PoliCert: Secure and Flexible TLS Certificate Management

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

Stephanos Matsumoto
CMU, United States
and
Inst. of Information Security
Dept. of Computer Science
ETH Zurich, Switzerland
smatsumoto@cmu.edu

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

ABSTRACT

The recently proposed concept of *publicly verifiable logs* is a promising approach for mitigating security issues and threats of the current Public-Key Infrastructure (PKI). Although much progress has been made towards a more secure infrastructure, the currently proposed approaches still suffer from security vulnerabilities, inefficiency, or incremental deployment challenges.

In this paper we propose PoliCert, a comprehensive log-based and domain-oriented architecture that enhances the security of PKI by offering: *a)* stronger authentication of a domain's public keys, *b)* comprehensive and clean mechanisms for certificate management, and *c)* an incentivised incremental deployment plan. Surprisingly, our approach has proved fruitful in addressing other seemingly unrelated problems such as TLS-related error handling and client/server misconfiguration.

Categories and Subject Descriptors

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

General Terms

Security

Keywords

Public-Key Infrastructure; SSL; TLS; certificate validation; security policy; public-key certificate; public log servers

1. INTRODUCTION

Certificate authorities (CAs) in today's TLS PKIs are endowed with great authority. As trusted parties, they sign certificates used by clients all over the world to authenticate servers and establish HTTPS connections. Browser and operating system vendors also hold significant power in the TLS infrastructure, since they manage the list of trusted root CA certificates which serve as the roots of trust for certifying TLS certificates.

Though these parties wield significant authority in the TLS ecosystem, their trustworthiness has been tarnished by several recent events. Operational mistakes, social engineering attacks, and governmental compulsion [28] have resulted in the issuance of fraudulent TLS certificates for many high-profile sites. In these cases, adversaries can impersonate domains to clients by performing active man-in-the-middle (MitM) attacks, intercepting secure connections and stealing potentially sensitive information.

In an effort to address this problem, proposals such as Certificate Transparency [24] have sought to increase CA accountability by using *public logs* to monitor CA behavior. Like CAs, logs are trusted parties that contribute towards certifying the validity of a certificate. Public logs record certificates issued by CAs and provide proofs that a certificate has been observed, thus making all certificates (even maliciously-issued ones) visible for public scrutiny.

However, there is a disconcerting imbalance of power in the TLS infrastructure. Domain owners have little control over how their own TLS certificates are used and verified. In log-based proposals, a valid CA signature and a proof from a public log are enough to convince a client that a certificate is authentic. Domains cannot specify any criteria of their own that their certificates must fulfill. In fact, browser vendors often set the majority of such criteria, by specifying root CAs whose keys anchor certificate chains and by determining how errors in the TLS handshake are handled (such as through a soft failure, which allows the user to proceed anyway, or a hard failure, which does not).

Additionally, CT and other log-based proposals [4, 20, 27] suffer from several inefficiencies. While proofs from public logs can be efficiently generated and validated, some of these proposals assume or explicitly argue for the global coordination of logs in order to ensure that they have a consistent view of valid certificates. This global coordination is inefficient because certificate issuances and revocations are frequent, and because it requires that each domain only have a single active certificate at a time. Some of these proposals also do not efficiently handle events such as certificate revocation, key loss, or key compromise.

To address these inefficiencies and, in particular, the imbalance of control in today's TLS PKI, we design and propose PoliCert, a log-based proposal that allows domains to define policies governing the usage of their TLS certificates. These policies, called *subject certificate policies (SCPs)*, provide domains with much greater control over their certificates by allowing them to specify parameters such as trusted CAs, update criteria, and error handling that their certificates must follow, and also gracefully handle the loss or compromise of a private key. Additionally, parameters in SCPs are inheritable and can be applied to subdomains, allowing more ex-

CCS'14, November 3–7, 2014, Scottsdale, Arizona, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2957-6/14/11 ...$15.00.
http://dx.doi.org/10.1145/2660267.2660355.

pressive certificate policies and providing some resilience against misconfiguration in subdomains.

We also propose a design for *multi-signature certificates (MSCs)*, which allow multiple CA signatures on a certificate and serve as the format for encoding SCPs. This feature is not part of the X.509 standard, but provides stronger authentication of a subject's public key and enhances resilience to CA compromise. Since MSCs consist of a set of X.509 certificates and extensions, our design requires very little change to the existing certificate standard.

To enable efficient monitoring, updates, and revocation of certificates and policies, we propose a system of public logs which track both MSCs and SCPs. These logs leverage Merkle Hash Trees (MHTs) to store and generate cryptographic proofs for certificates, revocations, and policies. Proofs in PoliCert can also prove the *absence* of MSCs and SCPs from the log in order to prevent adversaries from suppressing this information and forging fake certificates or policies. This property also secures an incremental deployment of PoliCert.

