# ARPKI: Attack Resilient Public-Key Infrastructure

David Basin
Inst. of Information Security
Dept. of Computer Science
ETH Zurich, Switzerland
basin@inf.ethz.ch

Cas Cremers
Dept. of Computer Science
University of Oxford, UK
cas.cremers@cs.ox.ac.uk

Tiffany Hyun-Jin Kim
Carnegie Mellon University
Pittsburgh, USA
hyunjin@cmu.edu

Adrian Perrig
Inst. of Information Security
Dept. of Computer Science
ETH Zurich, Switzerland
adrian.perrig@inf.ethz.ch

Ralf Sasse
Inst. of Information Security
Dept. of Computer Science
ETH Zurich, Switzerland
ralf.sasse@inf.ethz.ch

Pawel Szalachowski
Inst. of Information Security
Dept. of Computer Science
ETH Zurich, Switzerland
psz@inf.ethz.ch

## ABSTRACT

We present ARPKI, a public-key infrastructure that ensures that certificate-related operations, such as certificate issuance, update, revocation, and validation, are transparent and accountable. ARPKI is the first such infrastructure that systematically takes into account requirements identified by previous research. Moreover, ARPKI is co-designed with a formal model, and we verify its core security property using the TAMARIN prover. We present a proof-of-concept implementation providing all features required for deployment. ARPKI efficiently handles the certification process with low overhead and without incurring additional latency to TLS.

ARPKI offers extremely strong security guarantees, where compromising $n-1$ trusted signing and verifying entities is insufficient to launch an impersonation attack. Moreover, it deters misbehavior as all its operations are publicly visible.

## Categories and Subject Descriptors

K.6.5 [**MANAGEMENT OF COMPUTING AND INFORMATION SYSTEMS**]: Security and Protection—*Authentication*; C.2.0 [**COMPUTER-COMMUNICATION NETWORKS**]: General—*Security and protection*

## General Terms

Security

## Keywords

Public-Key Infrastructure; TLS; certificate validation; public log servers; formal validation; attack resilience


*CCS'14,* November 3–7, 2014, Scottsdale, Arizona, USA.

ACM 978-1-4503-2957-6/14/11 ...$15.00.
http://dx.doi.org/10.1145/2660267.2660298 .

## 1. INTRODUCTION

In the current trust model of TLS PKI, a single compromised (or malicious) Certification Authority (CA) can issue a certificate for any domain [9,21,23,26]. Moreover, such bogus certificates can go unnoticed over long periods of time. This glaring vulnerability is widely recognized.

In response, the research community has proposed different approaches to mitigate this problem. Recent proposals include Certificate Transparency (CT) [18], which adds accountability by using log servers to make compromises visible, and the Accountable Key Infrastructure (AKI) [15] that prevents attacks by using checks-and-balances to prevent a compromised CA from impersonating domains. Although such proposals provide good starting points and building blocks, they require many interacting features to be viable and thus are inherently highly complex. History has shown that humans will miss cases when considering the security of such complex systems. Moreover, they must satisfy efficiency requirements and fit with existing business models, as well as offer improved security. Finally, even advanced proposals such as CT and AKI are still incomplete and have been designed in an ad-hoc fashion, without a formal proof of correctness. We will discuss limitations of the existing state-of-the-art further in Section 2.

We now need to take the next step and get assurance of both completeness of features as well as correctness of the security claims, which can only be achieved by using a principled approach.

**Contributions.** We present ARPKI, the first co-designed model, verification, and implementation that provides accountability and security for public-key infrastructures. In contrast to other PKI proposals, ARPKI offers:

- substantially stronger security guarantees, by providing security against a strong adversary capable of compromising $n-1$ entities at any time, where $n \geq 3$ is a system parameter that can be set depending on the desired level of security;
- formal machine-checked verification of its core security property using the TAMARIN prover; and
- a complete proof-of-concept implementation that provides all the features required for deployment and demonstrates its efficient operation.

The full implementation, formal model and security properties, and the analysis tools are available [1].

**Organization.** In Section 2 we review the state-of-the-art in PKI and in Section 3 we motivate properties that all PKI architectures should have. We introduce ARPKI in detail in Section 4 and describe its modeling and formal analysis in Section 5. We present its implementation and evaluation in Section 6 before we draw conclusions in Section 7.

# 2. PKI BACKGROUND

A variety of proposals have been made to address the security issues in X.509 PKIs and to reduce trust in the CAs. As illustrated in Figure 1, existing approaches can be classified as being client-centric, CA-centric, or domain-centric. We first look at these different approaches from a high level and then focus on AKI, which is closest to our work.

## 2.1 Alternative Approaches

**Client-centric approaches.** Proposals in this class empower clients to select dedicated entities to evaluate a certificates' correctness before accepting it. *Policy engine* [5] supports clients in defining local policies (e.g., cryptographic requirements, consistency of certificates based on an observed history, etc.) for trust decisions. This proposal is at an early stage and may rely on other proposals.

There are several proposals to create public repositories of domain certificates and enable clients to select repositories to compare the received key (of a domain) with the version stored in the repositories. *Perspectives* [29] and *Convergence* [2] fall in this category.

Although servers need no modifications, client-centric approaches require additional connections to query the repositories. This increases latency when establishing an HTTPS connection.

**CA-centric approaches.** X.509 PKI includes standards for *Certificate Revocation Lists (CRL)* [13] that are issued by CAs to prevent clients from establishing a TLS connection with domains with revoked certificates. Unfortunately, clients must be able to access the current CRLs. To resolve this online validation requirement, the *Online Certificate Status Protocol (OCSP)* [22] allows clients to check domains' certificate status by querying CAs' OCSP servers. However, OCSP has security, privacy, and performance concerns. Another approach is *Short-lived certificates (SLC)* [28] with which domains acquire certificates with short validity lifetimes and update them daily. SLC provides similar security benefits to OCSP while eliminating the need for online validation. The major drawback with CA-centric approaches is their heavy reliance on browser vendors to detect and blacklist certificates issued by compromised CAs.

**Domain-centric approaches.** Three approaches allow domain owners to actively control and protect their public keys despite CAs' potential vulnerabilities. They are based on (1) pinning, (2) DNSSEC, or (3) log servers.

Pinning-based approaches, such as *Public Key Pinning (PKP)* [3,4] and *Trust Assertions for Certificate Keys (TACK)* [19], allow a domain to declare which keys are valid for that domain such that clients "pin" the keys. However, these approaches have security vulnerabilities, such as providing no protection on the first visit to domains.

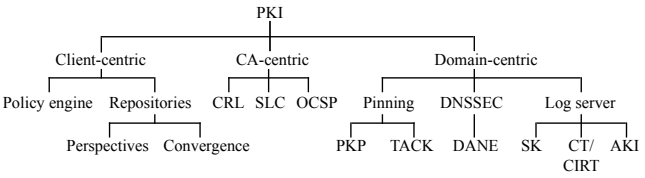


**Figure 1: Classification of PKI proposals.**

The DNSSEC-based proposal called *DNS-based Authentication of Named Entities (DANE)* [12] enables domain owners to assert certificate-specific fields on DNSSEC entries, such as a list of acceptable CAs for issuing their domain's certificates, specific acceptable certificates, or specific trust anchors to validate certificates. However, the security of DANE heavily relies on the security of DNS operators, which historically is not one of their concerns.

Log server-based approaches allow domain owners to record their certificates to public log servers, creating accountability for the CAs' actions. For example, *Sovereign Keys (SK)* [10] require domain owners to generate a sovereign key pair to sign their TLS public key and to log the sovereign key pair to read- and append-only timeline servers. Unfortunately SK requires clients to query servers, increasing latency and sacrificing privacy.

*Certificate Transparency (CT)* [18] proposes that each domain owner registers the CA-issued certificate to append-only log servers with a Merkle hash tree structure. The servers return a non-repudiable audit proof to the domain such that the domain can provide its certificate along with the audit proof to clients for a TLS connection setup. However, as CT's goal is only to make used certificates visible, it is still vulnerable to attacks when an adversary compromises a CA to create and register the fraudulent certificates, and CT does not prevent clients from accepting these certificates. Because CT itself is not designed to address certificate revocation, a supplementary system called *Revocation Transparency* was proposed [17]. Also *Certificate Issuance and Revocation Transparency (CIRT)* [24] proposes efficient revocation for CT, but it requires a client to create a new identity once its key is lost.

