proof that ψ is non-revoked that was issued prior to its revocation becomes stale, i.e., expire. Full Δ-Revocation ensures that the revocation operation by a benign CA is always effective (after at most Δ); a relying party cannot be misled to rely on a revoked certificate (for more than Δ).

A natural (but unfortunately not sufficient) approach to ensure Full Δ-Revocation is to validate ψ using a *non-revocation certificate* $\psi_{NR} \in \omega$ that certifies that ψ was not revoked until a given time $\psi_{NR}.\tau$; as a result, the certificate will be considered by relying parties as valid until time $\psi_{NR}.\tau + \Delta$. In X.509 and PKIs based on it, including PKIX and CT, the non-revocation certificate[11] ψ_{NR} must indicate the same issuer as of ψ, i.e., $\psi_{NR}.issuer = \psi.issuer$.

However, this design fails to ensure the Full Δ-Revocation requirement. We present the *Zombie certificate attack*, where an attacker is able to cause ψ to be valid at time $t \in [\psi.from, \psi.to]$ although it was revoked by a benign issuer $\psi.issuer$.

Claim 3 (The Zombie certificate attack). *PKIX and CT, using CRLs or OCSP for revocation, fail to ensure the Full Δ-Revocation requirement.*

Proof (informal). We present the proof for PKIX using CRLs, but it applies with trivial changes also to CT and when using OCSP for revocation. The proof is by a counter example: the *Zombie-certificate attack*.

Let ψ be a certificate issued by a benign CA $\psi.issuer$, with validity period $[\psi.from, \psi.to]$, which is later revoked by $\psi.issuer$, at time t_R s.t. $\psi.from < t_R < \psi.to - \Delta$. We show how an attacker is able to certify a fake-yet-valid CRL which indicates that ψ was not revoked and is still valid during $[t_R, \psi.to]$, i.e., after ψ was revoked by the benign issuer $\psi.issuer$.

[11] X.509 defines two types of non-revocation certificates: the *Certificate Revocation List (CRL)* [9] and the *Online Certificate Status Protocol (OCSP)* [33].

The attacker controls a rogue CA $\iota_R \in T.\mathsf{F}$ which is able to issue valid CA certificates. This may be the root CA that is the trust anchor of ψ's certificate chain, or an intermediate CA that has a certificate chain rooted by the same trust anchor of ψ's[12]. The attacker generates a keypair $(sk_R, pk_R) \leftarrow C.\text{KG}(1^n)$ and then certifies a fake CA-certificate ψ_R for $\psi.issuer$, i.e., $\psi_R.pk = pk_R$ (which, of course, is not the public key used by the benign $\psi.issuer$, e.g., $C.\text{Verify}_{\psi_R.pk}(\psi) = \perp$). Let ω_R denote the auxiliary information required to validate ψ_R, including a valid certificate chain ξ_R from a root CA. If ι_R is a root CA, then ξ_R will contain only ψ_R and the self-signed certificate of ι_R.

The attacker then uses sk_R to certify CRL_R, a fake-yet-valid CRL, where ψ is *not* marked as revoked. Hence, ψ would be considered valid during $[t_R, \psi.to]$ when provided together with the (fake yet valid) CRL CRL_R and with ω_R (containing the (misleading yet valid) certificate chain ξ_R, validating the fake CA-certificate C_R). Namely, the scheme fails to satisfy the Full Δ-Revocation requirement. □

Notes: (1) A naive 'fix' is to require the non-revocation certificate (CRL/OCSP) to be signed by the same key used to certify ψ. However, this would make it impossible for a benign issuer $\psi.issuer$ to change its key if necessary, e.g., due to exposure. (2) Short-lived certificates are a simple way to ensure Full, using $\psi.to - \psi.from \leq \Delta$.

Weak Δ-Revocation. This requirement *allows* ψ to be considered valid even after being revoked by its benign CA $\psi.issuer$, provided that the PKI can identify a valid-yet-rogue CA certificate ψ_R that is responsible for ψ being valid after it was revoked. If the PKI can identify the rogue CA certificate C_R, and also ensures *accountability*, then we can identify a root CA which is accountable for the ψ_R. We identify ψ_R as a rogue CA certificate, since it certifies public key $\psi_R.pk$ for $\psi.issuer$, although the benign $\psi.issuer$ did not output a corresponding self-signed certificate, i.e., it didn't generate $\psi_R.pk$.

We define both Full and Weak Δ-Revocation requirements in Definition 8, using the function $f_{\Delta\text{Rev}}$ presented in algorithm 25. In §6 we show that PKIX and CT ensure Weak Δ-Revocation.

Definition 8 (Δ-Revocation requirements). *We define the Full and Weak Δ−revocation predicates as:*

$$\pi_{Full-\Delta\text{Rev}}(T) = \left\{ \begin{array}{c} \top \text{ if } f_{\Delta\text{Rev}}(T) = \top, \\ \perp \text{ otherwise} \end{array} \right\}$$

$$\pi_{Weak-\Delta\text{Rev}}(T) = \left\{ \begin{array}{c} \perp \text{ if } f_{\Delta\text{Rev}}(T) = \perp, \\ \top \text{ otherwise} \end{array} \right\}$$

And define the Full and Weak Δ-Revocation requirements as:

$$\mathcal{R}_{Full-\Delta\text{Rev}} = (\pi_{Full-\Delta\text{Rev}})$$
$$\mathcal{R}_{Weak-\Delta\text{Rev}} = (\pi_{Weak-\Delta\text{Rev}})$$

4.5 The Δ-Transparency Requirement

Accountability, as described in §4.3, is a *deterrent* against rogue or negligent behavior by root CAs. For many years this reactive measure was viewed as a sufficient defense, under the assumption that root CAs, and CAs certified by them, were respectable and trustworthy entities who would not risk, intentionally or otherwise, being implicated in issuing rogue certificates. However, repeated cases of rogue certificates issued by compromised or dishonest CAs,

[12] RFC5280 states that "The trust anchor for the certification path MUST be the same as the trust anchor used to validate the target certificate".

have proven this assumption to be overly optimistic. It turned out that punishing root CAs is non-trivial: beyond negative publicity, punishment was often ineffective [37, 1, 18].

Furthermore, 'punishment' 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 [44] and the EFF SSL Observatory[11] provide some assistance to the discovery of rogue certificates, but were not sufficient to address this concern.

This motivated *transparent* PKI designs, most notably, the standardized and deployed *Certificate Transparency (CT)* [23, 25]. Certificate transparency requires a certificate ψ to be certified by one or more *loggers*. Loggers are parties committed to include (transparent) certificates in a *public log* they maintain, and to make 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 suspect certificates and inform interested parties, such as domain owners and relying parties.

In other words, transparency prevents a CA from 'quietly' generating a rogue certificate ψ, and using 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 suspect certificates.

There is some unavoidable delay from the time that a certificate is submitted to the log and until the relevant monitors are aware of it. This delay is mainly due to significant allowed delay from receiving a certificate and until including it in a new signed log (STH).

Informally, a PKI scheme $\mathcal{P}$ satisfies the Δ-Transparency requirement if no adversary $\mathcal{A}$ can produce, with non-negligible probability, a pair (ψ, ω) where ψ is a certificate *valid* w.r.t. the auxiliary information ω on time clk, and yet, there exists a benign monitor ι_M which is unaware of ψ, even though ι_M monitors a log in the set of logs $\psi.logs$ in which ψ is (allegedly) published, prior to $\psi.\tau$.

We express this informal definition as a predicate $\pi_{\Delta_{\text{Tra}}}$ (Algorithm 26), and present the formal definition of Δ-Transparency in Definition 9.

Definition 9 (Δ-Transparency). *The Δ-Transparency requirement $\mathcal{R}_{\Delta_{\text{Tra}}}$ is defined as $\mathcal{R}_{\Delta_{\text{Tra}}} = (\pi_{\Delta_{\text{Tra}}})$, where $\pi_{\Delta_{\text{Tra}}}$ is defined in Algorithm 26.*

5 PROVABLY-SECURE PKI SCHEMES

In this section, we present the design of two important PKI schemes: PKIX and CT, where our design of CT is based on the CT 2.0 specification [36], using PKIX for aspects not covered in [36]. In both PKIX and CT, we use the CRL revocation mechanism. The full PKIX and CT specifications are far too complex to present and analyze in this paper; as such, we considered a significantly simplified design but we retained all aspects related to security considerations, e.g., by considering certificate chains (although we ignore the length, name and policy constraint extensions).

5.1 PKIX (with CRLs)

Entities and state. In PKIX, all entities (in N) are certificate authorities (CAs), i.e., they issue and revoke public key certificates. Some of these are *root CAs* (trust anchors), which relying parties trust directly (by maintaining their self-signed certificates in $store$), and others 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:

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

Certificate fields and scheme. PKIX_C uses a simple certification scheme C (Definition 2), defined in [31]. The PKIX_C implementation (Algorithms 4-8) uses the common certificate fields mentioned in §2.3, with the addition of the serial number of public key certificates $\psi.serial$.

Auxiliary information. In order for a certificate ψ to be considered valid in PKIX_C, it must have a valid chain-of-trust to a trusted root CA and a CRL that proves that ψ was not revoked. Specifically, the auxiliary information ω should contain the following information:

- $\omega.\psi_{CA}$, a certificate of the CA that issued ψ and $\omega.\omega_{CA}$, auxiliary information required to validate $\omega.\psi_{CA}$.
- $\omega.\psi_{CRL}$, a CRL showing that ψ was not revoked, and $\omega.\omega_{CRL}$, auxiliary information required to validate $\omega.\psi_{CRL}$.

Implementation. We now present the implementation (Algorithms 4-8), according to Definition 4, with the addition of one more operation: PKIX_C.GetCRL, which produces the current CRL that is properly certified (signed). This corresponds to the CRL's distribution point mechanism described in [31].

PKIX_C.Init *(Algorithm 4).* The PKIX Init function generates the CA's keypair (line 3) and self-certifies the public key (line 5). It also initializes the state st of the CA: the CA's identity $\text{st}.\iota$, the $\text{st}.\Delta_r$ parameter (the CRL validity period), both from the $params$ input, and the set of certificates issued ($\text{st}.certs$) and revoked ($\text{st}.CRL$) by the CA, both initially empty.

PKIX_C.Certify *(Algorithm 5).* PKIX certifies only public-key certificates. Therefore, the algorithm sets the certificate's type to be of public key (line 2), and then signs tbc using $C.\text{Certify}(\text{st}, clk, tbc)$ and the CA's secret signing key (line 3). The data to be certified and the signature over it are bundled to form the certificate ψ (line 4). Then, a copy of ψ is stored locally (line 5) and the algorithm outputs ψ along with its auxiliary information ω (line 6).