Logs maintain separate databases for MSCs and SCPs. This separation allows us to mitigate inefficiencies in previous proposals. Because a valid and logged MSC that meets the criteria specified in the SCP is considered authentic, logs only need to maintain a globally consistent view of domains' SCPs. This means that domains can only have a single active *policy* at any given time (which is reasonable), but can have as many active *certificates* as they wish.

To summarize, this paper makes the following contributions:

- An argument for the use and benefits of subject certificate policies (§4.1).
- A proposed format for multi-signature certificates, which allow an arbitrary number of independent CA signatures on a certificate (§5.1).
- A design of subject certificate policies, which specify how and which certificates of a domain or subdomain can be used (§5.2).
- Protocols for verifying the authenticity and checking the revocation status of an MSC using publicly-auditable logs (§6).
- A full implementation, and evaluation of the security and efficiency of our protocols (§7, §9).

2. PROBLEM DEFINITION

Our main objective in designing PoliCert is to explore the effects of enabling domains to define their own certificate compliance policies while making minimal changes to the current PKI environment. In doing so, we endeavor to provide a mechanism for enforcing expressive and extensible policies governing the usage of TLS certificates. We aim to allow domains to specify criteria such as CAs authorised to sign certificates for a domain, parameters for updating certificates and policies, and error handling behavior in TLS handshakes. Furthermore, we want to efficiently handle events such as certificate revocations, policy updates, and loss or compromise of a private key.

Our adversary's goal in this setting is to impersonate a website to a client in order to perform a MitM attack. To this end, the adversary may be able to compromise trusted parties by gaining access to their private keys and signing messages using these keys. However, this access may be short-term, such as the ability to sign a single message or certificate, and we assume that the adversary is not able to gain long-term access to a threshold number of trusted parties. We further assume that the adversary cannot mount other

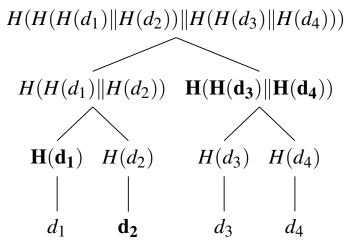


Figure 1: Merkle hash tree with bold nodes proving the existence of d_2 in the tree. The symbol $\|$ denotes concatenation and $H(\cdot)$ is cryptographic hash function.

attacks (such as social engineering attacks) on clients, and cannot break cryptographic primitives such as public-key algorithms or hash functions.

In this paper we assume that browsers are kept up-to-date, that their lists of root CA and log certificates are authentic, and that the various browser vendors reach a consensus of the security level of various cryptographic algorithms. We also assume that public logs and auditors (§4.2) are highly available. Finally, we assume that all parties are time synchronized within a few minutes.

The main properties we want to achieve are:

1. **Resilience** to compromise of trusted parties. Unless more than a threshold number of all public logs are compromised, it should be impossible for an adversary to impersonate a domain by forging a certificate or policy that would be accepted by clients.
2. **Efficiency** of certificate update and revocation. It should be possible to update or revoke a domain's certificate within hours of the domain requesting the action.
3. **Balanced control** among CAs, logs, browsers, and domains. All parties should be able to contribute towards determining whether or not a domain's certificate is valid, whether through signing information or specifying parameters for connection establishment.
4. **Interoperability** with the current CA business model. To facilitate deployment of PoliCert, CAs should not have to change their business model significantly from their current operations.

3. BACKGROUND AND RELATED WORK

3.1 MHTs and Presence/Absence Proofs

A *Merkle hash tree (MHT)* is a binary tree in which each leaf node contains data, and each non-leaf node contains the hashes of its child nodes [25]. The structure of the tree can be leveraged to efficiently prove that a leaf node (or its data) is in the tree. Since as shown in Figure 1, only one node per level of the tree is needed in a proof of presence, the proof size for any leaf node is proportional to the tree height, which is $O(\log_2 n)$ for n leaf nodes.

An entity wishing to verify the presence of d_2 in the tree shown in Figure 1 would receive the set $\{d_2, H(d_1), H(H(d_3)\|H(d_4))\}$, which forms the *proof of presence*. The entity then hashes d_2, which with $H(d_1)$ allows it to compute $H(H(d_1)\|H(d_2))$. This value can then be concatenated with $H(H(d_3)\|H(d_4))$ and hashed to obtain the *root hash*, the value at the root node of the tree. By comparing the computed root hash with the true root hash of the tree, a party can check whether or not d_2 is in the tree.