PoliCert [27] extends the Accountable Key Infrastructure (discussed below) by giving the domain a way to describe its own certificates and properties of TLS connections. It also includes a revocation system and a new certificate validation model. However, in this approach (as well as in previously mentioned ones), the mechanisms for detecting and disseminating log misbehavior are unspecified.

## 2.2 Accountable Key Infrastructure

We review the Accountable Key Infrastructure (AKI) [15] in more detail for two reasons. First, ARPKI is inspired by AKI's design and employs some of its concepts. Second, ARPKI addresses several shortcomings that we identified in AKI.

AKI proposes to protect domains and clients from vulnerabilities caused by single points of failure, such as a CA's root key compromise [9,21,23,26]. Through checks-and-balances among independent entities, AKI successfully distributes trust over multiple parties and detects misbehaving entities while efficiently handling certificate operations.

The AKI operates with the following three entities:

1. A **Certification Authority**, called Certification Agency in [15], authenticates domains and issues X.509 certificates.
2. To make CA-issued certificates publicly visible, an **Integrity Log Server (ILS)** maintains an Integrity Tree that logs certificates. Each ILS updates its Integrity Tree at a given interval, called `ILS_UP`.
3. Along with CAs, **validators** monitor ILS operations and detect misbehavior, such as the sudden (dis)appearance of certificates.

With these entities, the owner of a domain `A.com` defines X.509 certificate extension fields, including:

- `CA_LIST`: List of trusted CAs to sign the certificate;
- `ILS_LIST`: List of trusted ILSes to register the certificate;
- `ILS_TIMEOUT`: Timeout of an ILS's registration confirmation;
- `CA_MIN`: Minimum number of CA signatures needed to initially register and update a certificate to ILSes.

The domain owner then contacts at least `CA_MIN` trusted CAs to acquire certificates, the combination of which becomes an AKICert. After receiving a confirmation (i.e., signature) from a trusted ILS that promises to add this AKICert to its log and another confirmation from at least one validator that verifies the correct operation of the trusted CAs and ILSes, `A.com` uses the two confirmations along with the AKICert to establish TLS connections with clients.

**Integrity Trees.** Figure 2 illustrates the Integrity Tree maintained by ILSes. Integrity Trees ensure that the ILS cannot make false claims about any certificate it has or has not stored. It is implemented as a Merkle hash tree, whose leaves are lexicographically sorted by the domains' names and each parent node is computed as the hash of its two child nodes. Every leaf stores the AKICert corresponding to the given domain. After each `ILS_UP` period, the ILS updates the tree by (1) adding new entries, (2) replacing updated entries, and (3) deleting revoked and expired entries. Moreover, the ILS computes the new root hash for the current version of the tree.

This structure enables the ILS to create efficient proofs about its own content, including presence and absence proofs. To prove that AKICert exists for `A.com`, the ILS provides $h_1, AKICert_{\texttt{A.com}}, h_{10}$, and $h_{14}$. To prove that `E.com` does not have any registered AKICerts, the ILS provides the presence proofs for the immediate neighbors `D.com` and `F.com`: $h_9, AKICert_{\texttt{D.com}}, AKICert_{\texttt{F.com}}$, and $h_{14}$. Along with presence and absence proofs, the ILS provides the current signed root hash $root_i$ accompanied by the timestamp $i$ (i.e., the last time the tree was updated).

**AKI weaknesses.** AKI leaves several questions unanswered. First, AKI's setup suggests that validators can be non-profit organizations whose only incentive is to check the correctness of ILS operations. However, AKI's design implies that if validators are not continuously online or have low bandwidth, then delays will occur during the certificate registration and validation processes, contradicting AKI's claimed efficiency.

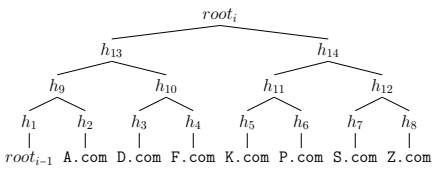


**Figure 2: Integrity Tree in $i$-th `ILS_UP` period.**

Second, parts of AKI's design are not specified in sufficient detail for a realistic implementation. In particular, it is unclear how ILSes interact during the AKICert registration and update processes, how validators monitor the ILSes, and how entities handle malicious events.

Third, AKI's security properties have not been proven in detail, and additional validation would help gain assurance in AKI's security claims. In particular, with no mature implementation available, certain edge cases are likely to have been missed which may impact the security claims. For instance, without synchronizing ILSes, an adversary can register malicious AKICerts. To attribute misbehavior, all ILSes and CAs must include the triggering messages (which are signed by the sender) for any action they perform. If the triggering requests are omitted, the ILSes and CAs will not be able to prove their correct operation to others when needed. In addition, the validators must consider both the updated entries in the Integrity Tree of the ILSes and the original request leading to the update (including its timestamp) to monitor ILSes' update handling.

Finally, AKI fails to prevent clients from accepting a compromised AKICert when an adversary successfully compromises two out of three signing entities because CAs are not actively involved in monitoring ILS or validator misbehavior. Consequently, if an adversary compromises an ILS and a validator, the compromised AKIcert stays valid until it expires, even if the domain updates its key and acquires new certificates from trusted CAs.

# 3. DESIRED PROPERTIES

In this section we state the adversary model and main security properties that all PKIs should ideally provide.

## 3.1 Adversary Model

Ideally, PKIs achieve security with respect to the strongest possible adversary (threat) model. Since PKIs operate over a possibly untrusted network, the adversary, in the worst case, can control the network. That is, we assume that the adversary can eavesdrop, modify, and insert messages at will.

We also assume that the adversary can compromise some entities, effectively obtaining their long-term secrets. However, for a PKI to satisfy any nontrivial security property, the adversary must not be able to compromise all entities. We therefore assume that the adversary can compromise the long-term secrets of some, but not all, parties.

## 3.2 Security Properties

In general PKIs should provide security, availability, and be efficient when clients authenticate domains, and these properties should hold even under the threat model described above.

**Core security property.** We first highlight the core security property that any PKI must satisfy, which prevents impersonation attacks.

- **Connection integrity.** If a client establishes a connection based on a certificate, the client must be communicating with the legitimate owner of the associated domain.

**Other security properties.** Besides the core property, PKIs should also satisfy the following security properties.

- **Legitimate initial certificate registration.** The infrastructure should register a domain's certificate only if the certificate satisfies the requirements specified by the infrastructure's policy. For example, CA-centric infrastructures allow the use of a certificate as long as it is signed by a non-revoked CA in the client browser's root CA list. As a second example, domain-centric infrastructures accept an initial certificate that is signed by (a set of) designated entities that the domain owner explicitly states to be trustworthy.
- **Legitimate certificate updates.** The infrastructure should update a domain's certificate only if the new certificate satisfies the requirements specified in the previously registered certificate.
- **Visibility of attacks.** If an adversary successfully launches an attack against the infrastructure by compromising entities, the attack should become publicly visible for detection.

## 3.3 Performance Properties

PKIs should have the following performance properties.

- **Low overhead.** The infrastructure should not substantially increase the TLS handshake message size and should have negligible impact on processing time.
- **Minimal additional latency over TLS.** The infrastructure should induce minimal (ideally zero) additional round trip latencies, possibly due to extra network requests, to the TLS handshake.

# 4. ARPKI: ATTACK RESILIENT PKI

We now present ARPKI, the end result of our co-design of model, verification, and implementation. We return to the modeling, verification and co-design aspects in Section 5 and present the implementation in more detail in Section 6.

ARPKI achieves strong security guarantees using three entities for the certificate operation: two CAs and an ILS. In particular, ARPKI's CAs conduct active on-line confirmations with validator-like capabilities. Consequently, ARPKI prevents compromise attacks such that even when $n - 1$ trusted entities are compromised, the security guarantees still hold. The description in this section is for the case of $n = 3$, but we explain its extension to arbitrarily many trusted entities in Section 4.5.

