

# Design, Analysis, and Implementation of ARPKI: an Attack-Resilient Public-Key Infrastructure

David Basin, Cas Cremers, Tiffany Hyun-Jin Kim, Adrian Perrig, Ralf Sasse, Pawel Szalachowski

**Abstract**—The current Transport Layer Security (TLS) Public-Key Infrastructure (PKI) is based on a weakest-link security model that depends on over a thousand trust roots. The recent history of malicious and compromised Certification Authorities has fueled the desire for alternatives. Creating a new, secure infrastructure is, however, a surprisingly challenging task due to the large number of parties involved and the many ways that they can interact. A principled approach to its design is therefore mandatory, as humans cannot feasibly consider all the cases that can occur due to the multitude of interleavings of actions by legitimate parties and attackers, such as private key compromises (e.g., domain, Certification Authority, log server, other trusted entities), key revocations, key updates, etc. We present ARPKI, a PKI architecture that ensures that certificate-related operations, such as certificate issuance, update, revocation, and validation, are transparent and accountable. ARPKI efficiently supports these operations, and gracefully handles catastrophic events such as domain key loss or compromise. Moreover ARPKI is the first PKI architecture that is co-designed with a formal model, and we verify its core security property using the TAMARIN prover. We prove that ARPKI offers extremely strong security guarantees, where compromising even  $n - 1$  trusted signing and verifying entities is insufficient to launch a man-in-the-middle attack. Moreover, ARPKI's use deters misbehavior as all operations are publicly visible. Finally, we present a proof-of-concept implementation that provides all the features required for deployment. Our experiments indicate that ARPKI efficiently handles the certification process with low overhead. It does not incur additional latency to TLS, since no additional round trips are required.



## 1 INTRODUCTION

Transport Layer Security (TLS) has been a tremendous success and is globally used to secure web-based communication. Given that the security of the majority of network-based financial and commercial transactions relies on TLS, one would hope that its security is commensurate with its widespread acceptance and use.

Unfortunately, many TLS attack vectors exist and recently several high-profile attacks have demonstrated its vulnerability in practice. In particular, in the current trust model of TLS PKI, a single compromised (or malicious) Certification Authority (CA) can issue a certificate for any domain [1]–[4], i.e., a realm of administrative autonomy with an associated unique identification string, called domain name. Moreover, such bogus certificates can go unnoticed over long time periods. These glaring weaknesses are widely recognized.

Unfortunately, designing a secure and viable PKI architecture is more involved than one may imagine, as complex corner cases must be handled. On the one hand, adversarial events such as CA private key compromise or domain private key compromise must be addressed. On the other hand, legitimate events such as switching to different CAs or key replacement after private key loss must be supported. For example, the legitimate replacement of a key pair and certificate after private-key loss may appear to be an impersonation attempt. Also, legitimately switching to a new CA to cease using a compromised CA that signs fraudulent certificates may also appear to be a malicious event. Hence, a

PKI architecture must prevent attacks, yet gracefully handle legitimate key and certificate management events.

The research community has proposed new PKI architectures to address these issues. Recent proposals include Certificate Transparency (CT) [5] and Sovereign Keys [6], which add accountability by using log servers to make compromises visible, and the Accountable Key Infrastructure (AKI) [7] 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 entities and thus are inherently highly complex. History has shown that humans will miss cases when considering the security of such complex systems. Moreover, a PKI architecture 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 (as they do not handle all corner cases in the certificate life cycle) and have been designed in an ad-hoc fashion, without a formal proof of correctness. We will discuss the limitations of the existing state-of-the-art further in Section 3.

We now need to take the next step and gain assurance about both the completeness of the features used as well as correctness of the security claims, which can only be achieved by using a principled approach. We present the Attack-Resilient Public-Key Infrastructure (ARPKI), the first co-designed PKI model, verification, and implementation that provides accountability and security for public-key infrastructures. ARPKI integrates an architecture for key revocation for all entities (e.g., CAs and domains) with an architecture for accountability of all infrastructure parties through checks-and-balances. ARPKI efficiently handles common certification operations and gracefully handles catastrophic events such as domain key loss or compromise.

- D. Basin, A. Perrig, R. Sasse, and P. Szalachowski are with the Dept. of Computer Science, ETH Zurich, Switzerland.
- C. Cremers is with the Dept. of Computer Science, University of Oxford.
- T. Kim is with HRL Laboratories LLC.

To reduce trust in any single entity, we leverage globally visible directories (i.e., public log servers) that enable public integrity validation for certificate information. Such public validation provides accountability for CA actions, and thus creates deterrence against fraudulent CA activities. To reduce the number of trusted CAs, as well as the trust placed in any single CA, a domain can define which and how many CAs are required to update its certificate. To enable recovery from unanticipated events, certificates can be updated through another set of CAs; however, the certificates become active only after a domain-specified hold time. In case of fraudulent updates, legitimate domains can react during the hold time to have the fraudulent certificate removed.

**Contributions.** In contrast to other 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;
- the formal machine-checked verification of its core security property using the TAMARIN prover; and
- a complete implementation that provides all the features required for deployment, and the demonstration of ARPKI’s efficient operation.

The full implementation, formal model and security properties, and the analysis tools are available [8] and this work substantially extends the conference publication [9].

**Scope.** Details on the economic aspects of the CA ecosystem (business models, business relationships, commercial strategies, and operational issues) are out of scope for this paper, although, we do discuss some aspects of the digital certificates market in Section 7.2. Also out of scope are PKI governing aspects such as formal and legal procedures involved when adding or removing a CA, and interactions between browser vendors and CAs.

**Organization.** In Section 2 we motivate the properties that PKI architectures should have and in Section 3 we review the state-of-the-art for PKIs. We present 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 and discuss additional practical concerns in Section 7 before drawing conclusions in Section 8.

## 2 DESIRED PROPERTIES

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

### 2.1 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. We therefore assume that the adversary can eavesdrop, modify, and insert messages.

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.

### 2.2 Security Properties

In general, PKIs should provide security, availability, and be efficient when clients authenticate domains. Moreover, 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 accept or 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. Note that some PKIs, e.g., X.509, do not have a notion of certificate registration.
- **Legitimate certificate updates.** The infrastructure should invalidate a domain’s certificate and replace it by a new one 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, thereby allowing it to be detected.

### 2.3 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.

## 3 RELATED WORK

Numerous proposals have been addressing the security and trust issues in the standard X.509 PKIs. As shown in Figure 1, existing approaches can be classified as client-, CA-, or domain-centric. We first survey this landscape and then focus on AKI, which is closest to our work.

### 3.1 Alternative Approaches

**Client-centric approaches.** Proposals in this class empower clients to select dedicated entities to evaluate a certificate’s correctness before accepting it. *Policy engine* [10] analysis supports clients in defining local policies (e.g., cryptographic requirements, consistency of certificates based on an observed history, etc.) for trust decisions.

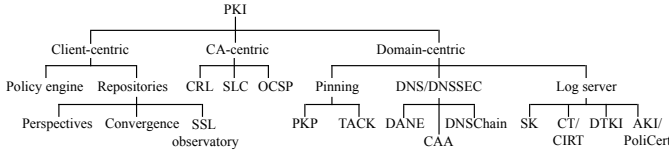


Fig. 1. Classification of PKI proposals.