If leaf nodes are ordered in some way (such as lexicographic ordering for strings), the tree can also provide *proofs of absence*.

For example, if the leaf nodes in Figure 1 are domain names $d_1 =$ a.com, $d_2 =$ c.com, $d_3 =$ d.com, and $d_4 =$ e.com, then the set of nodes $\{d_1, d_2, H(H(d_3) \| H(d_4))\}$ forms a proof of absence for b.com. This is because d_1 and d_2 are adjacent nodes between which b.com would be located if it were in the tree.

If leaf nodes are ordered chronologically for when they were inserted into the tree, then the tree can provide proofs of *consistency* showing that leaf nodes have only been added to the tree. Such a proof of consistency takes the root hashes of the tree at different times r_t and $r_{t'}$ and provides an efficient proof (logarithmic in the number of leaf nodes) that the two root hashes are linked in time. This can be done by providing a set of nodes which can be used to compute both r_t and $r_{t'}$, or by providing a set of nodes containing r_t which can be used to compute $r_{t'}$. The append-only property of these MHTs can be used to implement tamper-evident logs [10].

3.2 Related Work

There has been much recent work on the problems of the current TLS infrastructure and possible solutions to those problems [8, 15, 30]. Systems such as Perspectives [32], Convergence [1], and SSL Observatory [12] introduce a trusted party called a notary, which confirms that a TLS certificate seen by a client is the same as the one seen by the notary. Other approaches attempt to reduce the scope of CAs' authority [13, 16, 19], thereby reducing the amount of trust and power held by CAs today. Below we describe proposals similar to PoliCert that use publicly-verifiable logs.

Google's **Certificate Transparency (CT)** was among the first to propose public logs and append-only MHTs as a way of providing better CA accountability [24]. Its goal is to make all certificates visible to alert domain owners and clients of any possible misbehavior by CAs. CT creates a system of *public logs*, which maintain a database of observed certificates issued by CAs in an MHT. The log then provides a proof of a certificate's presence in the log's database, and this proof can be checked by clients during the TLS handshake. Additionally, the log is publicly-auditable so that any party can fetch proofs of presence or consistency from the log's hash tree to monitor its operations. Special entities called *auditors* and *monitors* may perform these functions as a service for clients, publishing any evidence of CA misbehavior.

However, CT has several critical shortcomings. By itself, CT cannot efficiently prove that a given certificate is absent from a log, since the observed certificates are stored chronologically to achieve the append-only property. (Revocation Transparency proposes the use of an additional tree to provide proofs of absence [23].) Furthermore, CT's main goal is to detect CA misbehavior, and thus it does not actually protect clients from ongoing attacks if an adversary successfully registers a fake certificate at a public log. Finally, CT does not handle certificate revocation. Although auxiliary revocation system was presented [23], so far it is not incorporated with CT. Instead, revocation is proposed to be handled by certificate revocation lists (CRLs) provided with a browser update, as is done in Google Chrome [22] and planned in Mozilla [26], and even then only for a subset of certificates.

The **Accountable Key Infrastructure (AKI)** [20] extends the previous architectures in several ways. First, it allows multiple CAs to sign a single certificate. Additionally, the domain can specify in its certificate which CAs and logs are allowed to attest to the certificate's authenticity. These features provide resilience against a certificate signed by a compromised or unauthorized CA. AKI can also handle key loss or compromise through *cool-off periods*. For example, if a domain loses its private key and registers a new certificate not signed by its old private key, the new certificate will be subject to a cool-off period (e.g., three days) during which the certificate is publicly visible but not usable. This ensures that even if an adversary obtains and registers a fake certificate, the domain has the opportunity to contact the CAs and logs to resolve the issue.

However, to ensure that any log server can provide a proof for a domain's certificate, AKI logs maintain a globally consistent view of the entries that they have for a given domain name. This applies for every certificate operation (registration, update, and revocation), meaning that even frequent certificate updates (such as in the case of short-lived certificates) are subject to successful log synchronization. AKI also requires that each domain name only has one active and valid certificate associate with it at any given time.

Certificate Issuance and Revocation Transparency (CIRT) incorporates a revocation monitoring mechanism into a CT-like architecture [27]. CIRT adds a binary search tree sorted by domain name (called a LexTree), with each node also storing all of the observed certificates for that name (but only the most recent certificate is considered valid). A LexTree is an MHT where a node's hash is equal to its domain name and certificates concatenated with its children's hash values. Using this binary search tree in conjunction with a CT-style append-only tree allows a log to prove with a logarithmic number of nodes that a certificate has been observed and that it has not yet been revoked.