Before we go into the details, let us first give a high-level summary of the actors and their responsibilities in this scheme: a domain registers an ARPKI certificate (ARCert) for itself with the ARPKI infrastructure, and can afterwards use the resulting ARCert to securely serve webpages to clients. The CAs check the identity of the domain owner on registration and then sign and give guarantees for the presented certificate. Throughout the lifetime of the ARCert, the CAs are responsible for checking the logs for this ARCert and assuring the correct operation of other entities involved in creating the ARCert. To check the ILSes behavior, the CAs download all accepted requests from the ILSes and compare them to the published integrity trees. The ILSes keep a log of all ARCerts registered with them, in a publicly verifiable way, and provide proofs of existence for ARCerts that are then used by CAs and domains. The set of ILSes synchronizes with each other in a secure and accountable manner.

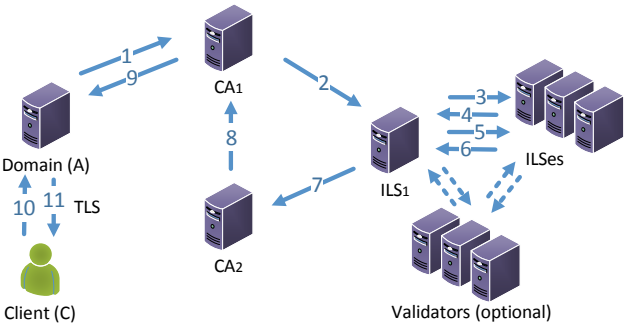


**Figure 3: Basic communication flow of ARPKI. Solid numbered lines represent the message flows to register an ARCert, and dotted arrows represent optional flows. 10 and 11 represent a TLS connection after registration is complete.**

Optionally there can be additional validators, that execute checks similar to those made by CAs, but without issuing ARCerts themselves.

We illustrate the process in Figures 3 and 4. We denote that entity $E$ signed message $M$ by $\{M\}_{K_E^{-1}}$, and $H(\cdot)$ stands for a cryptographic hash function. All signatures include timestamps and unique tags such that they cannot be mistaken for one another.

## 4.1 Initial ARCert Registration Process

**ARCert generation.** ARPKI supports trust agility, meaning that the domain owners can select their roots of trust and modify their trust decisions using extension parameters. A domain owner creates an ARPKI certificate (ARCert) by combining multiple standard certificates from trusted CAs. Note that in this step each CA checks the identity of the domain owner to authenticate domains correctly. We now consider the owner of domain $A$ registering her domain.

**ARCert registration request (Steps 1–2).** In ARPKI, three designated entities are actively involved in monitoring each other's operations. The core idea behind a Registration Request (RegReq) message is to let the domain owner explicitly designate the trusted entities, namely two CAs and one ILS ($CA_1$, $CA_2$, and $ILS_1$ in Figure 3).

ARPKI requires the domain owner to contact just one CA ($CA_1$). The main responsibilities of $CA_1$ are to validate the correctness of the other two entities' operations and act as a messenger between the domain owner and $ILS_1$ and $CA_2$.

The domain owner also designates $ILS_1$ to ensure that $\text{ARCert}_A$ is synchronized among all ILSes. $CA_2$ mainly takes the validator's role and ensures that $ILS_1$ as well as other ILSes operate accordingly, e.g., add $\text{ARCert}_A$ to their Integrity Trees as promised.

**ILS synchronization (Steps 3–6).** Ideally the same $\text{ARCert}_A$ should be publicly visible among all ILSes. However, synchronizing *all* ILSes may be inefficient, incurring significant time delay, and unrealistic. Instead, in ARPKI $ILS_1$ takes responsibility on behalf of the domain owner to synchronize $\text{ARCert}_A$ among at least a quorum of all existing

**ARCert Generation**

$A$ : Set X.509 extensions
: Contact trusted CAs, get authenticated
: Combine multiple certificates into $\text{ARCert}_A$

**ARCert Registration Request**

1. $A \rightarrow CA_1$ : $\text{RegReq} = \{ARCert_A, CA_1, ILS_1, CA_2\}_{K_A^{-1}}$
2. $CA_1$ : Verify signatures in RegReq
: Ensure $CA_1 \in \text{ARCert}_A$'s `CA_LIST`
: Add $\text{ARCert}_A$ into a pending request list
$CA_1 \rightarrow ILS_1$ : RegReq

**ILS Synchronization**

3. $ILS_1$ : Verify signatures in RegReq
: Ensure $ILS_1 \in \text{ARCert}_A$'s `ILS_LIST`
: Ensure $ILS_1$, $CA_1$, and $CA_2$ are different entities
: Ensure no ARCert was registered for A's domain
$ILS_1 \rightarrow ILS_n$: $\text{SynReq} = \{\text{RegReq}\}_{K_{ILS_1}^{-1}}$
4. $ILS_n$ : Verify signatures in RegReq
: Ensure no ARCert was registered for A's domain
$ILS_n \rightarrow ILS_1$: $\text{SynResp} = \{H(\text{RegReq})\}_{K_{ILS_n}^{-1}}$
5. $ILS_1$ : Collect SynResp from at least a quorum of ILSes
$ILS_1 \rightarrow ILS_n$: $\text{SynCommit} = \{H(\text{RegReq})\}_{K_{ILS_1}^{-1}}$
6. $ILS_n \rightarrow ILS_1$: $\text{SynAck} = \{H(\text{RegReq})\}_{K_{ILS_n}^{-1}}$

**Registration Confirmation**

7. $ILS_1$ : Collect SynAck from at least a quorum of ILSes
: $\text{Accept} = \{H(ARCert_A)\}_{K_{ILS_1}^{-1}}$
$ILS_1 \rightarrow CA_2$ : $\text{RegResp} = \{\text{Accept}, \text{RegReq}, List(\text{SynAck})\}_{K_{ILS_1}^{-1}}$
8. $CA_2$ : Verify signatures in RegResp
: Ensure $CA_2 \in \text{ARCert}_A$'s `CA_LIST`
: Ensure $ILS_1$, $CA_1$, and $CA_2$ are different entities
$CA_2 \rightarrow CA_1$ : $\text{RegConf} = \{\{\text{Accept}\}_{K_{CA_2}^{-1}}, List(\text{SynAck})\}_{K_{CA_2}^{-1}}$
9. $CA_1$ : Verify signatures in RegConf
: Ensure $ILS_1$, $CA_1$, and $CA_2$ are different entities
: Remove $\text{ARCert}_A$ from the pending request list
$CA_1 \rightarrow A$ : $\{\{\text{Accept}\}_{K_{CA_2}^{-1}}\}_{K_{CA_1}^{-1}}$
$A$ : Ensure $ILS_1$, $CA_1$, and $CA_2$ are different entities

**TLS Connection**

10. $C \rightarrow A$ : TLS connection request
11. $A \rightarrow C$ : $\text{ARCert}_A$, $\{\{\text{Accept}\}_{K_{CA_2}^{-1}}\}_{K_{CA_1}^{-1}}$

**Figure 4: Message flows for the initial ARCert registration process in Figure 3.**

ILSes.[1] This ensures that only one $\text{ARCert}_A$ is registered for the domain $A$, and the majority of the world maintains a consistent certificate entry for the domain $A$ in order to prevent impersonation attacks.

**Registration confirmation (Steps 7–9).** When the majority of ILSes agree to add $\text{ARCert}_A$ to their public Integrity Trees, $ILS_1$ schedules domain $A$'s ARCert to appear in its Integrity Tree during its next update (i.e., at the end of the current `ILS_UP` time interval), which is stated and signed in an Acceptance Confirmation (Accept) message. $ILS_1$ then sends to $CA_2$ a Registration Response (RegResp) message, which serves as a proof to $CA_2$ that $ILS_1$ (and a quorum of ILSes) indeed accepted domain $A$'s RegReq.