Several proposals create public repositories that continuously observe domain certificates and enable clients to compare the domain’s key with the version stored in the repositories. This approach requires using an integrity-protected connection between repositories and browsers. *Perspectives* [11] and *Convergence* [12] allow specific requests for each visited domain. While *Perspectives* repositories learn to which domains the clients are connecting, *Convergence* provides anonymity in this regard. The EFF’s *SSL Observatory* [13] also collects global certificate information. However, the information is infrequently updated and no online queries are supported.

Client-centric approaches require the clients to make additional connections to query the repositories. This modification increases latency when establishing an HTTPS connection. On the positive side, servers do not need to be modified or to be aware of clients’ extra checks. Moreover, these approaches can be used to increase confidence in self-signed certificates.

**CA-centric approaches.** The PKI standards for X.509 include *Certificate Revocation Lists (CRL)* [14] 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 online; otherwise a window of opportunity for attacks stays open even after revocation. The CRLs themselves are also quite large. To resolve this burden on network and client resources, *Online Certificate Status Protocol (OCSP)* [15] allows clients to check domains’ certificate status by querying CAs’ OCSP servers, and enables real-time status checks. However, OCSP has security, privacy, and performance concerns. *Short-lived certificate (SLC)* [16] enables domains to acquire certificates with short validity lifetimes and to update the certificates daily. SLC provides security benefits similar to OCSP without the online validation process. The major drawback with CA-centric approaches is their heavy reliance on browser vendors to detect and to blacklist certificates issued by compromised CAs.

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

Pinning-based approaches, such as *Public Key Pinning (PKP)* [17,18] and *Trust Assertions for Certificate Keys (TACK)* [19], allow a domain to declare the valid keys for that domain such that clients “pin” the keys. However, these approaches have security vulnerabilities, such as no protection on the first visit to domains.

The DNSSEC-based proposal called *DNS-based Authentication of Named Entities (DANE)* [20] 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. Similarly, *DNS Certificate Authority Authentication (CAA)* [21] enables domain owners to assert acceptable or unacceptable CAs for issuing their domain’s certificates, and *DNSChain* [22] combines DNS with Namecoin (a decentralized key/value registration and transfer system based on Bitcoin technology) to authenticate domain names. However, the security of DANE, CAA, and DNSChain relies heavily on the security of DNS servers.

Log server-based approaches allow domain owners to record their certificates on public log servers, creating accountability for CAs’ actions. For example, *Sovereign Key (SK)* [6] requires 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. After registering a sovereign key, any certificate for a domain must be signed by the associated sovereign key. Consequently, CAs need the key holder to create certificates, increasing the difficulty to create fraudulent certificates. Unfortunately SK requires clients to query servers, increasing latency and sacrificing privacy.

*Certificate Transparency (CT)* [5] proposes that each domain owner registers the CA-issued certificate to an append-only log with a Merkle hash tree structure maintained by log servers. 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 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 [23]. 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.

*Distributed Transparent Key Infrastructure (DTKI)* [25] combines techniques in SK, CT, and Accountable Key Infrastructure (AKI) discussed in Section 3.2, to manage certificates without trusted validators. Similar to SK, domains in DTKI maintain a master signing key to validate/ revoke the domain’s certificate, and similar to CT and AKI, domains register the CA-signed and master-key-signed TLS certificate to public logs. Consequently, clients only accept a certificate that has been issued by a CA and validated by the domain owner, and that is currently in the log. DTKI removes the reliance on trusted parties by requiring clients to verify the integrity of the log. Unfortunately, DTKI relies heavily on gossiping protocols to synchronize the log status from other clients and does not provide details on how the log misbehavior evidence is disseminated. Furthermore, DTKI requires the use of a single global root, called a *mapping log maintainer*, that everyone must trust. DTKI also increases latency for connections, since clients must contact log servers before every connection. Hence, the log servers know with which domains the clients communicate, violating clients’ privacy.

The main objective of *PoliCert* [26] is to give the domains a way to describe their own certificates and properties of TLS connections. PoliCert relies on a log-based scheme

(like AKI) to publish, manage, and enforce such policies. It also includes a revocation system and a new certificate validation model. However, in this approach (as well as in the previously mentioned ones), the mechanisms for detecting and disseminating log misbehavior are unspecified.

### 3.2 Accountable Key Infrastructure

We review the Accountable Key Infrastructure (AKI) [7] in detail for two reasons: (1) ARPKI is inspired by AKI’s design and employs some of its concepts; (2) ARPKI addresses several shortcomings that we identified in AKI. However, throughout this paper we focus on explaining ARPKI and refer to the conference paper [9] for more details on the differences of the two systems.

AKI proposes to protect domains and clients from vulnerabilities caused by single points of failure, such as a CA’s root key compromise [1]–[4]. Through checks-and-balances among independent entities, AKI distributes trust over multiple parties and detects misbehaving entities while efficiently handling certificate operations.

AKI operates with the following three entities:

1. A **Certification Authority** 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, e.g., the (dis)appearance of certificates.

With these entities, the owner of a domain  $A$  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; and
- $CA\_MIN$ : Minimum number of CA signatures needed to initially register and update the 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,  $A$  uses 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

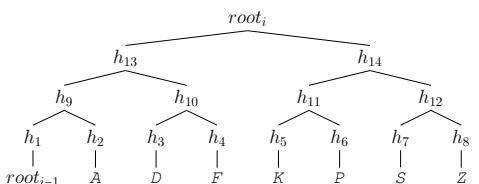


Fig. 2. Integrity Tree in  $i$ -th  $ILS\_UP$  period. Leaves represent domain names in a lexicographic order.

leaves are lexicographically sorted by domain 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. At each  $ILS\_UP$  period, the ILS updates the tree by (1) adding new entries, (2) replacing updated entries, (3) deleting revoked and expired entries, and (4) computing the new root hash for the tree’s current version.

This structure enables the ILS to create efficient proofs about its own content, including presence and absence proofs. To prove that an AKICert exists for  $A$ , the ILS provides  $h_1, AKICert_A, h_{10}$ , and  $h_{14}$ . To prove that  $E$  does not have any registered AKICerts, the ILS provides the presence proofs for the immediate neighbors  $D$  and  $F$ :  $h_9, AKICert_D, AKICert_F$ , and  $h_{14}$ . Along with presence and absence proofs, the ILS provides a signature on the combination of the current root hash  $root_i$  and the timestamp  $T_i$ , 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, delays will occur during the certificate registration and validation processes, contradicting AKI’s claimed efficiency.

Second, AKI’s security properties have not been proven in detail, and additional validation would help gain assurance with respect to AKI’s security claims. In particular, with no mature implementation, certain edge cases are likely to have been missed which impact the security claims. For instance, without synchronizing ILSes, an adversary can register malicious AKICerts, and to attribute misbehavior, all ILSes and CAs must include the triggering requests (which are signed by the sender) for any action they perform.

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.

### 3.3 Comparison

In Table 1 we compare different log-based public-key validation infrastructures based on security, availability, deployability, and usability metrics.

For security we analyze how long an attacker must successfully impersonate a sufficient number of trusted parties to launch an attack, and for how long this attack persists even after trusted parties have recovered. For example, when a trusted CA issues a bogus domain certificate after the CA’s key becomes compromised, such an action has no effect in SK since SK requires the compromised key to be cross-validated by the domain’s sovereign key. However, once CT logs a bogus certificate, it stays valid until the certificate expires, which may take up to a few years. For all other schemes, the logged bogus certificate remains valid until the log servers invalidate it.