Log proofs in CIRT contain a logarithmic number of nodes, but each node stores all observed certificates for a domain and thus may have a large number of observed certificates. Thus proofs in a LexTree will grow quite large with time, as certificates must be renewed periodically. Additionally, CIRT's LexTree allows each domain to have only a single active certificate at any time, preventing servers from using different certificates at once, which is common practice today [21]. CIRT also cannot handle key loss or compromise; in this event, the only way to recover is to resolve the issue with CAs and logs out of band.

Attack Resilient Public-key Infrastructure (ARPKI) [4] is a system inspired by AKI, which redesigns and improves many aspects of AKI. ARPKI introduces framework for accountability, validation, and consistency checking of public logs. It provides strong security guarantees by offering security against an adversary capable of capturing $n-1$ trusted parties at the same time (where $n \geq 3$ is a system parameter). It also relaxes AKI's synchronization requirement by proposing an accountable synchronization scheme with a quorum of logs involved.The main contribution of that work is that the claimed security properties of ARPKI were formally verified. PoliCert builds on top of ARPKI and extends it with the approaches we describe in this paper.

3.3 Summary of Remaining Challenges

We briefly summarize several challenges that remain despite the previous work. Motivated in part by their shortcomings and by other problems of the existing TLS infrastructure, we identify several important facets of the certificate policy problem that our work addresses.

A core challenge motivated by the previously mentioned schemes is how to overcome the inefficiency of certificate management or validation operations. In particular, logs need to be able to provide efficient proofs of both a certificate's presence and absence in the log's database. Additionally, every certificate registration and update should not require inefficient operations such as the global synchronization of logs, nor should they limit domains to a single certificate.

Another challenge is to incorporate certificate policies into the TLS infrastructure without inhibiting the existing system. Certificate policies allow domains to specify parameters or constraints for their certificates. While AKI and ARPKI maintain policy information such as trusted CAs and logs, this information is embedded into a domain's certificate and thus must be re-registered with every certificate update. Additionally, each domain is limited to a single certificate in order to avoid multiple conflicting policies. We make an important observation about certificates and policies: *while certificates may be updated frequently, the policy behind these certificates is only infrequently updated.* Therefore, the problem of optimizing a new certificate infrastructure to this observation is a central challenge in our work.

We also address the challenge of providing expressive policy control to domains, and explore how this challenge relates to other problems. For example, the current TLS warning model in web browsers is ineffective due to users and browser vendors who decide how a given error should be handled. [2, 14] Users (most of whom are not security-conscious) want to visit their desired website, and often do so despite browser warnings. Browser vendors do not want to lose users and determine how TLS errors should be handled, resulting in a "mass-effect" treatment of domain security. As a consequence, both parties have an incentive to avoid hard failure, which is the only effective protection during an actual attack. Hence, we argue that the domain should influence the error handling process, and we address the challenge of how to provide expressive control to domains.

4. PoliCert OVERVIEW

We provide a high-level overview of the PoliCert infrastructure. We begin by discussing the overarching design principles of PoliCert, and then discuss the salient features of multi-signature certificates, subject certificate policies, and log servers.

4.1 Main Design Principles

Our approach is centered around three main design principles that aim to address the shortcomings of §3.3.

Domain-controlled certificate policy. The primary objective of SCPs is to provide domain owners with greater control over their certificate policy. Specifically, domain owners can specify fine-grained policies governing the usage of certificates issued to them. Additionally, these policies can extend to govern certificate usage in subdomains, allowing for example the owner of `a.com` to constrain certificates for `b.a.com` or `*.a.com`. Additionally, the framework for these policies is easily extensible to allow for even more fine-grained certificate policy control in the future.

Separation of certificates and policies. The defining feature of our proposal is that we separate the keys for policies governing a domain's certificates from the keys for the certificates themselves. We observe that certificates are issued, updated, and revoked more frequently than policies. Because the keys in certificates are used much more frequently (with each TLS setup) and are critical to establishing a TLS connection, they are more likely to be compromised. By separating these keys, we can protect the key used for a domain's certificate policy, which provides control over all of a domain's certificates.

One policy, multiple certificates. We observe that while a domain may have multiple certificates, it has one certificate policy that remains consistent. Because we separate policies and certificates as described above, we can leverage globally synchronised logging scheme to monitor SCPs since domains only have a single SCP. On the other hand, because MSCs change more frequently, we avoid such a scheme for MSCs, monitoring them at selected logs specified by the domain's SCP. Logging MSCs in this way allows us to register and revoke MSCs efficiently without hindrance from global log synchronisation.