PKIX_C.Revoke *(Algorithm 6).* The algorithm verifies that the inputted parameters are legitimate, i.e., the certificate to be revoked was indeed issued by the CA (line 1). If so, the algorithm adds the certificate to the list of revoked certificates (line 2).

PKIX_C.Valid *(Algorithm 8).* The (stateless) PKIX.Valid algorithm starts by checking if the inputted certificate ψ is one of the trusted certificates in $store$. If so, the only additional check required is to ensure that ψ did not expire (line 1). Next, the algorithm ensures that the inputted certificate is certified correctly, has a valid certificate chain ξ and both ψ and ξ were not revoked. To that end, the algorithm ensures that (line 2):

(1) The auxiliary information ω contains a certificate chain ξ which starts with ψ.
(2) ξ is well structured and all the certificates are: correctly certified, did not expire and they were not revoked.
(3) ξ contains a trusted certificate, i.e., in $store$.

To check that the certificates in ξ were not revoked, the algorithm ensures that each certificate ψ' has a *valid* CRL in ω which does include ψ' (line 11). Namely, a valid CRL is:

(1) A valid certificate of type 'CRL' (line 9).
(2) Was issued by the same issuer of the certificate (line 10).
(3) The CRL certificate is valid. Since the CRL certificate is a certificate like any other certificate, this is checked using a recursive call to PKIX_C.Valid (line 12).

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.Chains *(Algorithm 7).* In PKIX, all the certificate chains, as well as CRL, are given in the ω input. Therefore the Chains function simply returns the ω input.

PKIX_C.GetCRL *(Algorithm 9).* First, the algorithm removes all the certificates that expired from the current CRL (line 1). Then, it generates an updated CRL, i.e., defines a data to be certified tbc with CRL type (line 2), sets the issuer and validity period (lines 3-5) and the list of revoked certificates (line 6). Finally, the algorithm signs tbc using C.Certify(st, clk, tbc) and the CA's secret signing key (line 7) and outputs the CRL certificate along with the local state (lines 8-9).

5.2 Certificate Transparency (CT)

We now describe Certificate Transparency (CT), an extension to PKIX defined in [34, 36], as a transparent PKI scheme (Definition 5). Specifically, in Algorithms 11-24 we present the CT algorithms, utilizing, as subroutines, the PKIX algorithms (Algorithms 4-8).

Convention: CT uses a few timing parameters (Δ_r, Δ_{MMD}, Δ_{clk} and Δ_{com}), known to all parties. Formally, these should be part of the scheme name, but we omit them to avoid clutter.

Entities and state. In CT, we consider, in addition to certificate authorities (with the same local state information as defined for PKIX), also two additional types of entities (in N): *loggers* and *monitors*. Each logger maintains a log of certificates (st.$mylog$). Each monitor keeps the list of loggers it oversees (in st.$loggers$), and for each of these loggers, the set of certificates they logged (st.log).

SCTs and pre-certificates. CT certificates use the same certification schemes and certificate fields as in PKIX, and adds a new type of certified-objects, the *signed certificate timestamp (SCT)*. In CT, a certificate is valid only together with two[13] valid SCTs, from different loggers. A CA wishing to issue a CT certificate, typically first certifies and submits to loggers, using the CT.AddPreChain-Req operation, a so-called *precertificate*, which is a PKIX-valid[14] certificate. After validating the (pre)certificate, the loggers add it to the log, certify and return the corresponding SCT. We consider the SCT as a field of the CT certificate; the CT.Valid operation requires a valid SCT field[15].

Implementation. Algorithms 11-24 present the implementation of CT. The operations defined include these required from any PKI (Definition 4) and from any transparent PKI (Definition 5), and operations[16] defined in CT specifications [34]: GetSTH-Req, GetSTH-Resp, AddPreChain-Req, GetEntries-Req and GetEntries-Resp. One more operation, not explicitly defined in the CT specifications [34], is CT.Wakeup. The Wake-up operation is required to perform scheduled operations defined in [34], such as updating the log's root signature and periodically retrieving the updated logs by monitors.

To schedule a Wakeup operation at time t with input $data$, the protocol outputs ('Wake-up', t, $data$). The $\pi^{\text{Wake-up}}_{\Delta_{clk}}$ model predicate ensures that a CT.Wakeup operation is invoked, with input $data$, on the entity requesting Wake-up, at real time within window of Δ_{clk} before and after time t, i.e., in $[t - \Delta_{clk}, t + \Delta_{clk}]$.

We describe the implementation, focusing on the differences from PKIX. The revocation mechanism used in PKIX and CT is identical, hence, the CT.Revoke (Algorithm 13) and CT.GetCRL (Algorithm 14) algorithms simply invoke their PKIX counterparts. Similarly, CT.Chains (Algorithm 15) does not introduce any new functionality, and simply invokes its PKIX counterpart.

CT.Init *(Algorithm 11).* The algorithm adds the initialization details for loggers and monitors. First, the algorithm calls PKIX.Init to handle any of PKIX related details (line 1). Then, the algorithm performs additional operations for loggers and monitors. First, the algorithm requests a Wake-up within Δ_{MMD} (line 3). Finally, for monitors, the algorithm initializes an empty set of monitored logs and stores the set of trusted loggers' certificates, while for loggers, the algorithm initializes an empty maintained log (lines 4-9).

CT.Certify *(Algorithm 12).* In addition to the 'regular' certificates issued in PKIX, CT also allows to issue pre-certificates. To that end, CT.Certify checks if the inputted data to be certified contains the 'pre-certificate' type (line 1), and if so, adds the 'poison' extension as described in RFC6962 [34] (line 2). In any case, the certificate is generated using the PKIX.Certify algorithm (line 4).

CT.Valid *(Algorithm 16).* First, use PKIX.Valid to validate that ψ is a PKIX-valid certificate. From line 3, we verify that ψ is submitted with two valid SCTs, issued for ψ, from different trusted loggers (i.e., with certificates from $store.logger$).

CT.Monitor *(Algorithm 17).* The algorithm receives a logger identifier ι and adds it to the list of monitored logs (line 1). As a result, starting from the next time the CT.Wakeup algorithm is invoked, ι will be monitored along with the other loggers.

[13] The decision of which logger-SCTs are sufficient can differ among relying parties; for simplicity, and since it appears irrelevant to our analysis, we assume two SCTs from different loggers are required.

[14] Precertificates may include a critical 'poison' extension that makes them PKIX-invalid, to prevent their use with relying parties that accept PKIX certificates. For simplicity, we ignore this, and consider them PKIX-valid.

[15] The RFC allows the SCT to be encoded as a certificate extension or to be used as a separate data structure in certificate validation.

[16] We kept the operation names from [34].

Algorithm 4 PKIX_C.Init: initialization

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

1: $\text{st}.\Delta_r \leftarrow params.\Delta_r$ ▷ *Save Δ_r (validity interval for CRLs)*
2: $\text{st}.\iota \leftarrow params.\iota$ ▷ *Save identity (ι)*
3: $(\text{st}.sk, \text{st}.pk) \leftarrow C.\text{KG}(params.\kappa)$ ▷ *Generate signature key pair*
4: $tbc \leftarrow (\text{'pk'}, \text{st}.pk) + params.tbc$ ▷ *Set public verification key*
5: $\psi \leftarrow \{\text{PKIX}_C.\text{Certify}(\text{tbc})\}$ ▷ *Issue self-signed certificate*
6: $(\text{st}.certs, \text{st}.CRL) \leftarrow \perp$ ▷ *Initialize empty sets of certificates certified and revoked by the current entity*
7: **return** (st, ψ)
end procedure

Algorithm 5 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** $tbc.subject = \text{st}.\iota$ **then return** $(\text{st}, \perp)$ ▷ *Self-signed cert only from Init!*
2: $tbc.type \leftarrow$ 'public-key' ▷ *Certify only public-key certs*
3: $\sigma \leftarrow C.\text{Certify}_{\text{st}.sk}(tbc)$ ▷ *Sign certificate*
4: $\psi \leftarrow (tbc, \sigma)$ ▷ *Certified data*
5: $\text{st}.certs \mathrel{+}= \psi$ ▷ *Store locally*
6: **return** (st, ψ)
end procedure

Algorithm 6 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** $\psi \in \text{st}.certs$ **then** ▷ *ψ was issued by this CA*
2: $\text{st}.CRL \mathrel{+}= \psi.serial$ ▷ *Add to local CRL*
3: **return** $(\text{st}, \top)$ ▷ *ψ revoked successfully*
4: **end if**
5: **return** $(\text{st}, \perp)$ ▷ *Failure*
end procedure

Algorithm 7 PKIX_C.Chains:

Input: local clock clk, certificate ψ, root store $store$ and auxiliary information ω.
Output: certificate chains for ψ.
procedure PKIX_C.Chains(clk, ψ, $store$, ω)

1: **return** ω ▷ *ω contains all the relevant data*
end procedure

Algorithm 8 PKIX_C.Valid: certificate validation

Input: local clock clk, certificate ψ, root store $store$ and auxiliary information ω.
Output: $\top$ if ψ is a valid certificate, and $\perp$ otherwise.
procedure PKIX_C.Valid(clk, ψ, $store$, ω)

1: **if** $\psi \in store \wedge clk \in [\psi.from, \psi.to]$ **then return** $\top$ ▷ *Root certs are valid*
2: **If** $\left[\begin{array}{c} \exists \xi \in \omega \text{ s.t. } \xi[0] = \psi \wedge \\ \text{IsChain}^C(\xi) \wedge \\ (\exists i)(\xi[i] \in store) \wedge \\ (\forall \psi' \in \xi)(clk \in [\psi'.from, \psi'.to]) \wedge \\ \text{ChainIsNotRevoked}(\xi) \end{array}\right]$ ▷ *ω has a chain ξ for ψ* ▷ *ξ is correctly structured* ▷ *ξ ends in a trusted CA* ▷ *ξ's certs did not expire* ▷ *ξ's certs are non-revoked*
3: **then return** $\top$
4: **end if**
5: **return** $\perp$ ▷ *Otherwise, ψ is invalid*
end procedure

6: **procedure** ChainIsNotRevoked(ξ)
7: **return** $\forall \psi' \in \xi$ ▷ *For every certificate in ξ*
8: $\exists \xi_{CRL} \in \omega$ **s.t.:** ▷ *Exists a CRL cert chain*
9: $\xi_{CRL}[0].type = CRL \wedge$ ▷ *$\xi_{CRL}[0]$ is a CRL*
10: $\xi_{CRL}[0].issuer = \psi'.issuer \wedge$ ▷ *Issued by $\psi'.issuer$*
11: $\psi'.serial \notin \xi_{CRL}[0].CRL \wedge$ ▷ *ψ' is not in the CRL*
12: $\text{PKIX}_C.\text{Valid}(\text{clk}, \xi_{\text{CRL}}[0], \text{store}, \omega - \{\xi\})$ ▷ *And $\xi_{CRL}[0]$ is valid*
13: **end procedure**

Algorithm 9 PKIX_C.GetCRL: retrieve CRL

Input: local state st and local clock clk.
Output: updated state and current CRL.
procedure PKIX_C.GetCRL (st, clk)

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

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

CT.Lookup *(Algorithm 18).* 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).