When a trusted log server’s key becomes compromised, SK requires several days until the client learns the updated

TABLE 1

Comparison of different log-based public-key validation infrastructures based on security, availability, and efficiency metrics. Entries in red indicate major disadvantages of the corresponding scheme.

	SK	CT	CIRT	DTKI	AKI	ARPKI
<b>Security</b>						
MitM attack mitigation	Y	N	Y	Y	Y	Y
Duration of trusted CA key compromise	0	months	LUP	LUP	LUP	LUP
Duration of untrusted CA key compromise	0	months	LUP	LUP	0	0
Duration of trusted log server key compromise	days	days	0	0	0	0
Duration of domain key(s) compromise	mins-hours	$\infty$	LUP	LUP	LUP	LUP
Multiple valid certificates supported for a domain	Y	Y	N	Y	N	N
Built-in revocation mechanism	Y	N	Y	Y	Y	Y
Proof of domain certificate absence	N	N	Y	Y	Y	Y
Security correctness guaranteed by formal proof	N	N	Y	Y	N	Y
Number of parties that must be compromised for successful attack out of total external parties that the scheme requires	1/1	1/1	1/1	1/1	2/3	n/n
<b>Availability</b>						
Initial registration duration of unavailability (DoU)	0	0	LUP	LUP	0	0
Certificate update DoU	0	0	LUP	LUP	0	0
Recovery from key(s) loss DoU	0	0	LUP	LUP	HT	HT
CA compromise DoU	0	0	0	0	0	0
Log server compromise DoU	days	days	days	days	0	0
<b>Deployability</b>						
Domain-side changes required	Y	N	N	N	Y	Y
CA-side changes required	N	Y	N	N	N	Y
Extra communication required for clients	Y	N	Y	Y	N	N
Additional bandwidth requirement for TLS setup	kbytes	bytes	-	-	kbytes	kbytes
Synchronization between parties required	Y	N	N	N	Y	Y
<b>Usability</b>						
Privacy-preserving connection	N	Y	N	N	Y	Y
Flexibility to meet domain's security concern	N	N	N	N	Y	Y
Additional end-user action required	Y	Y	N	N	N	N

**Legend for table values:**

'LUP' denotes log server's update period

'HT' denotes hold time set by the domain owner to inactivate the certificate

'n' is a system parameter

key through a software update, resulting in a window of vulnerability; for other proposals, however, compromising only the log server itself is insufficient to launch an attack. When the domain key becomes compromised, SK may require in the range of hours to days until the compromised key is revoked. CT logs do not, however, support any means to prevent attackers from using stale information in the absence of Revocation Transparency, which has not yet been fully specified. All other schemes support revoking the compromised entry after some given update period.

We analyze the scale of compromise required for attacks. While compromising a log server's key suffices to launch an attack in SK, CT, CIRT, and DTKI, AKI requires compromising both a trusted log server and a validator (but no trusted CAs) to launch an attack. ARPKI requires that all trusted log servers and CAs are compromised for a successful attack.

We also compare availability, measured by the Duration of Unavailability (DoU), which is the period during which clients cannot establish a secure connection with a domain after the occurrence of some event. We investigate availability with respect to the following certificate management operations: *a*) Initial registration of a certificate (some schemes are designed to serve the domains or CAs immediately, without waiting for the next log update period or other event); *b*) certificate update (as a standard operation, this operation should not introduce any unavailability period); *c*) recovery from key loss (although such an event should be infrequent, a scheme should support a recovery mechanism). We further investigate DoU in terms of CA and log compromise. While CA compromise does not prohibit clients from establishing a secure connection with a domain for any log-based schemes, log server compromise results in days of DoU for SK, CT,

CIRT, and DTKI until a new software version is pushed out with a new key. In contrast, AKI and ARPKI support the registration of the log server's new key without delay.

We also consider the deployability of the different schemes. First we indicate whether the domains or the CAs need to change their configurations or operations to deploy a given scheme. For example, SK requires domains to generate SK key pairs, and AKI and ARPKI require domains to contact multiple trusted CAs and append their certificates. For the changes required on the CA side, CT requires CAs to append the log proof (called Signed Certificate Timestamp) to the logged certificate. On the other hand, ARPKI requires (selected) CAs to: *a*) relay communications on behalf of the domain owners, and *b*) verify correctness of the certificate registration and update processes. The table further indicates whether the scheme requires additional communication on the client side. Such communication is usually a request to a log, which allows one to deploy a scheme without reconfiguring the domain servers. This communication, however, introduces latency for every TLS connection and it exposes clients to blocking attacks, where a response from a server blocks connection setup. We also investigate the schemes in terms of the additional bandwidth required for a TLS connection setup. Finally we check whether the schemes, during their deployment, require a synchronization or consensus protocol for frequent operations such as certificate update.

The last aspect we investigate is usability. Regarding privacy, any scheme that requires a client-log connection for every connection violates privacy as the log server learns which domains clients are contacting. We also consider the flexibility (i.e., trust agility) to meet the domain's security

needs. For example, AKI and ARPKI provide support for domains to specify the trusted CAs as well as the number of entities required to satisfy the domains' security needs. Finally, we indicate whether end-users are required to perform additional actions after each TLS connection. More specifically, SK requires end-users to review results from timeline servers for the consistent observation of sovereign keys, and CT assigns special roles on clients to monitor and validate the log behavior and exchange observed log tree information to maintain a consistent view.

#### 4 ARPKI: ATTACK-RESILIENT PKI

To construct a PKI that can withstand the compromise of several trusted parties and can gracefully handle catastrophic events (such as domain key loss and compromise), we base ARPKI on publicly verifiable logs and extend the AKI system described in Section 3.2. We now present ARPKI in detail, which is 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 the implementation aspects in Section 6.

ARPKI achieves strong security guarantees using  $n$  entities for the certificate operations:  $n - 1$  CAs and one ILS (Integrity Log Server, see Section 3.2 for details, also on CAs and Integrity Trees). In particular, ARPKI's CAs conduct active on-line confirmations and cross-check each other's actions. We design ARPKI to prevent attacks such that even when  $n - 1$  trusted entities are compromised, the security guarantees still hold.

We assume that some data is initially known to all ARPKI participants (the clients, CAs, and ILSes). In particular, the number  $n$  and the parameter `ILS_UP` (the time interval between ILSes signing their log) are system-wide parameters. Furthermore, every participant has a list of all CAs and ILSes with their corresponding public keys.

Due to space limitations, the full specification of ARPKI and its formal verification model are available online at [8]. We start with a high-level summary of the actors and their responsibilities.

- 1) A **domain** with identity  $A$  registers an ARPKI certificate (ARCert) for itself with the ARPKI infrastructure (CAs and ILSes). Afterwards it can use the resulting ARCert to setup secure connections with clients.
- 2) The **CAs** have multiple responsibilities.
  - They check the identity of the domain owner on registration before signing an X.509 certificate.
  - When  $n$  certificates from different CAs are combined into an ARCert, each CA checks the actions of the other CAs.
  - Throughout the lifetime of the ARCert, the CAs are responsible for checking the logs for this ARCert.
  - The CAs check the ILSes' behavior by downloading all accepted requests from the ILSes, checking their correctness, and comparing them to the published Integrity Trees.