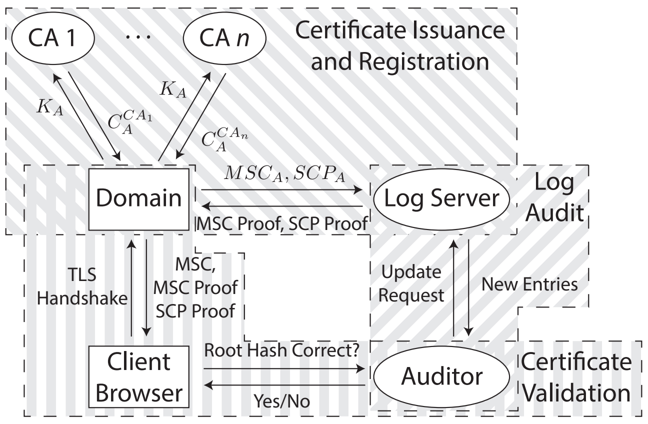


Figure 2: Overview of the MSC registration and validation process. Only a single log server and auditor are shown.

4.2 Overview

The main insight of our work is embodied in *subject certificate policies (SCPs)*, which bind a *policy key pair* (i.e., a *policy public key* and a *policy private key*) to parameters governing and protecting the usage of a domain's certificates. In particular, SCPs provide information such as CAs authorized to sign certificates for a domain, minimum security levels for TLS parameters, and how certain TLS handshake errors should be handled, as well as which of these requirements also must be respected by certificates issued to subdomains. SCPs are themselves certificates, allowing us to leverage their signature and validation mechanisms and to provide resilience against compromised CAs.

We also extend the current X.509 format and propose *multi-signature certificates (MSCs)*, which allow multiple CA signatures to authenticate a single public key. A valid MSC requires only a certain threshold of the signatures to be valid, providing some resilience against CA compromise. We encode MSCs as a series of X.509 certificates for interoperability with today's TLS PKI.

An overview of the PoliCert infrastructure is shown in Figure 2. There are five main parties in PoliCert:

1. The **Client** wants to establish a secure connection to various sites in the Internet. The client browser is designed by a browser vendor, who determines among other things the behavior of the browser in case of errors during the TLS handshake.

2. A **Domain** is a site to which the client wishes to connect. The domain creates one or more keypairs and public key certificates, which are signed and presented to the client during TLS handshakes.

3. A **Certificate Authority (CA)** signs domains' public-key certificates with their own private keys. *Root CAs* have their public keys included in client browsers, while *intermediate CAs* are certified by other CAs.

4. A **Log Server** maintains a Merkle hash tree-based database of domains' certificates and policies. Log servers generate cryptographic proofs of a certificate's validity and periodically provide these proofs to domains. As with root CAs, log servers do not present certificates to clients, but rather have their public keys included in browsers.

5. An **Auditor** periodically fetches a log's database to verify that all of the information is correct. It also allows clients to verify the correctness of proofs from log servers.

As Figure 2 shows, all actions and messages in PoliCert fall into one of three categories. In the first category, *certificate issuance and revocation*, the domain begins by creating a policy keypair and obtaining CA signatures on these to create a subject certificate policy. The domain registers this policy at its trusted log servers. Similarly, the domain creates a keypair for its certificate and obtains signatures from a set of CAs (which can be different from those that signed the domain's SCP) to create a multi-signature certificate and registers it at the log servers. The logs return a signed receipt of these registrations, which contains a time after which the domain's certificate and policy are guaranteed to be in the log.

In the *log audit* category, auditors periodically query the log servers for newly registered or updated entries. The log servers provide these entries and their corresponding proofs, allowing the auditors to verify these proofs. The logs also provide a signed copy of the root hash to the auditors, which can be used to corroborate a client's calculated root hash. For efficiency, the auditor can also sign the root hash that it has computed and provide this to domains, who can then staple this root hash to its certificate and proofs.

The final category is *certificate validation*, in which a client initiates a TLS handshake with the server and receives its certificate, policy, log proofs, and (if included) auditor-computed root hashes. To validate a certificate, a client's browser must verify several criteria: 1. validity of the CA signature(s) on the certificate, 2. presence of the certificate in the logs' databases, 3. absence of the certificate's revocation, 4. compliance of the certificate with all applicable policies, and 5. presence of these policies in the logs' databases.

Because the domain has a signed MSC and corresponding log proof attesting to its current and valid status, it can prove to the client that its certificate meets the first three criteria. The domain may or may not specify any parameters in its own SCP, but will have a proof for the SCP which by design contains all applicable policies for the MSC and proves that these policies are registered and current in the log (described in §5.3). With this proof the domain can prove to the client that its certificate meets criteria 4 and 5.

The client then verifies these proofs by computing the root hash for each of the proofs it has received. If the client did not receive a root hash from the auditor along with the server's information, it can contact one or more auditors to corroborate the root hashes it has computed. If the auditor has the same root hash that the client has computed, then the proofs are considered valid.