CT.Wakeup *(Algorithm 19).* First, the algorithm asks for another Wake-up after another Δ_{MMD} interval (line 1). In a logger, the algorithm updates the *signed tree hash* over the local log maintained by the logger (lines 3-9), and stores the STH locally (line 10) so it can be retrieved by monitors in CT.GetSTH-Req (Algorithm 21). In a monitor, CT.Wakeup outputs a request to get the latest STH for each of the loggers overseen by this monitor (line 12).

CT.AddPreChain-Req *(Algorithm 20).* Add a (pre)certificate to the log. The algorithm ensures that the (pre)certificate is a valid PKIX certificate, not already logged, and issued before at most the maximal delay and clock drift (line 1). Then, the algorithm adds the certificate to the local log (line 2), generates a Signed Certificate Timestamp (SCT; lines 3-8) and outputs the SCT (line 9).

CT.GetSTH-Req *(Algorithm 21).* The algorithm outputs the latest STH (line 1), that was generated in CT.Wakeup (Algorithm 19).

CT.GetSTH-Resp *(Algorithm 22).* The algorithm 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 (line 5), until the newly logged certificates will be retrieved and the STH can be verified as a valid root that reflects the addition of the newly logged certificates. To that end, the algorithm outputs a 'GetEntries' request, to be sent to the logger, to which the logger should respond by sending the entries (of the newly logged certificates) (line 6).

CT.GetEntries-Req *(Algorithm 23).* The algorithm is asked to output a subset of the logged certificates, starting from index *start* until index *end*. Thus, the algorithm takes the relevant subset (line 4), generates a signature over it (lines 1-6) and outputs it (line 7).

CT.GetEntries-Resp *(Algorithm 24).* The algorithm uses the logger's certificate, which is stored in the monitor's local state (line 2) to check that the input was indeed signed by the logger (line 3). Then, the algorithm calculates the updated tree hash by adding the newly logged certificates to the local copy of the log (lines 4-5). If the STH that was stored temporarily in CT.GetSTH-Resp (Algorithm 22) indeed matches the STH that was calculated in line 5, the monitor inspects the newly logged certificates according to its implementation (line 9). Finally, the STH and new certificates are added to the local state (lines 7-8).

6 SECURITY ANALYSIS

In this section, we analyze the security of PKIX with and without the CT extension against the security requirements defined in Section 4. First, we define the model specifications (assumptions) in §6.1, then, we show which requirements each PKI satisfies under these model assumptions in §6.2.

6.1 Models

Different protocols assume different models; even the same protocol may assume different models in order to ensure different requirements. The PKIX_C and $\mathsf{CT}_C^{\mathcal{MT}^h}$ protocols assume models based on the following predicates, defined in [16]; these model predicates formalize standard adversary, communication and synchronization assumptions.

- The $\pi^{\text{Drift}}_{\Delta_{clk}}$ predicate ensures clock drifts from real time are bounded by Δ_{clk}.
- The $\pi^{Wake-up}_{\Delta_{clk}}$ predicate ensures *Wake-up* operations are invoked within Δ_{clk} of the requested (real) time.
- The $\pi^{\text{Com}}_{\Delta_{com}}$ predicate ensures reliable communication between non-faulty parties, with delays bounded by Δ_{com}. Namely:
  (1) Whenever an entity ι includes in its output a triple of the form (α-req,ι',x), where α-req is one of the request operations of CT, then, within Δ_{com}, there is a α-req event in 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 a α-resp event in ι with input x.
- The π^{F} predicate ensures that all entities follow the protocol correctly, except for entities in T.F (the set of faulty entities outputted by the adversary).
- The $\pi^{\mathsf{F,HL}}$ predicate extends π^{F} to also ensure *Honest Loggers (HL)*. Namely, if any benign monitor has a 'Monitor' operation for logger ι, then ι has to be benign.

To illustrate, we include the $\pi^{\text{Drift}}_{\Delta_{clk}}$ model predicate from [16] in Algorithm 10.

Algorithm 10 The $\pi^{\text{Drift}}_{\Delta_{clk}}$ model predicate

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** $T.\tau[\hat{e}] \geq T.\tau[\hat{e}-1]$ ▷ *The real time difference is monotonically increasing*

6.2 Security Analysis

Theorem 4. *If C is a secure certificate scheme, then PKIX_C satisfies the following requirements:*

- *Existential unforgeability, under model (π^{F}).*
- *Accountability, under the trivial (always true) model.*
- *Weak Δ_{Rev}-Revocation, where $\Delta_{\text{Rev}} = \Delta_r + \Delta_{clk}$, under model ($\pi^{\mathsf{F}}_{\Delta_r} \wedge \pi^{Drift}_{\Delta_{clk}}$).*

Proof. We proved existential unforgeability in Lemma 2. We prove accountability and weak Δ_{PKIX}-revocation in Lemmas 10 and 12, respectively. The proofs are presented in appendices B and C, respectively; we present their sketches below.

Accountability. Assume, to the contrary, that PKIX_C fails to ensure accountability. Namely, with non-negligible probability, the accountability predicate returns $\perp$, i.e., for a random transcript T:

$$\pi_{\mathsf{ACC}}(T) = \perp \tag{4}$$

Following the implementation of π_{ACC}, let:

$(clk, \psi, store, \omega) \leftarrow T.out_{\mathcal{A}}$ [Alg. 3 line 2]

then, following Equation (4):

$\mathcal{P}.\mathsf{Valid}(clk, \psi, store, \omega) = \top$ [Alg. 3 line 3]

and yet:

$\nexists \xi \in \mathcal{P}.\mathsf{Chains}(clk, \psi, store, \omega),\ j < |\xi|$ **s.t.** [Alg. 3 line 4]

$\xi[j] = \psi \wedge \textsc{IsChain}(\xi) \wedge \mathcal{P}.\mathsf{Valid}(clk, \xi[j+1], store, \omega)$ [Alg. 3 line 5]

However, according to the implementation of PKIX_C.Valid (Algorithm 8), the algorithm only outputs $\top$ if PKIX_C.Chains outputs a valid certificate chain for the same inputs, showing a contradiction. Hence, PKIX_C satisfies accountability.

Weak Δ_{PKIX}-Revocation. We prove in the following steps:

(1) We first define the $\overline{\mathbf{Exp}}^{Forge}_{\mathcal{A},\mathcal{P}}(params)$ experiment, based on the $\pi_{Weak-\Delta_{\text{Rev}}}$ predicate (Definition 8). The $\overline{\mathbf{Exp}}^{Forge}_{\mathcal{A},\mathcal{P}}(params)$ experiment executes an adversary $\mathcal{A}$ with the PKI scheme $\mathcal{P}$. Adversary $\mathcal{A}$ is given public validation key pk but not the corresponding signing/certifying key sk, and can use an oracle to the C.Certify$_{sk}$ operation. Adversary $\mathcal{A}$ 'wins' if it outputs a certificate ψ which is valid (according to pk), without $\mathcal{A}$ asking the oracle to certify ψ. See Definition 18.

(2) After that, we show that the existence of an adversary $\mathcal{A}$ that 'wins' the $\overline{\mathbf{Exp}}^{Forge}_{\mathcal{A},\mathsf{PKIX}_C}(params)$ game with non-negligible probability contradicts the security of the certification scheme C. See Claim 11.
(3) We conclude by showing that if PKIX_C *does not satisfy* the weak Δ_{Rev}-Revocation requirement, we can construct an adversary $\mathcal{A}_{\mathsf{Rev}}$ that wins the $\overline{\mathbf{Exp}}^{Forge}_{\mathcal{A}_{\mathsf{Rev}},\mathsf{PKIX}_C}(params)$ game with non-negligible probability. See Lemma 12.

□

Theorem 5. *Let C be a secure certificate scheme and let h be an ACR (any collision resistant) keyed hash function. Then, $\mathsf{CT}_C^{\mathcal{MT}^h}$ satisfies the following requirements:*

- *Existential unforgeability, under model (π^{F}).*
- *Accountability, under the trivial (always true) model.*
- *Weak Δ_{Rev}-Revocation, where $\Delta_{\mathsf{Rev}} = \Delta_r + \Delta_{clk}$, under model ($\pi^{\mathsf{F}} \wedge \pi^{Drift}_{\Delta_{clk}}$).*

Proof. For existential unforgeability, see Lemma 2. For accountability and weak Δ_{Rev}-Revocation, the implementation related to certificates issuance and revocation in $\mathsf{CT}_C^{\mathcal{MT}^h}$ is identical to PKIX_C, since the implementation in Algorithms 11-24 directly invokes the corresponding PKIX_C algorithms. The additional implementation in $\mathsf{CT}_C^{\mathcal{MT}^h}$ only regards transparency, and in particular, can only issue SCTs, STHs and signatures over newly added certificates to the log, but cannot issue public-key certificates or CRL certificates. Therefore, the proofs that PKIX_C satisfies these properties also hold for $\mathsf{CT}_C^{\mathcal{MT}^h}$; otherwise, the same adversary that breaks these properties in $\mathsf{CT}_C^{\mathcal{MT}^h}$ also breaks them in PKIX_C. □

Next, Theorem 6 proves that CT ensures also the Δ_{Tra}-Transparency requirement under the *CT honest-logger model*[17], which we define as:

$$\mathcal{M}^{\mathsf{CT-HL}} \equiv \left(\pi^{\mathsf{F,HL}} \wedge \pi^{\mathrm{Drift}}_{\Delta_{clk}} \wedge \pi^{\mathrm{Com}}_{\Delta_{com}} \wedge \pi^{Wake-up}_{\Delta_{clk}}\right) \quad (5)$$

Theorem 6. *Let C be a secure certificate scheme, h be an ACR keyed hash, N be a set of entities, and $\Delta_{\mathsf{Tra}} = 2\cdot\Delta_{MMD} + 3\cdot\Delta_{clk} + 5\cdot\Delta_{com}$. Then, $\mathsf{CT}_C^{\mathcal{MT}^h}$ satisfies the Δ_{Tra}-Transparency requirement, under model $\mathcal{M}^{\mathsf{CT-HL}}$.*

Proof. Suppose $\mathsf{CT}_C^{\mathcal{MT}^h}$ *does not* satisfy the Δ_{Tra}-Transparency requirement under model $\mathcal{M}^{\mathsf{CT-HL}}$. Namely, there exists an adversary $\mathcal{A}$ that produces, with non-negligible probability, a certificate ψ that is valid at time $\tau \geq \psi.from + \Delta_{\mathsf{Tra}}$, such that a benign monitor ι_M who is monitoring a logger of ψ since $\tau - \Delta_{\mathsf{Tra}}$ is, at time τ, unaware of ψ. Let us denote this logger by ℓ; from the honest-logger assumption of $\mathcal{M}^{\mathsf{CT-HL}}$, we know that ℓ is benign.