This choice of CAs' responsibilities is driven by (1) the unique knowledge that the honest CAs possess about the certificates that they have produced, (2) the possibility of compromising one or more CAs, and (3)

the CAs' business model that incentivizes them to invest in infrastructure that offers increased levels of security.

- 3) The **ILSes** keep a log of all ARCerts registered with them, in a publicly verifiable append-only Integrity Tree, and provide proofs of existence for ARCerts that are then used by CAs and domains. The ILSes synchronize with each other in a publicly accountable manner.

Whereas the CAs have knowledge about the certificates they produced, they do not have a global view of all certificates, even for a single domain. Furthermore, even though the CAs' responsibilities necessarily increase, we aim to minimize the increase in their traffic. This motivates the introduction of ILSes: these parties see all the certificates globally and can therefore offer global consistency checks. Maintaining a consistent global view requires a form of synchronization. Because we consider that ILSes can also be compromised, their actions must also be checked.

To simplify our presentation, we first describe ARPKI's design for  $n = 3$ . We explain its extension to arbitrarily many trusted entities in Section 4.5.

We give an overview of the main message flows in Figure 3. In the protocol descriptions, we denote that entity  $E$  signed message  $M$  by  $\{M\}_{K_E^{-1}}$ . We require that all signatures include timestamps and unique tags such that they cannot be mistaken for one another. To simplify the presentation, we often leave these timestamps and tags implicit. We write  $H(\cdot)$  to denote a cryptographic hash function.

We explain the main ARPKI processes in the following subsections. In Section 4.1 we explain the initial registration process and, in Section 4.2, how the resulting ARCert is used. We describe the processes of confirmation renewal and validation in Section 4.3 and certificate management in Section 4.4. In Section 4.5, we show how to increase security by choosing a larger  $n$ , thereby involving more CAs.

We assume that each participant follows the given protocol. In case of any deviation (e.g., an incorrectly formatted message), the message is discarded and the participant stops this protocol run.

##### 4.1 Initial ARCert Registration Process

To start using ARPKI, a domain must first obtain a valid ARCert. We refer to this as the initial registration. We now consider the owner of domain  $A$  registering her domain. The messages for  $A$ 's initial registration are given in Figure 4, where the message numbers refer to Figure 3.

**ARCert generation (Initialization step).** ARPKI supports trust agility, meaning that the domain owners can select their roots of trust and modify their trust decisions using X.509 extension parameters. The parameters available are:

- `CA_LIST`, i.e., all CAs that the domain trusts,
- `ILS_LIST`, i.e., all the ILSes that the domain trusts, and
- `ILS_TIMEOUT`, which denotes the duration that ILS's confirmations are valid for.

Additionally, the domain specifies hold times that must be applied by the logs during the ARCert update. This happens in two cases: (1) when an update request for a new ARCert is not signed by the previous private key of the domain, (2) and when a new ARCert is signed by CAs outside `CA_LIST`.



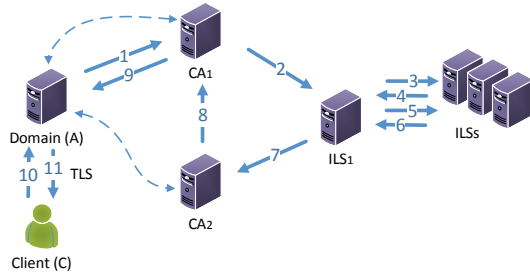


Fig. 3. Basic communication flow of ARPKI. Dashed lines represent the initial X.509 certificate request and delivery. Solid numbered lines represent the message flows to register an ARCert. Lines 10 and 11 represent a TLS connection setup after registration is complete.

A domain owner creates an ARPKI certificate (ARCert) by combining multiple X.509 certificates from trusted CAs. Note that each CA checks the identity of the domain owner to authenticate domains correctly when issuing an X.509 certificate. Each of these X.509 certificates contains the extended information.

**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 which entities she trusts, namely two CAs and one ILS. These are  $CA_1$ ,  $CA_2$ , and  $ILS_1$  in Figure 3.

After each of the initial X.509 certificates has been received from the respective CAs, which have checked the domain owner’s identity, ARPKI requires the domain owner to contact just one CA, here  $CA_1$ . 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,  $ILS_1$ , and  $CA_2$ .

The domain owner also designates  $ILS_1$  to ensure that  $ARCert_A$  is synchronized among all ILSes.  $CA_2$  mainly serves as a monitor and ensures that  $ILS_1$ , as well as other ILSes, operate accordingly, e.g., they add  $ARCert_A$  to their Integrity Trees as promised.

**ILS synchronization (Steps 3–6).** Ideally the same  $ARCert_A$  should be publicly visible among all ILSes. However, synchronizing *all* ILSes may be inefficient, incurring significant time delay, and would be unrealistic in practice. Instead, in ARPKI,  $ILS_1$  takes responsibility on behalf of the domain owner to synchronize  $ARCert_A$  among at least a quorum of all existing ILSes to ensure that at least one non-compromised ILS is part of each quorum.<sup>1</sup> This ensures that only one  $ARCert_A$  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  $ARCert_A$  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

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, which is generalized to an arbitrary  $n$  in Section 4.5.

#### ARCert Generation

- $A$  : Set X.509 extensions
- : Contact trusted CAs, get authenticated
- : Receive signed X.509 certificate
- : Combine multiple certificates into  $ARCert_A$

#### ARCert Registration Request

1.  $A \rightarrow CA_1$  :  $REGREQ = \{ARCert_A, CA_1, ILS_1, CA_2\}_{K_A^{-1}}$
2.  $CA_1$  : Verify signatures in REGREQ
- : Ensure  $CA_1 \in ARCert_A$ ’s  $CA\_LIST$
- : Add  $ARCert_A$  into a pending request list
- $CA_1 \rightarrow ILS_1$  : REGREQ

#### ILS Synchronization

3.  $ILS_1$  : Verify signatures in REGREQ
- : Ensure  $ILS_1 \in ARCert_A$ ’s  $ILS\_LIST$
- : Ensure  $ILS_1, CA_1$ , and  $CA_2$  are different entities
- : Ensure no ARCert was registered for  $A$ ’s domain
- $ILS_1 \rightarrow ILS_n$  :  $SYNREQ = \{REGREQ\}_{K_{ILS_1}^{-1}}$
4.  $ILS_n$  : Verify signatures in REGREQ
- : Ensure no ARCert was registered for  $A$ ’s domain
- $ILS_n \rightarrow ILS_1$  :  $SYNRESP = \{H(REGREQ)\}_{K_{ILS_n}^{-1}}$
5.  $ILS_1$  : Collect SYNRESP from at least a quorum of ILSes
- $ILS_1 \rightarrow ILS_n$  :  $SYNCOMMIT = \{H(REGREQ)\}_{K_{ILS_1}^{-1}}$
6.  $ILS_n \rightarrow ILS_1$  :  $SYNACK = \{H(REGREQ)\}_{K_{ILS_n}^{-1}}$