[1]The required quorum is one ILS more than 50% of all ILSes to allow detection, and $n$ ILSes more than 50% of all ILSes to prevent inconsistent states. Here the security parameter $n = 3$ is used, but generalized to arbitrary $n$ in Section 4.5.

**ILS Confirmation Request**

1. $A \rightarrow CA_1$ : $\text{CCReq} = \{A, CA_1, ILS_1, CA_2\}_{K_A^{-1}}$
2. $CA_1 \rightarrow ILS_1$ : CCReq

**Proof Generation**

7. $ILS_1 \rightarrow CA_2$ : $\text{Proof} = \{List(HashVal)\}_{K_{ILS_1}^{-1}}, \{Root\}_{K_{ILS_1}^{-1}}$
8. $CA_2 \rightarrow CA_1$ : $\{\{Root\}_{K_{ILS_1}^{-1}}\}_{K_{CA_2}^{-1}}$, Proof
9. $CA_1 \rightarrow A$ : $\{\{\{Root\}_{K_{ILS_1}^{-1}}\}_{K_{CA_2}^{-1}}\}_{K_{CA_1}^{-1}}$, Proof

**TLS Connection**

10. $C \rightarrow A$ : TLS connection request
11. $A \rightarrow C$ : $ARCert$, $\{\{\{Root\}_{K_{ILS_1}^{-1}}\}_{K_{CA_2}^{-1}}\}_{K_{CA_1}^{-1}}$, Proof

**Figure 5: Message flows for obtaining ARCert's log proof. Upon receiving any signed message, all entities verify the signatures (we omit these steps, focusing on the message flows).**

$CA_2$ now takes the validator's role to monitor and ensure that $ILS_1$ indeed made the majority of ILSes agree to accept $\text{ARCert}_A$ for their next update time. $CA_1$ also takes the validator's role to monitor that $CA_2$ correctly monitors $ILS_1$.

## 4.2 Clients Visiting a Domain using ARCert

**TLS connection (Steps 10–11).** The domain $A$ now has a confirmation message (Accept) that is signed by three trusted entities, and upon receiving Accept along with $\text{ARCert}_A$, clients can ensure that they are establishing a TLS connection with domain $A$.

Clients can validate an ARCert against an Accept message by verifying that the confirmation (1) is authentic, meaning the confirmation is signed by trusted entities in `ILS_LIST` and `CA_LIST`, (2) has not expired, and (3) is correct. Browsers also perform the standard validation [7,11] of every X.509 certificate in ARCert. When validation succeeds, clients accept the TLS connection to the domain $A$. The browser can store the root (confirmed by three trusted parties) for optional checks later.

## 4.3 Confirmation Renewal and Validation

Before $ILS_1$'s RegResp expires (before `ILS_TIMEOUT`) or after $ILS_1$ updates its tree (i.e., at the start of every `ILS_UP` interval), the domain owner must obtain a new proof that its ARCert is indeed logged at the ILSes. We illustrate the renewal process in Figure 5.

**ILS confirmation request (Steps 1–2).** The domain owner previously defined the trust entities in her $\text{ARCert}_A$. Unless any one of the same entities is compromised, the domain owner renews the ILS proof by contacting them.

**Proof generation (Steps 7–9).** At each $ILS_1$ update, $CA_1$ and $CA_2$ download all requests accepted by $ILS_1$ (during the last `ILS_UP` period), process it to maintain the local copy of the tree, and monitor that the root hash of each CA's local copy matches what $ILS_1$ publishes. If all the steps succeed, the domain owner receives the ILS proof that is validated by both CAs, as well as the root hash that is signed by all three entities, making themselves accountable for their actions.

**TLS connection (Steps 10–11).** Instead of the Accept message, the domain owner now provides the Proof message

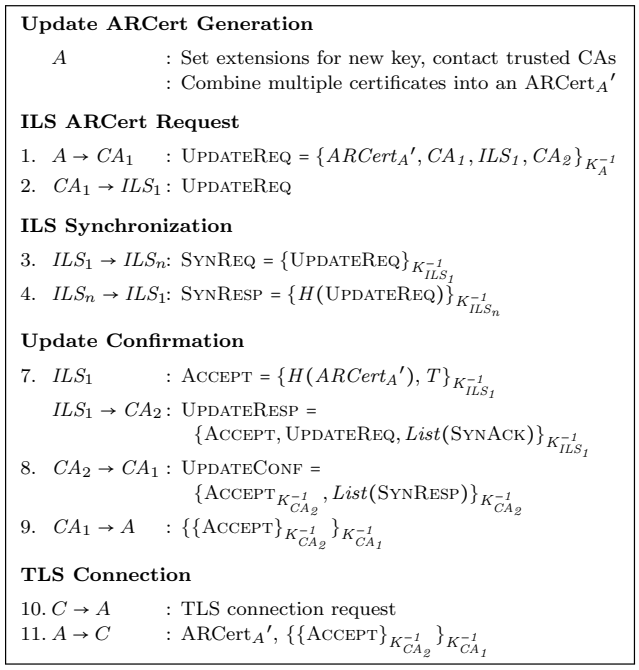


**Figure 6: Message flows for updating domain $A$'s ARCert.**

together with the Integrity Tree root (signed by all three entities) to TLS connection requests.

Clients can validate an ARCert against a confirmation, be it an ACCEPT message or a PROOF message with signed root, by following the steps outlined in Section 4.2. For example, the correctness validation of a PROOF takes the form of the browser re-computing the root of the Integrity Tree, using the intermediate hash values as specified in the PROOF and comparing with the signed root.

## 4.4 Certificate Management

ARPKI supports certificate update, revocation, and recovery from loss or compromise of private key. Figure 6 illustrates how ARPKI supports certificate update.

**Update ARCert generation.** For a proper update, the domain owner must satisfy the trust requirements that were defined in the previously registered ARCert$_A$ in $ILS_1$. For example, the domain owner's new ARCert$_A$' must be signed by `CA_MIN` number of CAs in `CA_LIST` as specified in the old ARCert$_A$. Furthermore, the three designated entities for updating the certificate must be in `CA_LIST` and `ILS_LIST` of both the old ARCert$_A$ and the new ARCert$_A$'; otherwise, the update process is delayed by cool-off periods.

**ILS request and synchronization (Steps 1–4).** To update the ARCert, $ILS_1$ proceeds with the update only if an old ARCert$_A$ exists for the domain $A$.

**Update confirmation (Steps 7–9).** $ILS_1$ confirms the replacement of ARCert$_A$ with ARCert$_A$' only when at least a quorum of all existing ILSes agree. This ensures that the world continues to have a consistent view on domain $A$'s ARCert.

The mutual checks in Steps 2, 7, and 8 in Figures 4–6 are needed for misbehavior detection of $CA_1$, $CA_2$, and $ILS_1$ during the initial ARCert registration and subsequent ARCert updates. Thus, an attack requires all three of them to be compromised, since a single non-compromised entity detects and blocks the attack.

## 4.5 Security Considerations

We have described the process for $n = 3$ above, which prevents attacks based on the compromise of at most two parties. To get stronger security guarantees, the process can be extended for larger $n$: instead of the message sent directly from $CA_2$ to $CA_1$ in Step 8 in Figure 3, additional CAs inbetween $CA_2$ and $CA_1$ receive, check, sign, and send the message to the next CA in line. Subsequently, the system as a whole provides better security guarantees as it tolerates $n-1$ compromised parties. The downside is that $n$ entities must be involved in registration, confirmation, and update, and this may cause inefficiency in the client connection later on.

Note that if an adversary can compromise $n$ entities (CAs or ILSes) in the *overall system*, the following attack is possible: Given two disjoint sets of CAs, where one set is honest and the other is compromised, if a domain successfully registered a certificate for itself using the honest CAs, we would like to guarantee that no bogus certificate can be registered for that domain by the adversary. But, if all the ILSes are compromised and willing to keep two separate logs, then the adversary can register an ARCert for the domain using the disjoint set of compromised CAs and ARPKI would not prevent this attack. However, this attack is highly likely to be detected quickly, and all the dishonest ILSes and CAs can be held accountable.