Since ψ is valid at time τ, then, from lines 2-5 of CT.Valid (Alg. 16), it follows that ψ must contain an SCT, denoted ψ_{sct}, which is properly certified by the benign logger ℓ. Furthermore, $\psi_{sct}.timestamp - \Delta_{clk} \leq \tau$.

From Theorem 5, $\mathsf{CT}_C^{\mathcal{MT}^h}$ satisfies existential unforgeability. Hence, ψ_{sct} *must* have been issued by logger ℓ, using its self-signed key. CT.AddPreChain-Req (Algorithm 20) is the only operation where ℓ issues an SCT certificate, and the value of $\psi_{sct}.timestamp$ would be the value of ℓ's clock at the time. Let τ_{SCT} denote the real time when ℓ issued the SCT; from the $\pi^{\mathrm{Drift}}_{\Delta_{clk}}$ model predicate, $\tau_{SCT} \leq \psi_{sct}.timestamp + \Delta_{clk}$. On the other hand, ℓ would issue the SCT only if its clock upon receiving ψ is at most $\psi_{sct}.timestamp \leq \psi.from + \Delta_{clk} + \Delta_{com}$.

Since ℓ is benign, it asks for Wake-up every Δ_{MMD} (see the CT.Init and CT.Wakeup operations); by the $\pi^{Wake-up}_{\Delta_{clk}}$ model predicate, the operation will occur after (at most) $\Delta_{MMD} + \Delta_{clk}$, i.e., at or before $\tau_{SCT} + \Delta_{MMD} + \Delta_{clk}$. At this time, ℓ updates its STH to reflect the logging of ψ (and possibly other certificates). The STH is signed by ℓ, and includes the Merkle-tree (accumulator) digest of the log of all certificates logged by ℓ until this time.

The benign monitor ι_M that monitors ℓ also invokes CT.Wakeup every Δ_{MMD}. In every invocation of CT.Wakeup, ι_M requests a CT.GetSTH-Req operation from every logger that ι_M is monitoring, which includes ℓ. From the $\pi^{\mathrm{Com}}_{\Delta_{com}}$ model, the CT.GetSTH-Req operation in ℓ occurs at most Δ_{com} afterwards. Namely, a CT.GetSTH-Req operation, invoked by ι_M, must happen in ℓ when ℓ's clock is at or before $\tau_{SCT} + \Delta_{MMD} + \Delta_{clk} + \Delta_{com}$ (including the possible impact of the clock bias and of the network delay).

From the $\pi^{\mathrm{Com}}_{\Delta_{com}}$ predicate, the corresponding CT.GetSTH-Resp event in ι_M occurs within Δ_{com}. As shown in CT.GetSTH-Resp (Algorithm 22), the monitor ι_M would identify that new certificates were logged and ask for them, i.e., invoke CT.GetEntries-Req at ℓ; and the logger ℓ immediately responds by sending the certificates, including ψ. 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:

$$\begin{aligned}\tau_\psi &\equiv \tau_{SCT} + 2\cdot\Delta_{MMD} + \Delta_{clk} + 4\cdot\Delta_{com}\\ &\leq \psi_{sct}.timestamp + 2\cdot\Delta_{MMD} + 2\cdot\Delta_{clk} + 4\cdot\Delta_{com}\\ &\leq \psi.from + 2\cdot\Delta_{MMD} + 3\cdot\Delta_{clk} + 5\cdot\Delta_{com}\\ &= \psi.from + \Delta_{\mathsf{Tra}}\end{aligned} \quad (6)$$

Which contradicts the assumption that $\mathcal{A}$ produces, with non-negligible probability, a certificate ψ that is valid at time $\tau \geq \psi.from + \Delta_{\mathsf{Tra}}$ but not known to ι_M. □

Finally, we show that CT *fails* to satisfy the Δ_{Tra}-Transparency requirement, if we allow rogue loggers, i.e., under model:

$$\mathcal{M}_{\Delta_{clk},\Delta_{com}} \equiv \left(\pi^{\mathsf{F}} \wedge \pi^{\mathrm{Drift}}_{\Delta_{clk}} \wedge \pi^{\mathrm{Com}}_{\Delta_{com}} \wedge \pi^{Wake-up}_{\Delta_{clk}}\right) \quad (7)$$

Claim 7. *Let C be a secure certificate scheme, h be an ACR keyed hash, N be a set of entities, and $\Delta_{\mathsf{Tra}} = 2\cdot\Delta_{MMD} + 3\cdot\Delta_{clk} + 5\cdot\Delta_{com}$. Then, $\mathsf{CT}_C^{\mathcal{MT}^h}$ does not satisfy the Δ_{Tra}-Transparency correctness requirement under model $\mathcal{M}_{\Delta_{clk},\Delta_{com}}$.*

Argument. Let $\mathcal{A}$ be an adversary that controls a logger $\ell \in T.\mathsf{F}$ for a transcript T. $\mathcal{A}$ instructs ℓ to execute $\mathsf{CT}_C^{\mathcal{MT}^h}$ as described in §5.2, with the single change of eliminating line 2 from Algorithm 20. As a result, ℓ generates an SCT to every valid certificate, yet ℓ never logs these certificates, and therefore, never informs monitors about them. This means that despite the fact that there is a valid SCT from ℓ for each of the certificates, none of these certificates will be known to monitors. Hence, the existence of such $\mathcal{A}$ means that

[17] In Claim 7, below, we show that the honest-logger assumption is necessary.

the Δ_{Tra}-Transparency requirement is not satisfied under model $\mathcal{M}_{\Delta_{clk},\Delta_{com}}$. □

7 CONCLUSIONS AND FUTURE WORK

We presented specifications for secure PKI schemes, and applied them to prove the security of two applied PKI schemes: PKIX and Certificate Transparency (CT). Future work may apply our specifications to other PKI schemes, present additional specifications for PKI schemes (capturing additional properties, e.g., privacy properties), and apply a similar approach to other practical cryptographic protocols, e.g., blockchain protocols.

REFERENCES

[1] 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*.

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

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

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

[9] David Cooper, Stefan Santesson, Stephen Farrell, Sharon Boeyen, Russell Housley, and William Polk. 2008. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. Tech. rep.

[10] 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).

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

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

[13] 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.

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

[15] 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.

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

[17] 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.

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

[19] 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.

[20] 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.

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

[22] 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*.

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

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

[25] Ben Laurie, Adam Langley, Emilia Kasper, Eran Messeri, and Rob Stradling. 2019. Certificate transparency version 2.0. *IETF TRANS (Public Notary Transparency) WG, Internet-Draft*, (Nov. 2019).

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

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

[28] 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.

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

[30] 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.

[31] 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.

[32] 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.

[33] S. Santesson, M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams. 2013. X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP. RFC 6960 (Proposed Standard). RFC. Updated by RFC 8954. Fremont, CA, USA: RFC Editor, (June 2013). DOI: 10.17487/RFC6960.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

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

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

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

[46] 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.

A CRYPTOGRAPHIC PRIMITIVES

A.1 MoSS

First, Definition 10 defines the advantage of adversary $\mathcal{A}$ against protocol $\mathcal{P}$, for given some specification predicate π.

Definition 10 (Advantage of adversary $\mathcal{A}$ against protocol $\mathcal{P}$ for specification predicate π). *Let $\mathcal{A}, \mathcal{P}$ be algorithms and let π be a specification predicate. The advantage of adversary $\mathcal{A}$ against protocol $\mathcal{P}$ for specification predicate π is defined as:*

$$\epsilon^{\pi}_{\mathcal{A},\mathcal{P}}(params) \stackrel{def}{=} \Pr\left[\begin{array}{c} \pi(T) = \bot, \text{ where} \\ T \leftarrow \mathbf{Exec}_{\mathcal{A},\mathcal{P}}(params) \end{array}\right] \tag{8}$$

Now, Definition 11 defines an adversary that satisfies a given model $\mathcal{M}$.

Definition 11 (Adversary $\mathcal{A}$ satisfies model $\mathcal{M}$). *Let $\mathcal{A} \in PPT$, and let $\mathcal{M}$ be a model specification, i.e., $\mathcal{M} = (\pi, \beta)$. We say that adversary $\mathcal{A}$ satisfies model $\mathcal{M}$, denoted $\mathcal{A} \models_{poly} \mathcal{M}$, if for every protocol $\mathcal{P} \in PPT$ and $params \in \{0,1\}^*$, the advantage of $\mathcal{A}$ against $\mathcal{P}$ for π is at most negligibly greater than $\beta(params)$, i.e.:*

$$\mathcal{A} \models_{poly} \mathcal{M} \stackrel{def}{=} \left[\begin{array}{c} (\forall \mathcal{P} \in PPT, params \in \{0,1\}^*): \\ \epsilon^{\pi}_{\mathcal{A},\mathcal{P}}(params) \leq \beta(params) + Negl(params.1^{\kappa}) \end{array}\right] \tag{9}$$

Finally, Definition 12 defines what it means for a protocol $\mathcal{P}$ to satisfy a specific requirement under a specific model.

Definition 12 (Protocol $\mathcal{P}$ satisfies requirement $\mathcal{R}$ under model $\mathcal{M}$). *Let $\mathcal{P} \in PPT$, and let $\mathcal{R}$ be a requirement specification, i.e., $\mathcal{R} = (\pi, \beta)$. We say that protocol $\mathcal{P}$ satisfies requirement $\mathcal{R}$ under model $\mathcal{M}$, denoted $\mathcal{P} \models^{\mathcal{M}}_{poly} \mathcal{R}$, if for every PPT adversary $\mathcal{A}$ that satisfies $\mathcal{M}$ and every parameters $params \in \{0,1\}^*$, the advantage of $\mathcal{A}$ against $\mathcal{P}$ for π is at most negligibly greater than $\beta(params)$, i.e.:*

$$\mathcal{P} \models^{\mathcal{M}}_{poly} \mathcal{R} \stackrel{def}{=} \left[\begin{array}{c} (\forall \mathcal{A} \in PPT \text{ s.t. } \mathcal{A} \models_{poly} \mathcal{M},\ params \in \{0,1\}^*): \\ \epsilon^{\pi}_{\mathcal{A},\mathcal{P}}(params) \leq \beta(params) + Negl(params.1^{\kappa}) \end{array}\right] \tag{10}$$

A.2 Accumulator scheme and the Merkle Tree construction

We define an accumulator scheme and its security properties.