#### Registration Confirmation

7.  $ILS_1$  : Collect SYNACK from at least a quorum of ILSes
- :  $ACCEPT = \{H(ARCert_A)\}_{K_{ILS_1}^{-1}}$
- $ILS_1 \rightarrow CA_2$  :  $REGRESP = \{ACCEPT, REGREQ, List(SYNACK)\}_{K_{ILS_1}^{-1}}$
8.  $CA_2$  : Verify signatures in REGRESP
- : Ensure  $CA_2 \in ARCert_A$ ’s  $CA\_LIST$
- : Ensure  $ILS_1, CA_1$ , and  $CA_2$  are different entities
- $CA_2 \rightarrow CA_1$  :  $REGCONF = \{\{ACCEPT\}_{K_{CA_2}^{-1}}, List(SYNACK)\}_{K_{CA_2}^{-1}}$
9.  $CA_1$  : Verify signatures in REGCONF
- : Ensure  $ILS_1, CA_1$ , and  $CA_2$  are different entities
- : Remove  $ARCert_A$  from the pending request list
- $CA_1 \rightarrow A$  :  $\{\{ACCEPT\}_{K_{CA_2}^{-1}}\}_{K_{CA_1}^{-1}}$
- $A$  : Ensure  $ILS_1, CA_1$ , and  $CA_2$  are different entities

#### TLS Connection

10.  $C \rightarrow A$  : TLS connection request
11.  $A \rightarrow C$  :  $ARCert_A, \{\{ACCEPT\}_{K_{CA_2}^{-1}}\}_{K_{CA_1}^{-1}}$

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

“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 message.

$CA_2$  now monitors and ensures that  $ILS_1$  actually made the majority of ILSes agree to accept  $ARCert_A$  for their next update time.  $CA_1$  monitors that  $CA_2$  correctly monitors  $ILS_1$ . The next time the ILSes publish a signed log, both CAs check the presence of the expected entries in the log.

## 4.2 Clients Visiting a Domain using ARCert

After completing the initial registration process, the domain  $A$  has a confirmation message (ACCEPT) that is signed by three trusted entities.

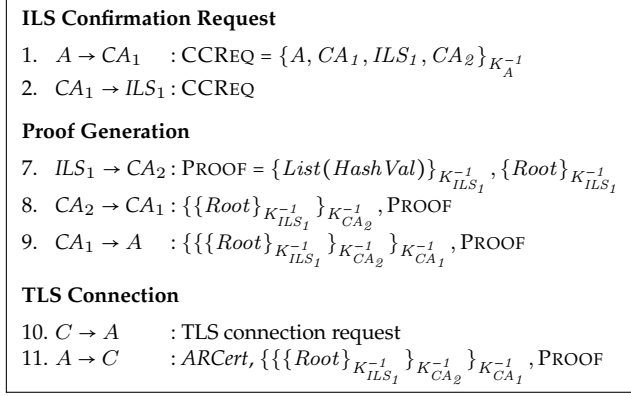


Fig. 5. Message flows for obtaining ARCert's log proof. Upon receiving any signed message, all entities verify the signatures (we omit these steps, focusing on the message flows).

**TLS connection (Steps 10–11).** When the clients connect to domain  $A$ , they receive ACCEPT along with  $ARCert_A$ , which enables them to verify that they are establishing a TLS connection with domain  $A$ . In particular, the 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. The browsers also perform the standard validation [27,28] of every X.509 certificate in the ARCert. If the validation succeeds, the clients accept the TLS connection to the domain  $A$ .

### 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 trusted entities in her  $ARCert_A$ . Unless one or more of these 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 them accountable for their actions.

**TLS connection (Steps 10–11).** Instead of the ACCEPT message, the domain owner now provides the PROOF message together with the Integrity Tree root (signed by all three entities) to TLS connection requests.

The clients can validate an ARCert against a confirmation, be it an ACCEPT message or a PROOF message with a 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.

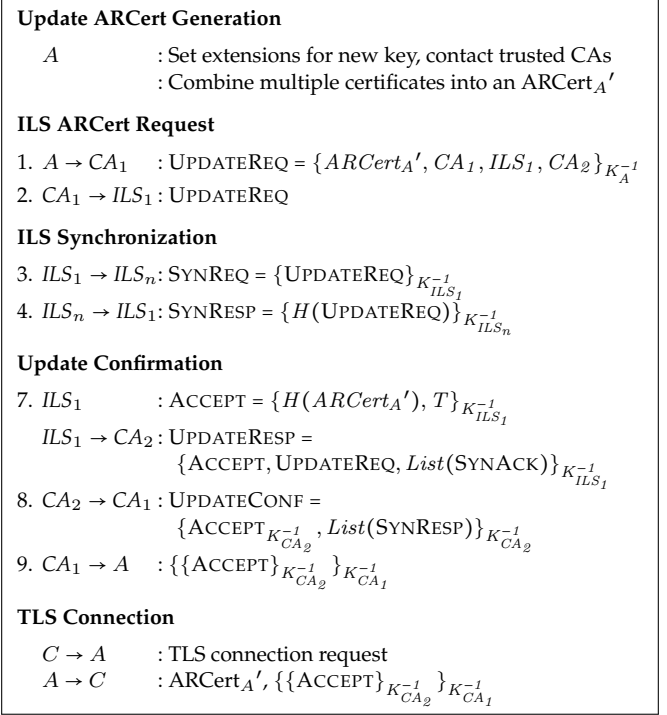


Fig. 6. Message flows for updating domain  $A$ 's ARCert.

## 4.4 Certificate Management

Figure 6 illustrates how ARPKI supports certificate update, which can be used to renew  $ARCert_A$ , or to recover from the loss or compromise of a private key. Note that ARPKI's update procedure differs from certificate revocation known from X.509 PKI. In ARPKI, certificates are revoked by replacing them by new ones.

**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 hold times and the log makes this update visible.

ARPKI's update procedure gracefully handles catastrophic events like (1) recovery from a domain's key compromise or key loss, (2) it can protect an ARCert from being updated by untrusted parties, and (3) it gives the domain owner time to react when an ARCert's update violates the domain's update policy.

**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 a quorum of all existing ILSes agree. This ensures that the world continues to have a consistent view of domain  $A$ 's ARCert.

The mutual checks in Steps 2, 7, and 8 in Figures 4–6 are needed to detect the misbehavior of  $CA_1$ ,  $CA_2$ , and  $ILS_1$



during the initial ARCert registration and subsequent ARCert updates. An attack therefore 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 just described the process for  $n = 3$ , 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 in-between  $CA_2$  and  $CA_1$  receive, check, sign, and send the message to the next CA in line. The resulting system provides better security guarantees as it tolerates  $n - 1$  compromised parties. The downside is that  $n$  entities must be involved in the registration, confirmation, and update operations, which may cause inefficiency in the subsequent client connection.

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. ARPKI would not prevent this attack. However, this attack is very likely to be quickly detected, and all the dishonest ILSes and CAs can be held accountable. This requires use of gossiping protocols to spread information; see the discussion on this in Section 7.

In case of catastrophic failure, for example, when all trusted entities of a domain are compromised, then action outside of the described system must be taken to remove all (or at least some) of those compromised entities.

### 5 MODEL AND ANALYSIS

To establish high assurance guarantees, we formally analyze ARPKI's core security property using the TAMARIN Prover [29,30]. We chose TAMARIN because it is a state-of-the-art protocol 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 construction of the proof tree. When used as an automatic procedure, if the tool terminates without finding a proof, it then returns a counter-example. Counter-examples can be used to refine the model, and give feedback to the implementer and designer.

We now recall some TAMARIN details.

**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  $Fr$  with a fixed semantics, where  $Fr(n)$  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  $l \multimap [a] \multimap r$ , 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 Out fact, which the adversary adds to its knowledge  $K$  fact, and then can send to the protocol participants using the In fact. The adversary can also combine knowledge, using any operator, for example, given  $K(x)$  and  $K(y)$  and a binary operator  $f$ , the adversary deduces  $K(f(x, y))$ .