## 4.6 Novelty of ARPKI

As mentioned earlier, the AKI design inspired ARPKI. However, ARPKI introduces the following novel design aspects:

1. ARPKI introduces an entirely new role for the CAs, supporting active on-line confirmations with validator-like capabilities. This changes the dynamics of the entire system in terms of cost tradeoffs, possible resources, and incentives.
2. ARPKI offers different message flows to protect against the weakness in AKI that we identified, such as the attack based on two compromised parties. This also leads to a simpler infrastructure from the domain's perspective as $CA_1$ is the interface to the ARPKI infrastructure.
3. ARPKI supports a secure and accountable synchronization sub-protocol for ILSes.
4. ARPKI's CAs (working as validators) automatically download requests accepted by the ILS to detect subtle misbehaviour. This frees the domain from having to contact validators to verify the confirmation.
5. Initial registration is simplified compared to AKI.

# 5. MODEL AND ANALYSIS

To establish high assurance guarantees, we formally analyze ARPKI's core security property using the TAMARIN Prover [20,25]. We chose TAMARIN because it is a state-of-the-art protocol analysis and verification tool that supports unbounded verification, mutable global state, induction, and loops.

In TAMARIN, protocols are modeled using multiset rewriting rules and properties are specified using a fragment of

first-order logic that supports quantification over timepoints. TAMARIN is capable of automatic verification in many cases, and it also supports interactive verification by manual traversal of the proof tree. If the tool terminates without finding a proof, it returns a counter-example. Counter-examples are given as so-called dependency graphs, which are partially ordered sets of rule instances that represent a set of executions that violate the property. Counter-examples can be used to refine the model, and give feedback to the implementer and designer.

## 5.1 Tamarin Background

TAMARIN follows the Dolev-Yao model, where the adversary can see and block all messages, as well as see the content and manipulate messages (or their parts) that are not cryptographically protected. The execution of a security protocol in the presence of an adversary is modeled in TAMARIN using labelled multiset rewriting rules, as described below.

**States.** A state models a snapshot of a protocol's execution: the protocol participants' local states, information about fresh values, the adversary's knowledge, and the messages on the network. States are formalized as a finite multiset of terms called *facts*. There is a special fact symbol $\mathsf{Fr}$ with a fixed semantics, where $\mathsf{Fr}(n)$ models that $n$ is freshly generated. The semantics of all other fact symbols is given by the multiset rewriting rules.

**Rules.** Labeled multiset rewriting rules model the possible actions of protocol participants and the adversary, who controls and may modify messages on the network. Rules are triples written as `l -[ a ]-> r`, where `l, a, r` are all finite sequences of facts. We call `l` the *premises*, `r` the *conclusions*, and `a` the *actions*.

We employ various modeling conventions. For example, the protocol participants send messages using the $\mathsf{Out}$ fact, which the adversary adds to its knowledge $\mathsf{K}$ fact, and then can send to the protocol participants using the $\mathsf{In}$ fact. The adversary can also combine knowledge, using any operator, for example, given $\mathsf{K}(x)$ and $\mathsf{K}(y)$ and a binary operator $f$, the adversary deduces $\mathsf{K}(f(x, y))$.

**Labeled multiset rewriting.** For a given state $s$, for each rule of the form `l -[ a ]-> r`, if there exists a substitution $\sigma$ such that $\sigma(\mathtt{l}) \in s$, then the rule can be triggered, resulting in a new state $s' = (s \setminus \sigma(\mathtt{l})) \cup \sigma(\mathtt{r})$. Each time a rule is triggered, $\sigma(\mathtt{a})$ is appended to the trace, which acts like a log.

An execution is an alternating sequence of states and multiset rewriting rules, where the initial state is empty, and a state is followed by its successor state using the rule in-between them. The trace of the execution is then the list of the actions of the rules used in the sequence. Actions are ordered sequentially and timestamped by the timepoint when they occur. Properties are defined in a fragment of first order logic and can refer to the actions in the trace and their order. For more details, see [20,25].

If a counterexample to the specified property is found, TAMARIN's GUI shows this as a graph of instantiated rules and their connections. The graph includes the modeled rules as well as the built-in adversary rules. Edges denote the connections between facts produced by the right hand side of rules and consumed by the left hand side of rules.

## 5.2 Tamarin Extensions

The size and complexity of ARPKI substantially surpassed all protocols previously modeled with TAMARIN. This required several improvements to the TAMARIN tool chain.

First, protocols can now be specified using macros for terms, which are used for repeating or large terms. These macros, which may be nested, are expanded using the C preprocessor. This change increased modeler productivity and model maintainability. On the output side, we added functionality to TAMARIN's GUI that allows a compact representation of the huge output graphs that result from the ARPKI model. This makes it easier to understand attacks found by the tool. Finally, we introduced additional means for the user to guide the proof search by annotating rules with a measure of their relevance for the proof. These annotations can help TAMARIN find a proof in cases where its default heuristics fail.

## 5.3 Modeling ARPKI

We modeled the communication flow of ARPKI for the initial ARCert generation and registration according to Figures 3 and 4.

**Abstractions used.** We employed several abstractions during modeling. We used lists instead of Merkle hash trees to store the registered certificates. As we do not give the adversary the ability to tamper with these lists, and all protocol participants only access them in the designated way, this encodes the assumption that the hash tree cannot be tampered with. However, the adversary can create such lists (representing hash trees) himself by compromising parties and using their long-term private keys to sign the lists. We model signatures using a signing operator `sign` with a private key as one of the parameters.

**Model excerpt.** The full TAMARIN model for ARPKI is available [1] and contains 23 rules taking around 1000 lines, and is roughly 35000 characters before macro expansion, and 54000 characters after macro expansion. Even though in principle our model allows arbitrarily many CAs and ILSes, for the analysis we restrict ourselves to the minimal case of two CAs and one ILS.

We present a simplified fragment of the rules to explain the model's key elements. The two rules below model the execution of a domain that wants to register an ARCert for its use, where it requests two CAs to sign off on the new public key before it combines them. The state fact `DomainCombineARCertA` connects the two rules, so that the second rule can only be triggered if the first rule was previously triggered and messages of the expected form are available.

```
rule A_Create_AR_Cert:
  let ILSlist = $ILSk
      pkA = pk(~ltkA) in
    [ !Ltk($A, ~ltkA), F_CERT($A,pkA) ]
 --[ OnlyOne('A_Create_AR_Cert')
   , AskedForARCert($A,~ltkA) ]->
   [ DomainCombineARCertA($A, CertA, $CA1, $CA2)
   , Out(<$A, $CA1, SIGNREQ>), Out(<$A, $CA2, SIGNREQ>)]
```

In the above rule, we model the ILS list as a single public name of an arbitrary ILS called ILSk. The $ prefix denotes that ILSk is of type 'public name' and the ~ prefix denotes terms of type 'fresh', i.e., freshly generated terms. Additional annotation of the type of each entity and timestamps have been omitted here and throughout this section. Fact

symbols with the ! prefix are never consumed and can be used repeatedly.

To model asymmetric keys, we use fresh (unique) terms as long-term private keys (here: ~ltkA) and use an abstract one-way function $pk$ that yields the corresponding public key.

```
rule A_Receive_SignedCerts:
  let contactCA = $CA1 in
  [ DomainCombineARCertA($A, CertA, $CA1, $CA2) ,
    In(<$CA1, $A, SIGCert1>) , In(<$CA2, $A, SIGCert2>)]
--[ NotEq(~ltkCAx1, ~ltkCAx2)
  , ReceivedCASignedARCert($A,~ltkA) ]->
  [ DomainHasARCertA($A, contactCA, ARCertA) ]
```

We use the action NotEq(x,y) to specify that the rule can only be triggered for two different CAs. This concludes the ARCert generation as described at the top of Figure 4.

We next describe the remaining message flows for the domain registering the new ARCert and the creation of the registration confirmation by $CA_1$, $CA_2$, and ILS, ignoring the ILS synchronization as that is not part of our model. In Figure 4 these message flows are given by Steps 1–2 and Steps 7–9. Each such step is a message exchange between two parties, the sender and receiver. In the following step the previous receiver becomes the sender of the next message. As each rule is written from the point of view of one participant, we simply combine the receiving of one message and the sending of the next message into a single rule.