An accumulator receives as input an ordered set of elements $X = \{x_i\}_{i=1}^{k} \in \mathcal{X}^k$, and outputs their *digest* Δ, as well as a *proof of inclusion (PoI)* π_i for each element $accEl_i$ in the input. We will later define and discuss extensions, most notably, *dynamic/additive accumulators*, that allow to add elements to the accumulator in multiple operations.

Definition 13 (An accumulator scheme). *An* accumulator scheme *$\mathcal{ACC}$ is a tuple of algorithms ($\mathcal{ACC}$.Setup, $\mathcal{ACC}$.Add, $\mathcal{ACC}$.Verify) with element space $\mathcal{ACC}.\mathcal{X}$, where the algorithms are defined as:*

- $\mathcal{ACC}.\mathsf{Setup}(1^{\lambda}) \rightarrow pk$*: input security parameter $\lambda \in \mathbb{N}$ outputs a public key pk.*
- $\mathcal{ACC}.\mathsf{Add}_{pk}(X) \rightarrow (\Delta, \Pi)$*: add (accumulate) a sequence of elements $X = \{x_i\}_{i=1}^{k} \in \mathcal{ACC}.\mathcal{X}^k$. Output the digest Δ of X and a set $\Pi = \{\pi_i\}_{i=1}^{k}$ of* proofs-of-inclusions*, where π_i is the proof of inclusion for input element x_i.*
- $\mathcal{ACC}.\mathsf{Verify}_{pk}(\Delta, x, \pi) \rightarrow \{0,1\}$*: output 1 (true) if π is a valid proof of inclusion for x and 0 (false) otherwise. As we define below, a valid proof of inclusion shows that there is a set X such that $x \in X$ and $\mathsf{Add}_{pk}(X)$ outputs Δ.*

We next define three basic requirements from accumulator schemes: *correctness* and *unforgeability* of the proofs-of-inclusion, and *collision resistance* for the digest. We begin with the most basic property, *correctness*.

Definition 14 (Correctness of Accumulator Scheme). *An accumulator $\mathcal{ACC}$ is said to satisfy* correctness*, if the following holds for every security parameter $\lambda \in \mathbb{N}$, input sequence $X = \{x_i\}_{i=1}^{k} \in \mathcal{X}^k$ and $i \in [1, \ldots, k]$. Let $pk \leftarrow \mathcal{ACC}(\lambda)$ and $(\Delta, \Pi) \leftarrow \mathcal{ACC}.\mathsf{Add}(X)$. Then, for every $x \in X$, there exists $\pi \in \Pi$ s.t. $\mathcal{ACC}.\mathsf{Verify}_{pk}(\Delta, x, \pi) = 1$.*

Note: typically, each PoI is for the corresponding input element, i.e., $\mathcal{ACC}.\mathsf{Verify}_{pk}(\Delta, x_i, \pi_i) = 1$. However, this is not required by the definition above.

The next property we define is *collision resistance*. The collision-resistance of accumulators is the natural extension of the collision-resistance property of hash functions. We present only the keyed 'any collision resistance' variant; keyless variants and 'target collision resistance' can be defined similarly.

Definition 15 (Collision-Resistant Accumulator). *We say that an accumulator scheme $\mathcal{ACC}$ is* Collision-Resistant *if for any PPT adversary $\mathcal{A}$ and security parameter $\lambda \in \mathbb{N}$ holds:*

$$\Pr\left[\mathcal{ACC}.\mathsf{Add}(X) = \mathcal{ACC}.\mathsf{Add}(X') \,\middle|\, \left\{\begin{array}{l} pk \leftarrow \mathcal{ACC}.\mathsf{Setup}(\lambda); \\ (X, X') \leftarrow \mathcal{A}(pk) \end{array}\right\}\right] \in Negl(\lambda)$$

The final security property which we define in this section is *existential unforgeability*. Unforgeability extends collision-resistance, to ensure the integrity of specific elements in the accumulated sequence. The existential unforgeability definition is similar to the existential unforgeability of signature and MAC schemes.

Definition 16 (Existentially Unforgeable Accumulator). *We say that an accumulator scheme $\mathcal{ACC}$ is* Existentially Unforgeable *if for any PPT adversary $(\mathcal{A}_1, \mathcal{A}_2)$ and security parameter $\lambda \in \mathbb{N}$ holds:*

$$\Pr\left[\mathcal{ACC}.\mathsf{Verify}(\Delta, x, \pi_{\mathcal{A}}) = 1 \wedge x \notin X\right] \in Negl(\lambda)$$

Where pk, Δ, x and $\pi_{\mathcal{A}}$ are defined by the following process:

$$pk \leftarrow \mathcal{ACC}.\mathsf{Setup}(\lambda)\ ;\ (s, X) \leftarrow \mathcal{A}_1(pk)\ ;$$
$$(\Delta, \Pi) \leftarrow \mathcal{ACC}.\mathsf{Add}(X)\ ;\ (x, \pi_{\mathcal{A}}) \leftarrow \mathcal{A}_2(s, \Delta, \Pi)$$

A.3 The CT Merkle-Tree Accumulator

In this subsection, we present and prove the security of the CT Merkle-Tree Accumulator. This construction is a keyed-version of the construction used by the Certificate Transparency 2.0 standard, defined in Section 2.1 of RFC 9162 [36].

Definition 17 (The CT Merkle-Tree Accumulator). *Let h be a keyed (any-collision-resistant) hash function. The h-CT Merkle-Tree Accumulator, denoted $\mathcal{MT}^h$, is defined by the following algorithms:*

- $\mathcal{MT}^h.\mathsf{Setup}(1^{\lambda})$ *outputs a public key $pk \xleftarrow{R} \{0,1\}^{\lambda}$.*
- $\mathcal{MT}^h.\mathsf{Add}_{pk}(X)$ *outputs $(\Delta_{pk}(X), \pi_{pk}(X))$, where Δ, π are computed as follows.*
  First define $\Delta_{pk}(X)$:
  - *If X is an empty list then $\Delta_{pk}(X) = h_{pk}(\epsilon)$, where ϵ denotes the empty string.*

- *If $|X| = 1$, i.e., X contains a single entry x_1, then $\Delta_{pk}(X) = h_{pk}(0x00||x_1)$. This computation is sometimes called a leaf hash.*
- *If $|X| > 1$, then we compute $\Delta_{pk}(X)$ as follows. Let $n = |X|$, i.e., $X = \{x_1, \ldots, x_n\}$, and let $k = \max_{i \in \mathbb{N}}\{2^i < n\}$, i.e., the largest power of two smaller than n (i.e., $k < n \leq 2k$). We compute $\Delta_{pk}(X)$ recursively using $\Delta_{pk}(X) = h(0x01||\Delta_{pk}(\{x_1, \ldots, x_k\})||\Delta_{pk}(\{x_{k+1}, \ldots, x_n\}))$.*

- *We omit the (similar) definition of $\mathcal{MT}^h.\mathsf{Verify}_{pk}(\Delta, x, \pi)$ and $\pi_{pk}(X)$.*

Finally, we state security properties of the CT Merkle-Tree Accumulator.

Lemma 8. *The CT Merkle-Tree Accumulator is correct, collision-resistant and unforgeable.*

B PKIX ENSURES ACCOUNTABILITY

PKIX ensures accountability by ensuring that for a certificate ψ to be valid, it must come with a valid certificate chain. Let us first state and prove this claim, which we use both to prove accountability (next) and to prove weak Δ revocation (later).

Claim 9. *(A valid certificate must come with a valid chain) If, for some values $clk, \xi, store, \omega$ holds $\mathsf{PKIX}_C.\mathsf{Valid}(\mathsf{clk}, \psi, \mathsf{store}, \omega) = \top$, then*

$$\exists \xi \in \mathsf{PKIX}_C.\mathsf{Chains}(\mathsf{clk}, \psi, \mathsf{store}, \omega) \textbf{ s.t. :}$$
$$\xi[0] = \psi \ \wedge \ \textsc{IsChain}^C(\xi) \ \wedge (\exists i)(\xi[i] \in store) \wedge$$
$$(\forall \psi' \in \xi)(clk \in [\psi'.from, \psi'.to]) \ \wedge \textsc{ChainIsNotRevoked}(\xi)$$

Proof. The claim follows immediately from the implementation of PKIX.Valid (Algorithm 8 line 2), which checks exactly for this condition. □

Let us now prove that PKIX ensures accountability.

Lemma 10. *Let C be a secure certificate scheme. Then, PKIX_C satisfies the accountability requirement.*

Proof. Assume to the contrary that PKIX_C does not satisfy the accountability requirement. Hence, there exists a PPT adversary $\mathcal{A}_{\mathsf{ACC}}$ such that:

$$\epsilon^{\pi_{\mathsf{ACC}}}_{\mathcal{A}_{\mathsf{ACC}},\mathsf{PKIX}}(params) \notin Negl(params.1^\kappa) \tag{11}$$

Where π_{ACC} is defined in Algorithm 3.

In other words, from Definition 7, $\mathcal{A}_{\mathsf{ACC}}$ satisfies

$$\Pr\left[\begin{array}{c}\pi_{\mathsf{ACC}}(T) = \bot, \text{ where} \\ T \leftarrow \mathbf{Exec}_{\mathcal{A}_{\mathsf{ACC}},\mathsf{PKIX}}(params)\end{array}\right] \notin Negl(params.1^\kappa) \tag{12}$$

Note that for $\pi_{\mathsf{ACC}}(T) = \bot$, the adversary must have produced a certificate ψ with auxiliary information ω that is valid, i.e.:

$$\mathsf{PKIX.Valid}(clk, \psi, store, \omega) = \top$$

and yet, no valid chain exists. However, Claim 9 shows this cannot hold. Therefore, there cannot exist such adversary $\mathcal{A}_{\mathsf{ACC}}$ that produces an input which PKIX.Valid classifies as valid without PKIX.Chains able to produce a valid certificate chain, in contrast to the assumption that such adversary exists.