**Labeled multiset rewriting.** For a given state  $s$ , for each rule  $l \multimap [a] \multimap r$ , if there exists a substitution  $\sigma$  such that  $\sigma(l) \in s$ , then the rule can be triggered, resulting in a new state  $s' = (s \setminus \sigma(l)) \cup \sigma(r)$ . Each time a rule is triggered,  $\sigma(a)$  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 papers on TAMARIN [29,30].

#### 5.1 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 large or repeating 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 when displaying counterexamples. 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 to help guide TAMARIN's heuristics.

#### 5.2 Modeling ARPKI

We modeled the communication flow of ARPKI for the initial ARCert generation and registration following 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.

**Model excerpt.** The full TAMARIN model for ARPKI is available [8] 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, 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 = $ILSk in
    pkA = pk(~ltkA) in
    [ !Ltk($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 throughout this section. Fact symbols with the `!` prefix are never consumed and can be used repeatedly.

We model asymmetric keys by using fresh (unique) terms as long-term private keys (here: `~ltkA`) and use an abstract one-way function `pk` to yield 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, the 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 `ILStoAdd` 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 remaining message flows are modeled analogously.

### 5.3 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 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 and 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.

## 5.4 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 differs 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 compromise) 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 ( $cid, a, b, \dots$ ): for all values of those variables there should be a  $\text{GEN\_LTK}(\dots)$  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 2. As we will show next, it holds for ARPKI whenever at most two entities are compromised, and can be broken only with three or more compromised entities. This is a stronger property than AKI offers, since with AKI only two compromised entities result in a security breach already.

**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 5.1. We estimate the overall verification effort at several person months.

## 5.5 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, helping us to make all details of the protocol design precise and to uncover missing detail. 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 performed by the browser, to protect against a party signing more than once, were not performed by the domain owner during certificate creation. The missing checks were then added to the model, the specification document, and the implementation.

We improved the formal model over several iterations. For example, as our understanding of the implementation concerns and their impact on the design increased, we added more implementation details to the model. Additionally, we continuously reproved several basic properties that served as sanity checks, such as proving that certain expected behaviors can indeed occur. Finally, we sometimes found false attacks on our earlier models that were the result of over-approximating the actual protocol. Such attacks, which would not be possible on the implementation, indicated that the model needed to be further refined. Once the formal model had stabilized, further issues found in failed proof attempts were quickly communicated and fixed.

## 6 IMPLEMENTATION

In this section we describe our proof-of-concept implementation of ARPKI and assess its performance. Our implementation provides all the features required for deployment. In particular, we implemented the following parts: (1) the communication flows and processing logic for the message exchanges presented in Section 4, (2) the ILS process with the fully implemented Integrity Tree, and the capability to publish the information required for its consistency checks, (3) the CA process, which monitors the ILSes and publishes misbehavior, (4) the CA process that can produce on-line confirmations, (5) the client process, i.e., a browser extended with support for full ARCert validation, (6) the protocol for accountable synchronization, and (7) the domain tool, which can register, update, revoke, recover, and confirm ARCert.

### 6.1 Design and Implementation Choices

We implement the ARCert using a concatenation of standard X.509 certificates. We use X.509 extensions [31] to add ARPKI-related fields such as  $\text{CA\_LIST}$ ,  $\text{ILS\_LIST}$ , and  $\text{CA\_MIN}$ .

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 as modules with APIs. The communication module 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 misbehavior. Through the publishing module, ILSes and CAs show their current view of all Integrity Trees.

```

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 an 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' for domain a.
    & i3 < i4 )
    ==>
    ( (not (Ex #j. K(key) @j)) ) ) " // adversary cannot know that private key

```

Fig. 7. Main security property proven

Due to the reasonably small number of ILSes and CAs, the synchronization between parties is realized by unicast communication. Each ILS and CA contacts every ILS at update time to keep the Integrity Trees consistent. The synchronization protocol can tolerate a range of system failures as every ILS and CA acts as global state replication.

We implement the ARPKI-enabled web server by reconfiguring the Nginx HTTP server (version 1.5.7). We configure the web server to periodically interact with ARPKI’s infrastructure to fetch fresh confirmations that are provided to the browsers. 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 client connections.

We implement the client by extending the Chromium web browser and we deploy our system without significant changes to the 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 test-bed that included all entities: the ILS, CAs, 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 processing takes for the infrastructure for three requests initiated by the domain: `REGREQ`, `UPDATEREQ`, and `CCREQ`. `REGREQ` is sent once per domain, while `UPDATEREQ` is envisioned to be sent annually for each domain. The most common request is `CCREQ`, which is sent roughly every `ILS_TIMEOUT`. Measurements are given as the average over 1000 test runs in Table 2. We only measure the total processing time spent by the entities involved, without considering network latency.

To validate ARPKI’s deployability in a more realistic setting we conducted an experiment using a distributed infrastructure based on Amazon’s EC2. We deployed CAs, ILSes, and domains as separate EC2 instances located in

TABLE 2  
Total processing time (in milliseconds) per request per 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

different geographical locations in Asia, Europe, and the US. Then, for a randomly selected domain,  $CA_1$ ,  $CA_2$ , and  $ILS_1$ , we conducted ARCert registration, ARCert update, and ARCert confirmation, and measured the total time consumed by each operation. Measured over 100 such runs, these operations took on average 1.966, 1.930, and 1.921 second, respectively.

For 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 an ARCert, using the standard browser validation procedure. This includes checking whether the certificate is issued for a correct domain, has been signed correctly, has not expired, etc. The ARPKI validation phase additionally checks that (1) certificates within an ARCert are signed by CAs trusted by the domain, (2) proofs have been produced for the correct ARCert and that the proof validates with the correct root, and (3) 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 standard validation took 0.70ms on average, and the ARPKI validation took on average 1.55ms.

The most time-consuming operations in our system involve signature creation and verification. This overhead can be reduced by using state-of-the-art digital signature schemes [32,33]. However, this may not be backward-compatible with software using 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 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 configurable 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, a small amount of latency may be introduced by the transport layer protocol [34]. Note that our solution does not introduce any additional computational overhead for the server during regular HTTPS connections.

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

## 7 DISCUSSION

ARPKI's security parameter  $n$  describes the number of parties that must be compromised for a successful impersonation attack. This parameter expresses a basic trade-off between security and efficiency: by increasing  $n$ , all ARCert become more trustworthy, but system performance decreases and domain expenses increase.

Although we do not address the question of what an optimal value for  $n$  is (although this might be the subject of a future, empirical study), we do discuss below: (1) how ARPKI can be enhanced to detect powerful attacks where  $n$  or more parties are compromised, and (2) what could be the price of an ARCert on the current TLS certificate market.

### 7.1 Detection of a Successful Impersonation Attack

ARPKI provides strong security guarantees, namely that a successful impersonation attack is only possible when at least  $n$  different parties (i.e., at least one ILS and  $n - 1$  CAs) are under control of the adversary. If such a compromise occurs, the adversary can then produce a malicious ARCert along with fake proofs that this ARCert is in the log. Consequently, every client will accept a TLS connection based on this ARCert. Moreover, the adversary does not have to update the log with this ARCert, making the attack undetectable for the non-compromised part of the infrastructure (as only the attacked clients would see proofs for the malicious ARCert). Unfortunately, it is impossible to protect the clients from such a threat. However, one could consider whether it is possible to at least retroactively detect such an attack. ARPKI can be extended to support the detection of such a powerful adversary using the two approaches outlined below.