First, rule ILS_REG_A1 models sending the message from Step 1. Note that the presence of private keys does not mean that the participant actually knows the long-term private keys of $CA_1$ or $CA_2$, but rather that it can check that the correct signatures are used for those. The OnlyOne action guarantees that for each ARCert, represented by its key ~ltkA, this registration will only be started once by an honest participant. The state fact DomainHasARCertA connects this rule with the previous one (initial generation) and the final rule which receives the fully signed ARCert. Note that the message being sent, represented as RegReq, hides much of the complexity of generating and checking the content of messages.

```
rule ILS_Reg_A1:
  let ILSlist = $ILSk in
  [ DomainHasARCertA($A, $CA1, ARCertA)
  , !Ltk($A, ~ltkA), !Ltk($CA1, ~ltkCAx1)
  , !Ltk($CA2, ~ltkCAx2) ]
--[ OnlyOne(< 'ILS_Reg_A1' , ~ltkA >) ]->
  [ Out(< $A, $CA1, RegReq >),
    DomainHasARCertA($A, $CA1, ARCertA) ]
```

The rule ILS_REG_CA1_FORWARD models receiving the message from Step 1 and sending the message of Step 2. Note that $CA_1$ matches the contact CA in the ARCert. $CA_1$ keeps state of the received message in ContactCAStateILSReg and does additional checks later.

```
rule ILS_Reg_CA1_Forward:
  [ In(< $A, $CA1, RegReq >) ] --[ ]->
  [ Out(< $CA1, $ILSk, RegReq >)
  , ContactCAStateILSReg($A, $CA1, RegReq) ]
```

Next, rule ILS_REG_ILS receives the message from Step 2 and sends the message of Step 7 as an ILS. The two CAs are bound inside the message that is received and the private keys of $CA_1$ and $CA_2$ are again exclusively used for signature verification. The action OnlyOne ensures that each domain can only be registered once at the ILS. The state fact ILS-toAdd stores the list of new ARCerts that must be added. The message MSG1 that is sent out is a macro that expands to contain the relevant parts of the registration request RegReq and is signed by the ILS.

```
rule ILS_Reg_ILS:
  [ In(< $CA1, $ILSk, RegReq >) , !Ltk($ILSk, ~ltkK)
  , !Ltk($CA1, ~ltkCAx1) , !Ltk($CA2, ~ltkCAx2)
  , ILStoAdd(~ltkK, $ILSk, AddList) ]
--[ OnlyOne(< 'RegisterDomain', $A >) ]->
  [ Out(< $ILSk, $CA2, MSG1 >)
  , ILStoAdd(~ltkK, $ILSk, AddList + ARCertA) ]
```

The rule ILS_REG_CA2 receives the message of Step 7 and sends the message of Step 8. Again, the private keys of ILS and $CA_1$ are used only for signature verification. $CA_2$ here must take responsibility to only allow the registration of a domain once. The message MSG1 is as described previously, and MOK1 is an extension that is signed by $CA_2$.

```
rule ILS_Reg_CA2:
  [ In(< K, $CA2, MSG1 >), !Ltk($CA2, ~ltkCAx2)
  , !Ltk($ILSk, ~ltkK), !Ltk($CA1, ~ltkCAx1) ]
--[ OnlyOne(< 'RegisterDomainSecondCA', $A >) ]->
  [ Out(< $CA2, $CA1, MOK1 >) ]
```

The rule ILS_REG_CA1 receives the message of Step 8 and sends the message for Step 9. Again, the private keys of ILS and $CA_2$ are used only for signature verification. Here, $CA_1$ ensures that the domain is only registered once. The message sent out, MOK2, again extends MOK1 with a signature, and is the proof of registration that will be used later by the domain.

```
rule ILS_Reg_CA1:
  [ In(< $CA2, $CA1, MOK1 >), !Ltk($CA1, ~ltkCAx1)
  , !Ltk($ILSk, ~ltkK), !Ltk($CA2, ~ltkCAx2)
  , ContactCAStateILSReg($A, $CA1, RegReq) ]
--[ OnlyOne(< 'RegisterDomainFirstCA', $A >)
  , AcceptedARCert($A, ~ltkA) ]->
  [ Out(< $CA1, $A, MOK2 >) ]
```

Finally, the rule ILS_REG_A2 receives the MOK2 message of Step 9, i.e., as the registering domain. Here, the private keys of both CAs and the ILS are used just for signature verification. The domain also checks that the two CAs and the ILS are distinct entities using NotEq.

```
rule ILS_Reg_A2:
  [ DomainHasARCertA($A, $CA1, ARCertA)
  , !Ltk($CA1, ~ltkCAx1), !Ltk($CA2, ~ltkCAx2)
  , !Ltk($ILSk, ~ltkK), In(< $CA1, $A, MOK2 >) ]
--[ HasARCertNoLog($A, ~ltkA, ARCertANotVerified)
  , NotEq(~ltkCAx1, ~ltkCAx2) , NotEq(~ltkCAx1, ~ltkK)
  , NotEq(~ltkCAx2, ~ltkK)
  , ReceivedARCert($A, ~ltkA) ]->
  [ Out( ARCertANotVerified )
  , StoredARCertA($A, ARCertA, ARCertANotVerified) ]
```

The state fact StoredARCertA is used to store the ARCert that will subsequently be used by the domain for all connection requests.

## 5.4 Adversary Model

By default, TAMARIN's adversary model assumes that the adversary has full network control. All messages sent using Out(m) facts in the right hand side of rules are added to

```
lemma main_prop:
  "( All cid a b reason oldkey key  #i1 #i2 #i3 #i4 .
       ( GEN_LTK(a,oldkey,'trusted') @i1             // 'Honest' agent
       & AskedForARCert(a,oldkey) @i2                // domain has asked for a ARCert with this exact key
       & ReceivedARCert(a,oldkey) @i3                // domain has confirmation that its ARCert with this
                                                     //    exact key has been processed.
       & ConnectionAccepted(cid,b,a,reason,key) @i4  // browser accepted connection, based on private key
                                                     //   'key' in for domain a.
       & i3 < i4)
       ==>
       ( (not (Ex #j. K(key) @j)) )                  // adversary cannot know that private key
   ) "
```

**Figure 7: Main security property proven**

the adversary's knowledge, and any message that can be constructed from this knowledge can be used to trigger an `In(m)` fact in the left hand side of a rule. Thus, the adversary can eavesdrop, modify, and insert messages.

Additionally we assume that the adversary can compromise ILSes and CAs. For the main security property, we assume that the adversary has compromised at most two such entities. It is clear that for any design, if the adversary can compromise all involved entities (here: two CAs, one ILS) that the browser trusts, he can convince the browser that a certificate is good. We model this by adding rules that enable the adversary to register public keys that are later designated as a CA or an ILS. A compromised ILS will then sign any integrity tree represented as a list as usual.

## 5.5 Analysis Guarantees

**Proof goal.** Whenever (i) a domain $A$ has been registered initially by an honest party with an ARCert; and (ii) a browser later accepts a connection to domain $A$ with some ARCert (which may have been updated and hence differ from the original ARCert), then the adversary does not know the private key for that ARCert.

We require that the adversary does not know the private key for the ARCert to model that the browser communicates with the right domain, because for a bogus certificate the adversary would know the private key. The part (i) makes explicit the assumption that until a domain has had an ARCert issued using ARPKI, anyone can register that domain themselves, including the adversary, as long as they can fool (or have compromised) two CAs.

We analyze this proof goal twice: once for at most two compromised entities, and once for three or more compromised entities. The formula that encodes this property in Tamarin is shown in Figure 7 and takes the form of an implication. The formula starts with a quantification over variables $(cid, a, b, \ldots)$: for all values of those variables there should be a `GEN_LTK(...)` action in the trace at position $i1$ ($\#i1$ denotes a variable $i1$ of type 'timepoint'). In our model, this action can only be produced by a particular rule that generates initially trusted keys. If a domain $A$ has initially received an ARPKI certificate using a non-compromised private key, and the browser accepted a connection for that domain for any key pair (uniquely determined by the private key $key$), then the implication holds if the adversary does not know the private key $key$ (encoded by $K(key)$).