□

C PKIX SATISFIES THE WEAK REVOCATION REQUIREMENT

First, we define the $\overline{\mathbf{Exp}}^{Forge}_{\mathcal{A},\mathcal{P}}(params)$ experiment, a modification of the $\pi_{Weak-\Delta_{\mathsf{Rev}}}$ predicate (Definition 8). In the $\overline{\mathbf{Exp}}^{Forge}_{\mathcal{A},\mathcal{P}}(params)$ experiment, an adversary $\mathcal{A}^{C.\mathsf{Certify}_{sk}(\cdot)}_{pk}$ is given a verification key pk and has an oracle access to a corresponding signing oracle $C.\mathsf{Certify}_{sk}(\cdot)$. The adversary wins if after the execution of PKI scheme $\mathcal{P}$, the adversary produces a certificate ψ that validates successfully using the public verification key pk, without the adversary asking the oracle to certify ψ.

Definition 18 (The $\overline{\mathbf{Exp}}^{Forge}_{\mathcal{A},\mathcal{P}}(params)$ experiment). *Let $\mathcal{P}$ be a PKI scheme and $\mathcal{A}$ be a PPT algorithm. We define $\overline{\mathbf{Exp}}^{Forge}_{\mathcal{A},\mathcal{P}}(params)$ as the following random process:*

1. *Generate key pair $(sk, pk) \leftarrow C.\mathsf{KG}(params.1^\kappa)$.*
2. *Let $\mathcal{A}^{C.\mathsf{Certify}_{sk}(\cdot)}_{pk}$ denote algorithm $\mathcal{A}$ with input pk and oracle access to the function $C.\mathsf{Certify}_{sk}(\cdot)$.*
3. *Execute $\mathcal{A}^{C.\mathsf{Certify}_{sk}(\cdot)}_{pk}$ with $\mathcal{P}$, except for a random entity $\iota \xleftarrow{R} T.\mathsf{N}$, where entity ι also executes $\mathcal{P}$, except:*
   (a) *In Init, ι sets $(\mathsf{st}.sk, \mathsf{st}.pk) \leftarrow (\bot, pk)$, and*
   (b) *ι replaces calls to $C.\mathsf{Certify}_{\mathsf{st}.sk}(\cdot)$ with calls to $C.\mathsf{Certify}_{sk}(\cdot)$.*
   Let T denote the resulting transcript.
4. *Let $\psi \leftarrow T.out_{\mathcal{A}}$. Return 1 ($\mathcal{A}$ 'won') if:*
   (a) *$C.\mathsf{Verify}_{pk}(\psi) = \top$*
   (b) *$\mathcal{A}$ did not use the oracle access on $\psi.tbc$.*
   (c) *ι is an honest entity, i.e., $\iota \in T.\mathsf{N} - T.\mathsf{F}$.*
   Otherwise, return 0 ($\mathcal{A}$ 'lost').

We next observe that that if C is a secure certification scheme, then no PPT adversary $\mathcal{A}$ 'wins' the $\overline{\mathbf{Exp}}^{Forge}_{\mathcal{A},\mathsf{PKIX}_C}$ experiment with non-negligible probability.

Claim 11. *If there is a PPT adversary $\mathcal{A}$ s.t.:*

$$\Pr\left[\overline{\mathbf{Exp}}^{Forge}_{\mathcal{A},\mathsf{PKIX}_C}(params) = 1\right] \notin Negl(1^\kappa) \tag{13}$$

then C is not a secure certification scheme.

Proof. Follows by reduction to the existential unforgeability of C. Namely, if there is such an adversary $\mathcal{A}$, then there is also an adversary $\hat{\mathcal{A}}$ that wins the certificate forgery game against C. Adversary $\hat{\mathcal{A}}$ runs the $\overline{\mathbf{Exp}}^{Forge}_{\mathcal{A},\mathsf{PKIX}_C}$ experiment, except that $\hat{\mathcal{A}}$ outputs the certificate ψ that $\mathcal{A}$ produces to win the $\overline{\mathbf{Exp}}^{Forge}_{\mathcal{A},\mathsf{PKIX}_C}$ experiment (i.e., $\psi \leftarrow T.out_{\mathcal{A}}$). Since $\mathcal{A}$ 'wins' with non-negligible probability, it follows that $\hat{\mathcal{A}}$ will also succeed in forgery with non-negligible probability, contradicting the assumed security of C, as defined in Definition 1. □

Lemma 12. *Let C be a secure certificate scheme, let N be a set of entities and let $\Delta_{\mathsf{Rev}} = \Delta_r + \Delta_{clk}$, where Δ_r is the acceptable revocation delay that all entities in N are initialized with (or smaller), i.e., $params.\Delta_r \leq \Delta_r$. Then, PKIX_C satisfies the weak Δ_{Rev}-Revocation requirement under model $(\pi^{\mathsf{F}}_{\Delta_r} \ \wedge \ \pi^{Drift}_{\Delta_{clk}})$.*

Proof. Assume to the contrary, that PKIX_C fails to satisfy the weak Δ_{Rev}-Revocation under model $(\pi^{\mathsf{F}} \ \wedge \ \pi^{\mathrm{Drift}}_{\Delta_{clk}})$. Namely, there

is a PPT adversary $\mathcal{A}$ that satisfies model ($\pi^{\mathsf{F}} \wedge \pi^{\text{Drift}}_{\Delta_{clk}}$) and yet, has non-negligible advantage against the $\pi_{Weak-\Delta_{\text{Rev}}}$ requirement (Definition 8), i.e.:

$$\epsilon^{\pi_{Weak-\Delta_{\text{Rev}}}}_{\mathcal{A},\text{PKIX}_C}(params) \notin Negl(params.1^{\kappa}) \tag{14}$$

From Definitions 8 and 10, this implies that:

$$\Pr\left[\begin{array}{c} f_{\Delta_{\text{Rev}}}(T) = \perp, \text{where} \\ T \leftarrow \textbf{Exec}_{\mathcal{A},\text{PKIX}_C}(params) \end{array}\right] \notin Negl(params.1^{\kappa}) \tag{15}$$

For a given T, let $(clk, \psi, store, \omega) \leftarrow T.out_{\mathcal{A}}$. From Algorithm 25, $f_{\Delta_{\text{Rev}}}(T) = \perp$ only if:

$$\begin{array}{c} \psi.issuer \in T.\mathsf{N} - T.\mathsf{F} \wedge \\ \text{PKIX}_C.\text{Valid}(\text{clk}, \psi, \text{store}, \omega) \wedge \\ \textsc{RevocationRequested}(clk, \psi, T, \Delta_{\text{Rev}}) \wedge \\ (\textsc{CanFindFraudulentCert}(T, clk, \psi, store, \omega) = \perp) \end{array} \tag{16}$$

Since $\textsc{RevocationRequested}(\tau, \psi, T, \Delta_{\text{Rev}})$ returns $\top$, then ψ must have been previously issued and then revoked at $\psi.issuer$, both before $\tau - \Delta_{\text{Rev}}$; let $\tau_R \leq \tau - \Delta_{\text{Rev}}$ denote the time of revocation. Being non-faulty, $\psi.issuer$ have executed the PKIX_C.Certify and PKIX_C.Revoke operations correctly (Algorithms 5 and 6), and also includes ψ in every CRL certified in the PKIX_C.GetCRL operation after τ_R. As can be confirmed in the pseudocode of PKIX (Figure 1), PKIX_C.GetCRL is the only operation in which PKIX_C certifies a CRL (i.e., certificate with type CRL).

Since PKIX_C.Valid$(clk, \psi, store, \omega)$ return $\top$, i.e., ψ is valid at time τ with auxiliary information ω, it follows from the implementation of PKIX_C.Valid that ω must contain a chain ξ_{CRL} such that $\xi_{CRL}[0]$ is a valid CRL at time τ that does *not* list ψ as revoked. From line 10 of section 8, we know that $\xi_{CRL}[0].issuer = \psi.issuer$.

However, we next show that $\psi.issuer$ cannot have issued $\xi_{CRL}[0]$. Assume otherwise, i.e., that $\xi_{CRL}[0]$ was issued by $\psi.issuer$ (and does *not* list ψ as revoked). Since $\psi.issuer$ is benign, then is must have issued $\xi_{CRL}[0]$ before revoking ψ, i.e., at some time $\tau'_R < \tau_R$. Also, $\xi_{CRL}[0].from$ is the clock value of $\psi.issuer$ at τ'_R. Since the clock drift is bounded by Δ_{clk}, except with negligible probability ($\pi^{\text{Drift}}_{\Delta_{clk}}$), we have $\xi_{CRL}[0].from \leq \tau'_R + \Delta_{clk} < \tau_R + \Delta_{clk}$. And, again since $\psi.issuer$ is benign, we know that $\xi_{CRL}[0].to = \xi_{CRL}[0].from + \Delta_r$. Since $\tau_R \leq \tau - \Delta_{\text{Rev}}$, we have:

$$\begin{aligned} \xi_{CRL}[0].to &= \psi_R.from + \Delta_r \\ &< \tau_R + (\Delta_{clk} + \Delta_r) \\ &= (\tau - \Delta_{\text{Rev}}) + \Delta_{\text{Rev}} \\ &= \tau \end{aligned} \tag{17}$$

Namely, $\xi_{CRL}[0].to < \tau$, which contradicts the assumption that $\xi_{CRL}[0]$ is valid at time τ. This shows that the (benign) $\psi.issuer$ cannot have issued $\xi_{CRL}[0]$.

Still, the attacker obtained a valid CRL $\xi_{CRL}[0]$ in some other way, i.e., not issued by $\psi.issuer$, but with $\xi_{CRL}[0].issuer = \psi.issuer$. Suppose this is the case; recall that $\xi_{CRL}[0]$ must be valid, specifically, PKIX.Valid$(\tau, \xi_{CRL}[0], store, \omega - \{\xi\})$ (line 12 of Algorithm 8). Hence, from Algorithm 8, there is some chain[18] $\xi' \in \omega$ which includes a trust-anchor certificate $\xi'[i] \in store$, such that $\xi_{CRL}[0] = \xi'[0]$ and $\textsc{IsChain}^C(\xi')$ returns $\top$.

[18]Intuitively, we expect ξ' to be the same as ξ_{CRL}, but this is not required.

Hence, from Algorithm 1, C.Verify$_{\xi'[1].pk}(\xi_{CRL}[0])$ holds and $\xi_{CRL}[0].issuer = \xi'[1].subject$. Recall that $\xi_{CRL}[0].issuer = \psi.issuer$ and that $\psi.issuer$ is benign. Let $\hat{\psi}$ denote the self-signed certificate outputted by $\psi.issuer$ upon its initialization. We consider two cases: (1) $\xi'[1].pk = \hat{\psi}.pk$ and (2) $\xi'[1].pk \neq \hat{\psi}.pk$.