5. THE PoliCert ARCHITECTURE

To explain the details of SCPs, we first provide an in-depth treatment of MSCs (§5.1). We then describe the parameters and format of SCPs (§5.2). Finally, we explain the structure of log proofs (§5.3) and the benefits of structuring proofs in this way.

5.1 Multi-Signature Certificates

Multi-signature certificates (MSCs) authenticate a subject's public key using multiple CA signatures. For backwards compatibility with current PKI standards we encode an MSC as a series of standard X.509 certificates[1] authenticating a common public key. Thus with $n \geq 1$ CA signatures, we define an MSC as follows:

$$MSC_A = \{Cert_A^{CA_1}, Cert_A^{CA_2}, ..., Cert_A^{CA_n}, Cert_A^{P_A}\}, \qquad (1)$$

where $Cert_A^{CA_i}$ is an X.509 v3 certificate authenticating A's public key and signed by CA_i, and $Cert_A^{P_A}$ is a *policy binding* signed by A's policy private key. The same private key cannot be used to sign multiple certificates within an MSC (i.e. MSC is signed by distinct CAs).

[1]An alternative approach for MSC implementation could be dedicated extension of X.509 v3, that allows to authenticate the public key by multiple CA signatures.

Every X.509 certificate within an MSC is obtained by the domain in the same way that they are today, with the exception of the policy binding. The policy binding is signed by a private key controlled by A itself, and contains the current version number of A's SCP and a field `CERTS` in an X.509 extension, which lists the hashes of all non-policy bindings within the MSC. This field allows the domain owner to change the certificates within an MSC, and because the policy binding can be generated by A independently of any CAs, these changes can be made quickly.

In order for an MSC to be considered valid, some threshold number (defined in §5.2 by `CERT_TH`) of its certificates must be valid (e.g., not expired and with a valid signature). An MSC with one certificate and a threshold of 1 is equivalent to a regular X.509 certificate today, but contains a policy binding as well. Multi-signature certificates can be revoked by a set of CAs or by the domain itself. A CA can only revoke certificates that it has issued, meaning that an MSC is only revoked as a whole by CAs if enough CAs revoke certificates within the MSC so that it no longer has a threshold number of valid CA signatures.

5.2 Subject Certificate Policies

Subject certificate policies (SCPs) describe parameters regarding the usage and validation of a domain's MSCs. These parameters are bound to the subject's identity as well as to the policy public key. The policy private key is used to sign the policy binding of a domain's MSC, as well as to authorize certificate revocations and policy updates. Because the parameters in an SCP are bound to a domain's identity and policy keypair, we encode an SCP as an MSC in which each X.509 certificate authenticates the policy public key and lists the policy's parameters in an X.509 v3 extension [9]. An SCP must also be signed by a threshold number of CAs to be considered valid.

SCPs do not require a policy binding as other MSCs do, since the public key and parameters of the domain's policy are encoded in each of the SCP's X.509 certificates. Like any other MSC, however, an SCP must be signed by one or more CAs and registered at the log servers to be considered valid. Since domains are expected to only infrequently change their policy, SCPs are assumed to be stable (barring catastrophic events such as a weakness in a widely-used encryption scheme). Therefore, we require that SCPs be valid during an extended time period (e.g., five years).

All fields in an SCP are optional, except for the policy version. Browser vendors set default values for each field so that if a field is not specified by any applicable policy for a domain, that field takes the default value provided by the browser. A subject certificate policy contains the following fields:

1. General parameters

   `POLICY_VERSION`: the version number of the current policy.

   `LOG_LIST`: the domain's trusted logs, at which its certificates, revocations, and policies are registered. If blank, all logs are considered trusted.

   `LOG_TIMEOUT`: how long proofs from the above logs are considered valid.

   `CA_LIST`: CAs authorised to sign the domain's certificates and policies. If blank, all CAs are considered trusted.

   `CERT_TH`: the minimum number of CA signatures that must be valid on a MSC, excluding the signature by the domain's policy private key. This parameter must be positive and cannot exceed the number of CAs in `CA_LIST` (if the field is not blank).

   `REV_KEY`: flag which allows domain to revoke any of its MSCs using the private key connected with the domain's policy.

2. Additional parameters of standard certificates

   `EV_ONLY`: flag specifying that only extended validation (EV) certificates are valid in an MSC.
   `MAX_PATH_LEN`: maximum length of a certificate chain.
   `WILDCARD_FORBIDDEN`: forbids wildcard certificates [9].
   `MAX_LIFETIME`: the maximum duration of a certificate's validity.