**Validators.** In ARPKI, logs are publicly available and anyone can easily monitor them. A party that is not part of the ARPKI infrastructure, but constantly monitors the operations of logs, is called a validator (the same name is used in AKI, but in CT this party is called a *monitor*). To monitor the logs, each validator periodically downloads the logs, and checks whether the ILSes operate correctly. For this check, a validator performs exactly the same action as a CA; thus validators can be straightforwardly implemented with support for a subset of CA operations. As proposed in AKI [7], validators

could be operated by non-governmental Internet governance organizations, such as the EFF, which have no incentives to collude with corporations (like CAs) and governments.

For every ARPKI-enabled TLS connection, a client is provided with an ARCert, a presence proof for that ARCert, and the tree root of an ILS (signed by  $n$  parties). This information is required to establish a connection, and it is sufficient to check whether the view of the log presented to the client is consistent with the validators' views. For this, the client can periodically ask validators for their views of the log. Misbehavior is detected when the views are inconsistent, that is, when there are two or more different roots for the same `ILS_UP` period. Such inconsistent roots constitute a proof of misbehavior and can be reported, for example, to software vendors to remove misbehaving entities from their root certificate stores.

**Gossip protocols.** The main disadvantages of a validator-based approach is the requirement for additional infrastructure (as validators should be highly available), and the risk that for highly targeted attacks, one or more validators could still be compromised. To solve the problem of efficient misbehavior detection, without relying on any trusted party, *gossip protocols* have been suggested [35] where clients randomly exchange information about the log. Gossip protocols provide the clients with a lightweight way to guarantee that they have the same view of the log.

The first proposal that uses this approach was presented [36]. The clients exchange signed roots with the server during the TLS connection. This exchange is realized during normal client traffic, whereby messages about the log are piggybacked on top of native requests and responses. The protocol therefore requires neither additional infrastructure nor a dedicated connection. The only requirement is that some fraction of clients and servers gossip. Evaluation of the protocol shows that every log's inconsistency is eventually detected with high probability. Moreover, as in the previous approach, the proof of misbehavior is that different signed roots from the same time period exist. This protocol can be directly applied to ARPKI, making the detection of misbehavior more robust.

### 7.2 Economic Incentives

Although ARPKI requires changes to CAs, we believe that CAs have incentives to participate in the system. The reputation of CAs has been undermined by security breaches, which were challenging to detect and mitigate. Therefore, CAs understand the value of log-based approaches that make observing their own certificates and operations easier and more transparent. Moreover, high-quality CAs would gain a competitive advantage if their reliable operation would become publicly verifiable. Evidence for these points is given by the deployment of log servers in Certificate Transparency, which currently features 13 independent log servers operated for instance by Google, Symantec, DigiCert, and StartCom.

Many companies are interested in a secure Internet since this is likely to increase the Internet's use and therefore their profits; companies like Google and Facebook, which invest heavily in security, likely reason this way. Moreover, some of these high-profile domains have been targeted in previous CA breaches, which has increased the general awareness

of the drawbacks of the current generation of PKIs. Consequently, several companies are actively using and developing log-based security enhancements. For instance, Facebook, by monitoring CT logs, discovered an unauthorized issuance of two certificates on multiple Facebook domains [37].

An ARCert provides stronger assurance for a binding between a public key and a domain name than today's TLS certificates because it consists of multiple (namely,  $n - 1$ ) CA assertions. Such redundancy increases the security of the certificate, but also increases the price.

The market for digital certificates is diverse. Commercial CAs usually issue three types of certificates based on the validation they perform: (1) domain validation (DV), where a CA checks the right of the applicant to use a specific domain name by an automated e-mail, (2) organization validated (OV), where domain validation is performed and the CA conducts some vetting of the organization, and (3) extended validation (EV), where domain validation is performed as well as a thorough vetting of the organization. Arnbak et al. [38] survey the market for TLS certificates and present price ranges for these different types of certificates. According to this study, DV certificates cost \$0–\$249, OV certificates \$38–\$258, and EV certificates \$100–\$1,520.

The above cost estimates show that an ARCert implemented by concatenating today's certificates would not be too expensive. For instance, assuming *the most secure* EV certificates as components of an ARCert, and  $n = 4$ , the lowest price of an ARCert would be \$300. That is less than half the average price for a single EV certificate, which is around \$600 in 2014 [38].

### 7.3 Deployment Aspects

Google has already proposed, standardized, and deployed CT, a log-based approach [35]. Since March 2016, the Chrome Browser ceased displaying the green bar for EV certificates that are not registered in a log server. ARPKI's ILSeS could be deployed on top of the CT infrastructure, given their similarities.

While CT is optimized for deployability, ARPKI is optimized for security. Consequently, the deployment of ARPKI is more challenging. However, with the increased interest in secure network architectures, we have the opportunity to add a secure PKI ecosystem. Specifically, ARPKI is a component of the SCION secure Internet architecture [39]. SCION is already deployed by several ISPs and we have observed that there is substantial interest by security-conscious corporations in achieving a higher level of security for their certificates with ARPKI.

## 8 CONCLUSIONS

We have presented ARPKI, a new PKI with strong security guarantees. It offers resilience against impersonation attacks that involve  $n - 1$  compromised trusted entities. Moreover, if all entities involved in an ARCert are compromised, in which case domain impersonation cannot be prevented, the other CAs 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 CAs out of band, involving validators, or participating in a gossip protocol. Even though attack resilience cannot be achieved in this case, complete compromise situations are at least visible. We have implemented and evaluated our proposal, providing evidence that a TLS connection setup using ARPKI for certificate validation incurs only a small overhead.

Throughout the design and implementation of ARPKI, we used formal analysis to validate our design decisions. 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, 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 in part supported by CyLab at Carnegie Mellon University, by NSF under award CNS-1040801, by a gift from Google, and by ETH Zurich. The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement 617605. We thank Emilia Kasper for her feedback during the early stages of this work. We thank Lorenzo Baesso and Lin Chen for their programming assistance.

## REFERENCES

- [1] P. 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.
- [2] P. Roberts, "Phony SSL certificates issued for Google, Yahoo, Skype, others," <http://threatpost.com/phony-ssl-certificates-issued-google-yahoo-skype-others-032311/>, March 2011.
- [3] J. Menn, "Key internet operator VeriSign hit by hackers," <http://www.reuters.com/article/2012/02/02/us-hacking-verisign-idUSTRE8110Z820120202>, January 2012.
- [4] T. Sterling, "Second firm warns of concern after Dutch hack," <http://news.yahoo.com/second-firm-warns-concern-dutch-hack-215940770.html>, September 2011.
- [5] B. Laurie, A. Langley, and E. Kasper, "Certificate Transparency," <http://tools.ietf.org/pdf/rfc6962.pdf>, June 2013, IETF RFC 6962.
- [6] P. Eckersley, "Sovereign Key Cryptography for Internet Domains," <https://git.ietf.org/?p=sovereign-keys.git;a=blob;f=sovereign-key-design.txt;hb=HEAD>, 2012.
- [7] T. H.-J. Kim, L.-S. Huang, A. Perrig, C. Jackson, and V. Gligor, "Accountable Key Infrastructure (AKI): A Proposal for a Public-Key Validation Infrastructure," May 2013.
- [8] "ARPKI: Full implementation, formal model, and security properties," 2015, <http://www.netsec.ethz.ch/research/arпки>.