Case (1): $\xi'[1].pk = \hat{\psi}.pk$. Since $\psi.issuer$ is benign, then it never exposes the corresponding private signing key (generated upon initialization and used to sign $\hat{\psi}$ and other CRLs and public-key certificates issued by $\psi.issuer$). In fact, $\psi.issuer$ only uses this signing key to certify CRLs when doing the PKIX_C.GetCRL operation. Recall that $\xi_{CRL}[0]$ is a valid CRL not issued by $\psi.issuer$ and that C.Verify$_{\xi'[1].pk}(\xi_{CRL}[0])$ holds. Since the corresponding private key was never exposed or used to certify $\xi_{CRL}[0]$, such an adversary contradicts the existential-unforgeability property of the certificate scheme C.

Case (2): $\xi'[1].pk \neq \hat{\psi}.pk$. From Claim 9, $\xi_{CRL}[0]$ has also a valid chain; but then surely $\textsc{CanFindFraudulentCert}(T, clk, \psi, store, \omega)$ would return $\top$, which contradicts our assumption.

□

D EXPLANATION OF A SUBTLE POINT REGARDING THE ACCOUNTABILITY REQUIREMENT

Just before submission, a careful reviewer, which we will maintain anonymous for the submission, asked us a very natural question regarding the accountability requirement predicate (Algorithm 3). Rephrased, he asked why does the predicate, in line 5, check first that ξ is a chain ($\textsc{IsChain}(\xi)$) and then also that $\xi[j+1]$ is valid. Isn't the fact that $\xi[j+1]$ is valid follows from the fact that ξ is a chain?

The answer is: both checks are necessary. The $\textsc{IsChain}(\xi)$ function, Algorithm 1, checks (only) that ξ is a certificate chain, but does not check that each certificate $\xi[j]$ along the chain is $\mathcal{P}$.Valid. Namely, $\textsc{IsChain}(\xi)$ (Algorithm 1) checks that for every 'chain link', i.e., for every $j : 0 < j < |\xi|$, holds:

Line 3: Each certificate $\xi[j-1]$ in the chain (except the last one) is certified by the next certificate on the chain $\xi[j]$.

Line 4: The issuer of $\xi[j-1]$ is the subject of $\xi[j]$.

Line 5: Each cert $\xi[j]$ in the chain, except the first one, is a public key certificate. Notice that it is important to allow the first certificate on the chain, $\xi[0]$, to be a non-public-key certificate, in particular, a CRL.

Line 5: Each certificate $\xi[j]$ in the chain, except the first one, indicates that the certified entity $\xi[j].subject$ is a CA.

All of these checks are necessary for the accountability predicate - but not sufficient. We also need to confirm that every non-first certificate along the chain, $\xi[j]$, is $\mathcal{P}$.Valid (at the current time, with the current ω and $store$). This is checked by the (recursive) call to $\mathcal{P}$.Valid at the end of line 5 of the accountability requirement predicate (Definition 7).

A related question may be, why do we allow ψ to appear in an arbitrary location (j) along the chain ξ, and do not require it to be the very first certificate along the chain. In fact, we could have required ψ to be the first certificate along the chain ξ; however, that would have made the use of the $\mathcal{P}$.Valid quite inconvenient. This is since

Algorithm 11 CT.Init: initialization

Input: local state st, local clock clk and parameters $params$ which has the same parameters as in Algorithm 4 with the addition of the maximum merge delay Δ_{MMD} and a trusted root store $store$.
Output: updated state, the output of Algorithm 4 and the next wakeup time.
procedure CT.Init (st, clk, $params$)

$(\text{st}, out) \leftarrow$ PKIX.Init(st, clk, $params$) ▷ *See Algorithm 4*
if st.$role \in$ {'Logger', 'Monitor'} **then** ▷ *Only monitors and loggers*
  out += ('Wakeup', clk + st.Δ_{MMD}) ▷ *Add next wakeup time*
  if st.$role$ = 'Monitor' **then** ▷ *If monitor*
    st.$loggers \leftarrow \perp$ ▷ *Init empty list of monitored loggers*
    st.$log \leftarrow \perp$ ▷ *Init empty list of monitored logs*
  else ▷ *If logger*
    st.$mylog \leftarrow \perp$ ▷ *Initialize local log*
  end if
end if
return (st, out) ▷ *Output*
end procedure

Algorithm 12 CT.Certify: certificate issuance

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

if $tbc.type$ = 'pre-certificate' **then** ▷ *If pre-certificate*
  Add critical 'poison' extension to tbc ▷ *'Poison' extension*
end if
return (st, PKIX.Certify(tbc)) ▷ *See Algorithm 5*
end procedure

Algorithm 13 CT.Revoke: certificate revocation

Input: local state st, local clock clk and certificate ψ.
Output: the output of PKIX.Revoke.
procedure CT.Revoke (st, clk, ψ)

return PKIX.Revoke(ψ) ▷ *See Algorithm 6*
end procedure

Algorithm 14 CT.GetCRL: retrieve CRL

Input: local state st and local clock clk.
Output: the output of PKIX.GetCRL.
procedure CT.GetCRL (st, clk)

return PKIX.GetCRL(st, clk) ▷ *See Algorithm 9*
end procedure

Algorithm 15 CT.Chains:

Input: local clock clk, certificate ψ, root store $store$ and auxiliary information ω.
Output: the output of PKIX.Chains.
procedure CT.Chains (clk, ψ, $store$, ω)

return PKIX.Chains(clk, ψ, $store$, ω) ▷ *See Algorithm 7*
end procedure

Algorithm 16 CT.Valid: certificate validation

Input: local clock clk, certificate ψ, root store $store$ and auxiliary information ω.
Output: $\top$ if ψ is a valid certificate, and $\perp$ otherwise.
procedure CT.Valid (clk, ψ, $store$, ω)

return PKIX.Valid(clk, ψ, $store.CAs$, ω) $\wedge$ ▷ *See Algorithm 8*
$(\forall i \in \{1,2\}) \left(\begin{array}{c} \exists \psi_i^{sct} \in \psi.SCT \wedge \\ \exists \psi_i^{\ell} \in store.logger \end{array} \right)$ **s.t.:** ▷ *ψ contains two SCTs*
$\psi_i^{sct}.type$ = 'SCT' $\wedge$ ▷ *ψ_i^{sct} is an SCT*
$C.\text{Verify}_{\psi_i^{\ell}.pk}(\psi_i^{sct}) \wedge$ ▷ *Logger i signed ψ_i^{sct}*
$\psi_i^{sct}.tbc.cert = \psi.tbc - \psi.SCT \wedge$ ▷ *Both SCTs are for ψ*
$\psi_1^{sct}.issuer \neq \psi_2^{sct}.issuer$ ▷ *SCTs from different loggers*
end procedure

Algorithm 17 CT.Monitor: add to monitored logs

Input: local state st, local clock clk and log's identifier ι.
Output: updated state.
procedure CT.Monitor (st, clk, ι)

Add ι to st.$loggers$ ▷ *Add ι to monitored logs*
return (st,('Monitor', st.$loggers$)) ▷ *Output*
end procedure

Algorithm 18 CT.Lookup: subject's certificates lookup

Input: local state st, local clock clk and subject's identifier $subject$.
Output: updated state and all the certificates issued to $subject$ that are known to the entity.
procedure CT.Lookup (st, clk, $subject$)

$certs \leftarrow \perp$ ▷ *Initialize set to $\perp$*
if st.$role$ = 'Monitor' **then**
  $certs \leftarrow \left\{ \psi \;\middle|\; \begin{array}{c} \exists \iota \in \text{st}.loggers, \psi \in \text{st}.log[\iota] \text{ s.t.} \\ \psi.subject = subject \end{array} \right\}$ ▷ *$subject$'s certificates*
end if
return $(\text{st}, certs)$ ▷ *Output*
end procedure

Algorithm 19 CT.Wakeup: scheduled operations

Input: local state st, local clock clk and wake up related information $data$.
Output: updated state and next wakeup time.
procedure CT.Wakeup (st, clk, $data$)

$out \leftarrow$ ('Wakeup', clk + st.Δ_{MMD}) ▷ *Set next wakeup*
if st.$role$ = 'Logger' **then**
  $tbc \leftarrow (type, \text{'STH'})$ ▷ *Certificate of type STH*
  $tbc.issuer \leftarrow$ st.ι ▷ *The issuer of the STH*
  $tbc.from \leftarrow clk$ ▷ *When was issued*
  $tbc.to \leftarrow clk$ + st.Δ_{MMD} ▷ *Time for next STH*
  $tbc.STH \leftarrow \mathcal{MT}^{h}(\text{st}.log)$ ▷ *See CT Merkle-Tree Accumulator (Definition 17)*
  $tbc.size \leftarrow |\text{st}.log|$ ▷ *Log size*
  $\sigma \leftarrow C.\text{Certify}_{\text{st}.sk}(tbc)$ ▷ *Sign certificate*
  st.$\psi_{STH} \leftarrow (tbc, \sigma)$ ▷ *Certified data*
else if st.$role$ = 'Monitor' **then**
  out += {('GetSTH', ι) | $\forall \iota \in$ st.$loggers$} ▷ *Ask loggers for latest STH*
end if
return (st, out) ▷ *Output*
end procedure

Figure 2: CT (PKIX with certificate transparency) implementation

Algorithm 20 CT.AddPreChain-Req: add certificate to log

Input: local state st, local clock clk, (pre)certificate ψ and auxiliary information ω.
Output: updated state and SCT ψ_{SCT}.

procedure CT.AddPreChain-Req (st, clk, ψ, ω)
1: **if** $\left(\begin{array}{c}\text{PKIX.Valid}(\psi, \text{st}.store, clk, \omega) \wedge \\ \psi \notin \text{st}.mylog \ \wedge \ clk \leq \psi.from + \Delta_{clk} + \Delta_{com}\end{array}\right)$ **then**
2: st.$mylog$ += ψ ▷ *Store locally*
3: $tbc.type \leftarrow$ 'SCT' ▷ *Certificate of type SCT*
4: $tbc.issuer \leftarrow$ st.ι ▷ *The issuer of the SCT*
5: $tbc.timestamp \leftarrow clk$ ▷ *Timestamp (when issued)*
6: $tbc.cert \leftarrow \psi$ ▷ *Certificate information*
7: $\sigma \leftarrow C.\text{Certify}_{\text{st}.sk}(tbc)$ ▷ *Sign certificate*
8: $\psi_{SCT} \leftarrow (tbc, \sigma)$ ▷ *Certified data*
9: **return** (st, ('AddPreChain-Resp', ψ_{SCT})) ▷ *Respond with SCT*
10: **end if**
11: **return** (st, $\perp$) ▷ *Invalid request*
end procedure