3. Security parameters

   `CERT_SEC`: minimum security level of MSC's standard certificates.
   `TLS_SEC`: minimum security level of negotiated TLS parameters.

4. SCP update parameters

   `UP_CA_MIN`: number of signatures required to update the policy.
   `UP_CA_TH`: threshold number of signatures required to update policy if not signed by the policy private key.
   `UP_COP_UNTRUSTED`: cool-off period applied if the new policy is signed by a CA outside `CA_LIST`.
   `UP_COP_UNLINKED`: cool-off period applied if the new policy is not signed by the policy private key.

5. Soft/hard failure configurations (0 for soft failure, 1 for hard failure)

   `FAIL_CERT_TH`: if the MSC is invalid (`CERT_TH` not satisfied).
   `FAIL_TLS`: if the TLS security level is too low.
   `FAIL_EXP`: if the log proof has expired (older than `LOG_TIMEOUT`).
   `FAIL_POL`: if the policy version number is not the most recent.
   `FAIL_LOG`: if the log proof is invalid.
   `FAIL_*`: all other failures.

6. *Inheritance mask*, describing which fields are inherited by subdomains.

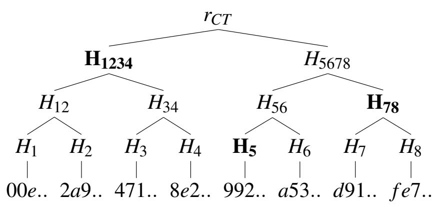
The value of `CERT_TH` also determines the number of CAs required to revoke an MSC. If an MSC is signed by n CAs, then $n -$ `CERT_TH` $+ 1$ revocations will invalidate the MSC. The `CERT_SEC` parameter specifies a minimum key length and strength of the cryptographic primitives used by the domain and CA to create the given certificate. The `CERT_SEC` and `TLS_SEC` parameters can have values of 0, 1, or 2, corresponding to low, medium, or high security. The evaluation of security level can be realized using standards and reports like [11, 17]. The values associated with these levels are set by browser vendors and can be changed by browser updates.

The ability to update these security levels allows browser vendors to protect users from cryptographic vulnerabilities as long as users update their browsers. These security levels also protect users and domains from dangerous misconfigurations. Each failure configuration has a value of 0 for a soft failure or 1 for a hard failure. In the case of soft failure, the browser would display the reason for the failure and give users the option to accept the certificate anyway, while a hard failure does not allow users to continue. These configurations allow the domain to take an active role in evaluating and mitigating threats to the security of its connections.

5.3 Log Servers

Log servers are trusted and highly-available entities that monitor issued certificates, revocations, and policies, which can be registered at log servers and maintained in the log's database, stored as MHTs. All logs maintain a *certificate tree*, which tracks certificates (MSCs), and a *policy tree*, which tracks policies (SCPs), and these trees are based on Merkle hash trees [25] as shown in Figure 1.

As explained in §3.1, the MHT structure allows the log server to produce efficient and cryptographically-secure proofs that a leaf is present in or absent from the tree. In the PoliCert infrastructure, these proofs demonstrate that a certificate is logged, not revoked, and compliant with all applicable subject certificate policies, as mentioned in §4.2. To avoid frequent updates to the trees and thus to the proofs, objects are batch-added periodically (e.g., every hour). The update frequencies of log servers are public information, allowing clients to query them after each update or as needed.



Proof of $MSC_{mail.a.com}$'s presence:

$$\{MSC_{mail.a.com}, R_{MSC_{mail.a.com}}, H_5, H_{78}, H_{1234}\}. \qquad (2)$$

Figure 3: Example of a certificate tree with a presence proof. The hash of $MSC_{mail.a.com}$ is assumed to begin with a53.

When an object is accepted for insertion into a tree, the log server schedules it and returns a signed *receipt* with a future time at which the object is guaranteed to be present in the log's database. Log servers are required to produce a proof for a specific entry (certificate or policy) on request, which certify the current validity of that entry. Log servers are also required to provide a proof of consistency by showing that its database has been extended from a previous version of the database with valid transactions.

In certificate and policy trees, a leaf node N_x is defined as a tuple (L_x, D_x, V_x) where L_x is a *label* used to order the nodes in the tree, D_x is a set of *data* associated with the node, and V_x is the node's *value* whose hash is the parent node's value. A non-leaf node consists only of a value and will have one or two child nodes; in the first case the node's value will be the hash of its child's value, and in the second case the node's value will be $H(V_1)\|H(V_2)$, where V_1 and V_2 are its children's values.