This is precisely the connection integrity property from Section 3. As we will show next, it holds for ARPKI whenever up to two entities are compromised, and can be broken only with three or more compromised entities.

**Analysis.** Using Tamarin, we find the expected attack for the case of three or more compromised CAs and ILSes. An adversary that, for example, controls two CAs as well as one ILS can create an ARCert for any domain. But, when at most two entities are compromised, Tamarin verifies the lemma. This guarantees that no attack with less than three compromised parties is possible.

We ran our experiments on a PC with an Intel Xeon CPU (2.60GHz) with 16 cores and 32 GB of RAM with Ubuntu 14.04 64bit as the operating system. The proof runtime with at most two compromised entities was 78 minutes, and the runtime for finding the attack with three or more compromised entities was 52 minutes. We had to develop extensions and provided hints to Tamarin as indicated in Section 5.2. We estimate the overall verification effort at several person months.

## 5.6 Co-design

We developed our formal specification of ARPKI in tandem with its implementation, working from a single evolving design document. As a result, the specification and the implementation are tightly linked, significantly reducing the possibility of modeling errors.

Tamarin played a critical role during the development in helping us make all details of the protocol design precise and in uncovering missing details. During development, we found a number of attacks on early designs, even when limited to two compromised parties. For example, in one case, we discovered that checks done on the browser side, to protect against a party signing more than once, were missing during certificate creation, where the domain owner did not perform these checks. The missing checks were then added to the model, the specification document, and the implementation.

Once the formal model had stabilized, further issues found in failed proof attempts were quickly communicated and fixed on all sides.

# 6. IMPLEMENTATION

In this section we describe our proof-of-concept implementation of ARPKI and assess its performance. Our prototype provides all the features required for deployment. In particular, we implemented the following parts: *(a)* the communication flows and processing logic for the message exchanges presented in Section 4, *(b)* the ILS process with the fully implemented Integrity Tree, and the capability to publish the information required for its consistency checks, *(c)* the validator process, which monitors the ILSes and publishes misbehaviors, *(d)* the CA process that includes the validator's role with the additional ability to produce on-line confirmations, *(e)* the client process, i.e., a browser extended

with support for full ARCert validation, *(f)* the protocol for accountable synchronization, and *(g)* the domain tool, which can register, update, revoke, recover, and confirm ARCerts.

## 6.1 Implementation Choices

We implement the ARCert certificates using a concatenation of standard X.509 certificates, where we used X.509 extensions [8] to introduce ARPKI-related fields such as `CA_LIST`, `ILS_LIST`, `ILS_TIMEOUT`, and `CA_MIN`.

The implementation of all processes is written in C++. We use OpenSSL (version 1.0.1) APIs for all cryptographic operations, and use the JavaScript Object Notation (JSON) and Base64 encoding for request and response messages. We implement the Integrity Tree using SHA-512 as the hash function, and use RSA-2048 as the signature algorithm.

Entities are implemented in a modular manner, where every module provides its own API. The communications module, which receives and serves requests, is implemented using a multi-threaded work queue with TCP sockets. The Integrity Tree is implemented as a separate database module, while the *publishing module* is realized by a local HTTP server. The ILS publishes all accepted requests and the root hash at each update time. This module also allows every entity to publish detected misbehaviors.

We implement the ARPKI TLS server by extending the Nginx HTTP server (version 1.5.7). We configure the TLS server to periodically interact with ARPKI's infrastructure to provide fresh confirmations to the browser. After at most every `ILS_TIMEOUT` (expected to be a few hours), the server sends a request CCReq and receives fresh confirmations, i.e., either Proof or Accept. The received confirmations are validated and saved for future HTTPS connections.

We implement the client by extending the Chromium web browser and we deploy our system without significant changes to the server code and TLS protocol. During the client-server connection, the server's ARCert is sent within the handshake's *Server Certificate* message while confirmations are provided to the browser using the existing *Online Certificate Status Protocol Stapling* extension. This architecture includes the ARCert validation process from Section 4, so the browser verifies the ARCert and the signatures of the received confirmation. The browser additionally verifies the consistency of the ARCert and the confirmation.

## 6.2 Performance Evaluation

We analyzed the performance of our prototype implementation in a real-world scenario. We set up a testbed that included all entities: the ILS, CAs, validators, ARPKI-supporting server, and browser. We ran our tests on a PC with an Intel i5-3470 (3.20GHz) CPU and 16GB of RAM with Ubuntu 12.04 64bit as the operating system. On this machine we ran three virtual machines, acting as $CA_1$, $CA_2$, and $ILS_1$, respectively.

First we investigated how long it takes for the infrastructure to process three requests initiated by the domain: RegReq, UpdateReq, and CCReq. Note that these actions are infrequently performed by our infrastructure for any given domain. RegReq is sent once, while UpdateReq is, in general, envisioned to be sent annually. The most common request is CCReq, which is sent roughly every `ILS_TIMEOUT`. Measurements are given as the average over 1000 test runs, and the results are presented in Table 1. The total processing time is calculated from the time that the domain sends the request to the time that $CA_1$ sends the response (which went through all required entities), without considering network latency. In other words, we measure the total processing time spent by the entities involved.

**Table 1: Total processing time (in milliseconds) required for the given request by the given entity.**

| Request | $CA_1$ | $CA_2$ | $ILS_1$ | Total |
|---|---|---|---|---|
| RegReq | 9.31 | 9.28 | 13.56 | 32.15 |
| UpdateReq | 9.49 | 9.33 | 12.98 | 31.80 |
| CCReq | 5.12 | 5.64 | 7.06 | 17.82 |

**Table 2: Detailed processing time (in milliseconds) by the given entity.**

| Request | | RSA | ARCert | Hash | Misc. |
|---|---|---|---|---|---|
| $CA_1$ | RegReq | 5.80 | 1.72 | 0.18 | 1.61 |
| | UpdateReq | 5.90 | 1.69 | 0.19 | 1.71 |
| | CCReq | 2.39 | - | 0.22 | 2.51 |
| $CA_2$ | RegReq | 5.97 | 1.70 | 0.24 | 1.37 |
| | UpdateReq | 5.72 | 1.93 | 0.24 | 1.44 |
| | CCReq | 2.40 | - | 0.25 | 1.28 |
| $ILS_1$ | RegReq | 10.09 | 1.38 | 0.37 | 1.72 |
| | UpdateReq | 9.32 | 1.40 | 0.36 | 1.90 |
| | CCReq | 5.68 | - | 0.15 | 1.24 |

We provide a further breakdown of the processing time in Table 2. For each entity, we distinguish four kinds of processing time. **RSA** denotes the time spent on signature verification and creation, required for creating and processing protocol messages. **ARCert** denotes the time spent on the verification and parameter validation of the X.509 certificates. **Hash** denotes the time required to compute hashes. **Misc.** encompasses message processing (Base64 and JSON encoding/decoding and parsing), database lookup, and input/output operations. For the CCReq, no ARCert validation needs to be done, which we denote by a hyphen in the table.

For the validation by the browser, we distinguish two phases: *standard validation* and *ARPKI validation*. During the standard validation phase, the browser validates every X.509 certificate within ARCert, using the standard browser validation procedure. This includes checking whether the certificate is issued for a correct domain, has been signed correctly, has expired, etc. The validation phase of ARPKI additionally checks that *(a)* certificates within ARCert are signed by CAs that are trusted by the domain, *(b)* proofs have been produced for the correct ARCert and that the proof validates with the correct root, and *(c)* the proof and the root are signed by the correct entities (i.e., they are distinct and trusted by the domain). We used an ARCert that consisted of three standard X.509 certificates. The entire validation took 2.25ms on average (the median was 2.20ms); the standard validation took 0.70ms on average (median 0.67ms), and the ARPKI validation took on average 1.55ms (median 1.53ms).