Algorithm 21 CT.GetSTH-Req: retrieve current STH

Input: local state st, local clock clk.
Output: updated state and STH ψ_{STH}.

procedure CT.GetSTH-Req (st, clk)
1: **return** (st, ('GetSTH-Resp', st.ψ_{STH})) ▷ *Output*
end procedure

Algorithm 22 CT.GetSTH-Resp: store latest STH

Input: local state st, local clock clk, root store $store$ and STH certificate ψ_{STH}.
Output: updated state and request for newly added log entries.

procedure CT.GetSTH-Resp (st, $clk, store, \psi$)
1: $out \leftarrow \perp$
2: $log \leftarrow$ st.$log[\psi_{STH}.issuer]$ ▷ *Logger's local data*
3: $\psi_\ell \leftarrow store.loggers[\psi_{STH}.issuer]$ ▷ *Logger's certificate*
4: **if** $C.\text{Verify}_{\psi_\ell.pk}(\psi_{STH}) \wedge |log.entries| < \psi_{STH}.size$ **then** ▷ *Valid STH*
5: $log.newSTH \leftarrow \psi_{STH}$ ▷ *Store new STH*
6: $out \leftarrow$ ('GetEntries-Req', $\psi_{STH}.issuer$,
7: $start = |log.entries|, end = \psi_{STH}.size$) ▷ *Ask for new certs*
8: **end if**
9: **return** (st, out) ▷ *Output*
end procedure

Algorithm 23 CT.GetEntries-Req: retrieve log entries

Input: local state st, local clock clk, the number of the first entry requested $start$ and the number of the last entry requested end.
Output: updated state and signed list of entries $\psi_{entries}$.

procedure CT.GetEntries-Req (st, $clk, start, end$)
1: $tbc \leftarrow (type,$ 'get-entries') ▷ *Certificate type*
2: $tbc.issuer \leftarrow$ st.ι ▷ *The issuer*
3: $tbc.from \leftarrow clk$ ▷ *When was issued*
4: $tbc.entries \leftarrow$ st.$mylog[start : end]$ ▷ *Desired certificates*
5: $\sigma \leftarrow C.\text{Certify}_{\text{st}.sk}(tbc)$ ▷ *Sign certificate*
6: $\psi_{entries} \leftarrow (tbc, \sigma)$ ▷ *Certified data*
7: **return** (st, ('GetEntries-Resp', $\psi_{entries}$)) ▷ *Output*
end procedure

Algorithm 24 CT.GetEntries-Resp: store log entries

Input: local state st, local clock clk, root store $store$ and certificate ψ.
1: **Output:** updated state.

procedure CT.GetEntries-Resp (st, $clk, store, \psi$)
2: $\psi_\ell \leftarrow store.loggers[\psi.issuer]$ ▷ *Logger's certificate*
3: **if** $C.\text{Verify}_{\psi_\ell.pk}(\psi)$ **then** ▷ *Valid certification*
4: $log \leftarrow$ st.$log[\psi.issuer]$ ▷ *Logger's local data*
5: $root \leftarrow \mathcal{MT}^h(log.entries + \psi.entries)$ ▷ *Calc new root*
6: **if** $root = log.newSTH$ **then** ▷ *Verify Merkle root*
7: $log.entries$ += $\psi.entries$ ▷ *Update local certs*
8: $log.STH \leftarrow log.newSTH$ ▷ *Update local root*
9: Optionally: examine new certificates for problems, e.g., potential phishing
10: **end if**
11: **end if**
12: **return** st ▷ *Output*
end procedure

Figure 3: CT implementation contd.

$\mathcal{P}$.Chains would need to return a separate chain for every certificate along ξ, the chains getting shorter and shorter until the trivial chain that only contains a trust anchor certificate. For example, if ψ is provided with the (short) chain $\xi[0] = \psi$, $\xi[1] = \psi'$ and $\xi[2] \in store$, then, to ensure that $\mathcal{P}.\text{Valid}(clk, \psi', store, \omega)$ would hold, we would need ω to also contain the chain ξ' s.t. $\xi'[0] = \xi[1] = \psi'$ and $\xi'[1] = \xi[2] \in store$. And this is even without considering the CRL chains required to validate ψ and ψ'. So, such an accountability predicate would be inconvenient, unnatural and inelegant, and definitely would require us to define a very strange implementations for the PKIs (and different from the real implementations).

Algorithm 25 $f_{\Delta_{\mathsf{Rev}}}$: Revocation function

Input: transcript T.

Output: $\top$ if the adversary fails to present a 'revoked-yet-valid' certificate ψ (which is valid on local clock clk although its benign issuer $\psi.issuer$ revoked it before $\tau - \Delta_{\mathsf{Rev}}$); 'Weak' if the adversary presents a 'revoked-yet-valid' certificate ψ, and the PKI identifies a fraudulent yet valid certificate which is valid with respect to a key not declared by its honest issuer; and $\perp$ if the adversary presents a 'revoked-yet-valid' certificate *and* the PKI fails to identify such a certificate.

procedure $f_{\Delta_{\mathsf{Rev}}}(T)$
  $(clk, \psi, store, \omega) \leftarrow T.out_{\mathcal{A}}$ ▷ *Extract adversary's output*
  if $\begin{bmatrix} \psi.issuer \in T.\mathsf{N} - T.\mathsf{F} \wedge \\ \mathcal{P}.\mathsf{Valid}(clk, \psi, store, \omega) \wedge \\ \textsc{RevocationRequested}(clk, \psi, T, \Delta_{\mathsf{Rev}}) \end{bmatrix}$ **then**
    if $\textsc{CanFindFraudulentCert}(T, clk, \psi, store, \omega)$ **then**
      return 'Weak'
    else
      return $\perp$
    end if
  end if
  return $\top$
end procedure
procedure $\textsc{CanFindFraudulentCert}(T, clk, \psi, store, \omega)$
  Output: $\top$ if $\mathcal{P}.\mathsf{Chains}$ returns a chain ξ which contains some 'valid yet fraudulent' public key certificate $\xi[i]$. By saying that $\xi[i]$ is fraudulent we mean that the subject $\xi[i].subject$ is a benign entity, but the key certified is *not* the key generated (and self-certified) by $\xi[i].subject$.
  for $\xi \in \mathcal{P}.\mathsf{Chains}(\psi, store, \tau, \omega)$ **do**
    if $\begin{bmatrix} \exists i \textbf{ s.t. :} \\ \mathcal{P}.\mathsf{Valid}(clk, \xi[i], store, \omega) \wedge \\ \xi[i].type = PubKey \wedge \\ \xi[i].subject \in T.\mathsf{N} - T.\mathsf{F} \wedge \\ \textsc{SelfCertified}(T, \xi[i].subject, \xi[i].pk) = \perp \end{bmatrix}$ **then return** $\top$
  end for
  return $\perp$
end procedure
procedure $\textsc{RevocationRequested}(clk, \psi, T, \Delta_{\mathsf{Rev}})$
  return $\exists e, e_R \; s.t. \; e < e_R \wedge$ ▷ *ψ issued, then revoked*
    $T.opr[e] =$ *'Certify'* $\wedge$ ▷ *e was a Certify operation*
    $T.opr[e_R] =$ *'Revoke'* $\wedge$ ▷ *e_R was a Revoke operation*
    $T.inp[e] = \psi \wedge$ ▷ *ψ issued by $\psi.issuer$*
    $T.inp[e_R] = \psi \wedge$ ▷ *ψ revoked by $\psi.issuer$*
    $T.\tau[e] \leq T.\tau[e_R] \leq clk - \Delta_{\mathsf{Rev}}$ ▷ *both at least $T.\tau[e]$ before local clock clk*
end procedure

Algorithm 26 $\pi_{\Delta_{\mathsf{Tra}}}$: the Δ_{Tra}-Transparency requirement predicate

Input: transcript T.

Output: $\perp$ if the adversary outputs a certificate ψ and auxiliary information ω, s.t. ψ is valid on local clock clk given ω, although $\psi.from \leq \tau - \Delta_{\mathsf{Tra}}$ and there exists a benign monitor ι_M, unaware of ψ at local clock clk, and monitoring a log in $\psi.logs$ since $\tau - \Delta_{\mathsf{Tra}}$ or earlier. Otherwise, the adversary failed, and the output is $\top$.

procedure $\pi_{\Delta_{\mathsf{Tra}}}(T)$
  $(clk, \psi, store, \omega, \iota_M) \leftarrow T.out_{\mathcal{A}}$ ▷ *Extract adversary's output*
  return $\perp$ if $\mathcal{P}.\mathsf{Valid}(clk, \psi, store, \omega) \wedge$ ▷ *ψ is valid certificate at τ*
    $\psi.from \leq \tau - \Delta_{\mathsf{Tra}}$ ▷ *ψ issued at least Δ_{Tra} before τ, and*
    $\textsc{IsBenignMonitor}(T, \psi, \iota_M, \tau - \Delta_{\mathsf{Tra}}) \wedge$ ▷ *ι_M monitors a log in $\psi.logs$ since $\tau - \Delta_{\mathsf{Tra}}$*
    $\textsc{MonitorIsUnaware}(T, \psi, \iota_M, \tau)$ ▷ *Yet ι_M is unaware of ψ at τ*
  return $\top$
end procedure
procedure $\textsc{IsBenignMonitor}(T, \psi, \iota_M, \tau')$
  return $\iota_M \in T.\mathsf{N} - T.\mathsf{F} \wedge$ ▷ *ι_M is benign, and*
    $\exists e, \Lambda$ **s.t.** ▷ *for some event e and set of logs Λ,*
      $T.ent[e] = \iota_M \wedge$ ▷ *ι_M was invoked*
      $T.out[e] =$ ('Monitor', Λ) $\wedge$ ▷ *to monitor set of logs Λ*
      $\psi.logs \cap \Lambda \neq \phi \wedge$ ▷ *which includes a log in $\psi.logs$*
      $T.\tau[e] \leq \tau'$ ▷ *before $\psi.\tau$*
end procedure
procedure $\textsc{MonitorIsUnaware}(T, \psi, \iota_M, \tau)$
  return $\exists e$ **s.t.** $T.opr[e] =$ *'Lookup'* $\wedge$ ▷ *Lookup operation was invoked*
    $T.ent[e] = \iota_M \wedge$ ▷ *on ι_M*
    $T.inp[e] = \psi.subject \wedge$ ▷ *w.r.t $\psi.subject$*
    $T.\tau[e] > \tau \wedge$ ▷ *sometime after $\psi.\tau + \Delta$*
    $\psi \notin T.out[e]$ ▷ *but ι_M was unaware of ψ*
end procedure

Figure 4: Remaining PKI security requirements