- [9] D. A. Basin, C. J. F. Cremers, T. H. Kim, A. Perrig, R. Sasse, and P. Szalachowski, "ARPKI: attack resilient public-key infrastructure," in *Proc. of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, G. Ahn, M. Yung, and N. Li, Eds. ACM, 2014, pp. 382–393. [Online]. Available: <http://doi.acm.org/10.1145/2660267.2660298>
- [10] M. Abadi, A. Birrell, I. Mironov, T. Wobber, and Y. Xie, "Global authentication in an untrustworthy world," in *HotOS*, P. Maniatis, Ed. USENIX Association, 2013.
- [11] D. Wendlandt, D. G. Andersen, and A. Perrig, "Perspectives: Improving SSH-style Host Authentication with Multi-Path Probing," Jun. 2008.
- [12] "Convergence," <http://convergence.io/>, 2011.
- [13] Electronic Frontier Foundation, "SSL Observatory," <https://www.eff.org/observatory>.
- [14] 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, Apr. 2002.
- [15] 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, United States, Jun. 1999.
- [16] E. Topalovic, B. Saeta, L.-S. Huang, C. Jackson, and D. Boneh, "Towards Short-Lived Certificates," May 2012.
- [17] "Public Key Pinning," <http://www.imperialviolet.org/2011/05/04/pinning.html>, May 2011.
- [18] "Public Key Pinning Extension for HTTP," <http://tools.ietf.org/html/draft-ietf-websec-key-pinning-01>, Dec. 2011.
- [19] M. Marlinspike and T. Perrin, "Trust Assertions for Certificate Keys," <http://tack.io/draft.html>, May 2012.
- [20] P. Hoffman and J. Schlyter, "The DNS-based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA," <http://tools.ietf.org/html/rfc6698>, 2012, IETF RFC 6698.
- [21] P. Hallam-Baker, R. Stradling, and B. Laurie, "DNS Certificate Authority Authentication (CAA) Resource Record," <https://tools.ietf.org/html/rfc6844>, 2013, IETF RFC 6844.
- [22] G. Slepak, "DNSChain," <https://github.com/okTurtles/dnschain>, 2014.
- [23] B. Laurie and E. Kasper, "Revocation Transparency," <http://sump2.links.org/files/RevocationTransparency.pdf>, 2012.
- [24] M. D. Ryan, "Enhanced certificate transparency and end-to-end encrypted mail," 2014.
- [25] J. Yu, V. Cheval, and M. Ryan, "DTKI: a new formalized PKI with no trusted parties," *IACR Cryptology ePrint Archive*, 2014. [Online]. Available: <http://eprint.iacr.org/2014/600>
- [26] P. Szalachowski, S. Matsumoto, and A. Perrig, "PoliCert: Secure and Flexible TLS Certificate Management," Nov. 2014.
- [27] R. Biddle, P. C. van Oorschot, A. S. Patrick, J. Sobey, and T. Whalen, "Browser interfaces and extended validation SSL certificates: an empirical study," in *Proc. of the 2009 ACM workshop on Cloud computing security*. ACM, 2009, pp. 19–30.
- [28] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The most dangerous code in the world: validating SSL certificates in non-browser software," in *Proc. of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 38–49.
- [29] B. Schmidt, S. Meier, C. Cremers, and D. Basin, "Automated analysis of Diffie-Hellman protocols and advanced security properties," in *Computer Security Foundations Symposium (CSF)*. IEEE, 2012, pp. 78–94.
- [30] S. Meier, B. Schmidt, C. Cremers, and D. Basin, "The TAMARIN Prover for the Symbolic Analysis of Security Protocols," in *Computer Aided Verification, 25th International Conference, CAV 2013, Princeton, USA, Proc., ser. LNCS*, vol. 8044. Springer, 2013, pp. 696–701.
- [31] 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), Internet Engineering Task Force, May 2008. [Online]. Available: <http://www.ietf.org/rfc/rfc5280.txt>
- [32] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang, "High-speed high-security signatures," *Journal of Cryptographic Engineering*, vol. 2, no. 2, pp. 77–89, 2012.
- [33] E. Kasper, "Fast elliptic curve cryptography in OpenSSL," in *Financial Cryptography and Data Security*, ser. LNCS. Springer, 2012, vol. 7126, pp. 27–39.
- [34] A. Langley, "Overclocking SSL," <https://www.imperialviolet.org/2010/06/25/overclocking-ssl.html>, Jun. 2010.
- [35] B. Laurie, A. Langley, and E. Kasper, "Certificate Transparency," <http://tools.ietf.org/html/draft-laurie-pki-sunlight-07>, Jun. 2013.
- [36] L. Chuat, P. Szalachowski, A. Perrig, B. Laurie, and E. Messeri, "Efficient gossip protocols for verifying the consistency of certificate logs," in *Communications and Network Security (CNS)*. IEEE, 2015.
- [37] D. Huang and B. Hill, "Early Impacts of Certificate Transparency," <https://www.facebook.com/notes/protect-the-graph/early-impacts-of-certificate-transparency/1709731569266987/>, 2016.
- [38] A. Arnbak, H. Asghari, M. Van Eeten, and N. Van Eijk, "Security collapse in the https market," *Queue*, vol. 12, no. 8, Aug. 2014.
- [39] D. Barrera, R. M. Reischuk, P. Szalachowski, and A. Perrig, "Scion five years later: Revisiting scalability, control, and isolation on next-generation networks," *arXiv e-prints*, Aug. 2015. [Online]. Available: <http://arxiv.org/pdf/1508.01651>

## AUTHOR BIOGRAPHIES

David Basin is a full professor at ETH Zurich. He received his Ph.D. from Cornell University in 1989. His research focuses on Information Security, in particular on foundations, methods, and tools for modeling, building, and validating secure and reliable systems.

Cas Cremers is a full professor at the University of Oxford. He received his Ph.D. from Eindhoven University of Technology in 2006. His research focuses on the foundations and analysis of secure systems, using approaches from formal methods, automated verification, and cryptographic analysis.

Tiffany Hyun-Jin Kim is a research scientist at HRL Laboratories LLC. She received her Ph.D. from Carnegie Mellon University in 2012, and her research interests include usable security and privacy, network security, and applied cryptography.

Adrian Perrig is a Professor at the Department of Computer Science at ETH Zurich. He is also a Distinguished Fellow at CyLab, and an Adjunct Professor of Electrical and Computer Engineering, and Engineering and Public Policy at Carnegie Mellon University. He received his Ph.D. from Carnegie Mellon University in 2002. His research revolves around building secure systems – in particular his group working on the SCION secure future Internet architecture.

Ralf Sasse is a senior scientist at ETH Zurich. He received his Ph.D. from the University of Illinois in 2012. His research focuses on security of software, particularly applying Formal Methods to security protocol verification as a building block.

Pawel Szalachowski is a postdoctoral researcher at ETH Zurich. He received his Ph.D. degree in Computer Science from Warsaw University of Technology, Poland, in 2012. His research interests include the Internet and network security, public-key infrastructures, and applied cryptography.