The most time-consuming operations in our system involve signature creation and verification. This computational overhead can be reduced by using state-of-the-art digital signature schemes [6,14]. However, such improvements may

not be backward-compatible with software that uses older cryptographic libraries.

In our design, the CAs are required to perform verification in addition to their normal operations. Even though our prototype is not yet optimized, our tests indicate that a CA on a single low-end machine can serve about 100 ARCert's registrations/updates and 200 confirmations per second. The bandwidth required for this is 10Mbit/s.

In terms of client-server communication, the biggest transmission overhead is introduced by using the ARCert, since it is implemented by concatenating standard X.509 certificates. Instead of sending one standard certificate, as is currently done, a domain sends the concatenation of standard certificates, each signed by a different CA. Note that the size of this overhead is not fixed: the domain can adjust the trade-off between processing/transmission overhead and the authentication of its own public key by combining the desired number of standard certificates into an ARCert. It is important that the latency introduced by the ARPKI infrastructure does not influence the client-server connection. The confirmations are obtained periodically and stored by the server for a selected amount of time. At each connection, the server provides these confirmations to the browser along with its ARCert. Our solution does not introduce any extra network requests for client-server connections. However, due to the size of ARCert and confirmations, small latency may be introduced by the transport layer protocol [16]. Note that our solution does not introduce any additional computational overhead for the server during regular HTTPS connections.

Overall, our analysis of the prototype implementation indicates that it is feasible to deploy ARPKI with reasonable overhead.

# 7. CONCLUSIONS

We have presented ARPKI, a new public-key infrastructure that offers very strong security guarantees. In particular, it offers resilience against impersonation attacks that involve $n-1$ compromised entities. Moreover, if all entities involved in an ARCert are compromised, in which case domain impersonation cannot be prevented, the validators may still obtain the evidence of the compromise, and can take compensating actions out of band. If such evidence cannot be obtained (an adversary uses compromised keys to produce an ARCert and its confirmation, without logging this malicious ARCert), then only an attacked client can make that attack detectable by contacting validators out of band. Even though attack resilience cannot be achieved in this case, complete compromise situations are at least visible. We have also implemented our proposal. Our evaluation of the complete proof-of-concept implementation provides strong evidence of the feasibility of deploying ARPKI with reasonable overhead.

Throughout the design and implementation of ARPKI, we used formal analysis to validate our design modifications. This co-design of the formal model and the implementation enabled us to detect numerous pitfalls early on. It also enabled us to make implementation choices that simplified the construction of proofs later, such as including unique tags in all messages. As a result, our formal model is much closer to the implementation than a typical after-the-fact analysis of a given implementation, thereby reducing the possibility of modeling errors.

Finally, ARPKI introduces a new model of public-key infrastructure and certificate validation. Future work therefore includes developing procedures for managing CA certificates, elaborating the CAs' policies and business models, improving the representation of ARCert, and developing incremental deployment strategies.

## Acknowledgments

This work was supported by CyLab at Carnegie Mellon University, NSF under award CNS-1040801, and a gift from Google. We thank Emilia Kasper for her feedback during the initial stage of this work. We thank Lorenzo Baesso and Lin Chen for their programming assistance.

# 8. REFERENCES

[1] ARPKI: Full implementation, formal model, and security properties. `http://www.netsec.ethz.ch/research/arpki`.

[2] Convergence. `http://convergence.io/`.

[3] Public Key Pinning. `http://www.imperialviolet.org/2011/05/04/pinning.html`, May 2011.

[4] Public Key Pinning Extension for HTTP. `http://tools.ietf.org/html/draft-ietf-websec-key-pinning-01`, December 2011.

[5] Martín Abadi, Andrew Birrell, Ilya Mironov, Ted Wobber, and Yinglian Xie. Global authentication in an untrustworthy world. In Petros Maniatis, editor, *HotOS*. USENIX Association, 2013.

[6] Daniel J Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, 2012.

[7] Robert Biddle, Paul C van Oorschot, Andrew S Patrick, Jennifer Sobey, and Tara Whalen. Browser interfaces and extended validation SSL certificates: an empirical study. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 19–30. ACM, 2009.

[8] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280 (Proposed Standard), May 2008.

[9] Paul Ducklin. The TURKTRUST SSL certificate fiasco - what really happened, and what happens next? `http://nakedsecurity.sophos.com/2013/01/08/the-turktrust-ssl-certificate-fiasco-what-happened-and-what-happens-next/`, January 2013.

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

[11] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: validating SSL certificates in non-browser software. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 38–49. ACM, 2012.

[12] Paul Hoffman and Jakob Schlyter. The DNS-based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA. `http://tools.ietf.org/html/rfc6698`, August 2012. IETF RFC 6698.

[13] R. Housley, W. Polk, W. Ford, and D. Solo. Internet X.509 Public Key Infrastructure: Certificate and Certificate Revocation List (CRL) Profile. Technical Report RFC 3280, Internet Engineering Task Force, April 2002.

[14] Emilia Kasper. Fast elliptic curve cryptography in OpenSSL. In *Financial Cryptography and Data Security*, volume 7126 of *LNCS*, pages 27–39. Springer, 2012.

[15] Tiffany Hyun-Jin Kim, Lin-Shung Huang, Adrian Perrig, Collin Jackson, and Virgil Gligor. Accountable Key Infrastructure (AKI): A Proposal for a Public-Key Validation Infrastructure. In Daniel Schwabe, Virgílio A. F. Almeida, Hartmut Glaser, Ricardo A. Baeza-Yates, and Sue B. Moon, editors, *Proceedings of the International World Wide Web Conference (WWW)*, May 2013.

[16] Adam Langley. Overclocking SSL. `https://www.imperialviolet.org/2010/06/25/overclocking-ssl.html`, June 2010.

[17] Ben Laurie and Emilia Kasper. Revocation Transparency. `http://sump2.links.org/files/RevocationTransparency.pdf`.

[18] Ben Laurie, Adam Langley, and Emilia Kasper. Certificate Transparency. `http://tools.ietf.org/pdf/rfc6962.pdf`, June 2013. IETF RFC 6962.

[19] Moxie Marlinspike and Trevor Perrin. Trust Assertions for Certificate Keys. `http://tack.io/draft.html`, May 2012.

[20] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In *Computer Aided Verification, 25th International Conference, CAV 2013, Princeton, USA, Proc.*, volume 8044 of *LNCS*, pages 696–701. Springer, 2013.

[21] Joseph Menn. Key internet operator VeriSign hit by hackers. `http://www.reuters.com/article/2012/02/02/us-hacking-verisign-idUSTRE8110Z820120202`, January 2012.

[22] M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams. X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP. Internet Request for Comments 2560, June 1999.

[23] Paul Roberts. Phony SSL certificates issued for Google, Yahoo, Skype, others. `http://threatpost.com/phony-ssl-certificates-issued-google-yahoo-skype-others-032311/`, March 2011.

[24] Mark D. Ryan. Enhanced certificate transparency and end-to-end encrypted mail. In *Proceedings of NDSS*. The Internet Society, 2014.

[25] Benedikt Schmidt, Simon Meier, Cas Cremers, and David Basin. Automated analysis of Diffie-Hellman protocols and advanced security properties. In *Computer Security Foundations Symposium (CSF)*, pages 78–94. IEEE, 2012.

[26] Toby Sterling. Second firm warns of concern after Dutch hack. `http://news.yahoo.com/second-firm-warns-concern-dutch-hack-215940770.html`, September 2011.

[27] Pawel Szalachowski, Stephanos Matsumoto, and Adrian Perrig. PoliCert: Secure and Flexible TLS Certificate Management. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. ACM, November 2014.

[28] Emin Topalovic, Brennan Saeta, Lin-Shung Huang, Collin Jackson, and Dan Boneh. Towards Short-Lived Certificates. In *Web 2.0 Security and Privacy*, May 2012.

[29] Dan Wendlandt, David G. Andersen, and Adrian Perrig. Perspectives: Improving SSH-style Host Authentication with Multi-Path Probing. In *Proceedings of USENIX Annual Technical Conference*, June 2008.