In a certificate tree such as the one shown in Figure 3, each leaf node N_{MSC_A} represents a multi-signature certificate MSC_A for the domain A in the form:

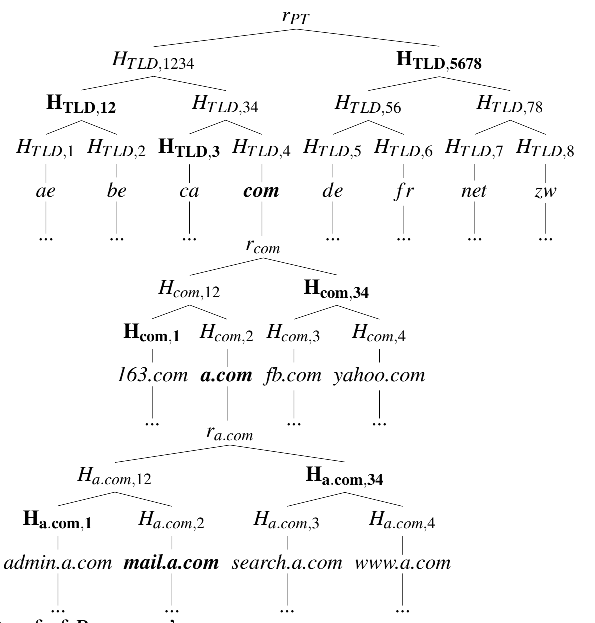
$$L_{MSC_A} = H(MSC_A) \qquad (3)$$
$$D_{MSC_A} = (MSC_A, R_{MSC_A}) \qquad (4)$$
$$V_{MSC_A} = MSC_A \| R_{MSC_A} \qquad (5)$$

where R_{MSC_A} is a revocation message for MSC_A or null if MSC_A is still valid. Note that if MSC_A has not yet been revoked, then $V_{MSC_A} = MSC_A$.

In a policy tree such as the one in Figure 4, a node N_A represents a domain name A. There are three data fields associated with N_A: 1. the SCP P_A of A (null if A has no SCP), 2. the policy subtree S_A of A (see below), and 3. the root hash r_A of S_A. The *policy subtree* is an MHT of all of A's immediate subdomain nodes (e.g., all nodes `*.com` for `.com`). A leaf node N_A in a policy tree has the following fields:

$$L_A = A \qquad (6)$$
$$D_A = (P_A, S_A, r_A) \qquad (7)$$
$$V_A = P_A \| r_A \qquad (8)$$

A policy tree's structure provides several useful properties. The hierarchical organization of the tree according to the DNS namespace hierarchy makes it straightforward to find all policies pertaining to a domain name. Furthermore, because each node's value includes its SCP, a proof for a node N_A will contain all policies of its higher-level domains, all of which may apply to A's certificates. This simplifies policy enforcement and forces logs to show that all applicable policies to a domain are logged and current.



r_{PT}
$H_{TLD,1234}$ $\mathbf{H_{TLD,5678}}$
$\mathbf{H_{TLD,12}}$ $H_{TLD,34}$ $H_{TLD,56}$ $H_{TLD,78}$
$H_{TLD,1}$ $H_{TLD,2}$ $\mathbf{H_{TLD,3}}$ $H_{TLD,4}$ $H_{TLD,5}$ $H_{TLD,6}$ $H_{TLD,7}$ $H_{TLD,8}$
ae be ca **com** de fr net zw
r_{com}
$H_{com,12}$ $\mathbf{H_{com,34}}$
$\mathbf{H_{com,1}}$ $H_{com,2}$ $H_{com,3}$ $H_{com,4}$
163.com **a.com** fb.com yahoo.com
$r_{a.com}$
$H_{a.com,12}$ $\mathbf{H_{a.com,34}}$
$\mathbf{H_{a.com,1}}$ $H_{a.com,2}$ $H_{a.com,3}$ $H_{a.com,4}$
admin.a.com **mail.a.com** search.a.com www.a.com

Proof of $P_{mail.a.com}$'s presence:

$$\{P_{mail.a.com}, r_{mail.a.com}, H_{a.com,1}, H_{a.com,34}, P_{a.com},\\ H_{com,1}, H_{com,34}, P_{com}, H_{TLD,4}, H_{TLD,12}, H_{TLD,5678}\}. \quad (9)$$

Figure 4: Example of Policy Tree, where bold nodes are used for *mail.a.com* policy's presence proof.

However, the structure of the policy tree also has a drawback: the size of a log proof is not quite logarithmic in the number of nodes. The size of a proof is instead $O(m \log_2 n)$, where m is the number of levels in the domain name (e.g., three for `mail.a.com`) and n is the greatest number of entries at any level of the domain name (112M names under `.com` [31] in this case). However, in almost all cases m will be very small (less than 5) and n will likely never exceed the number of `.com` domain names.

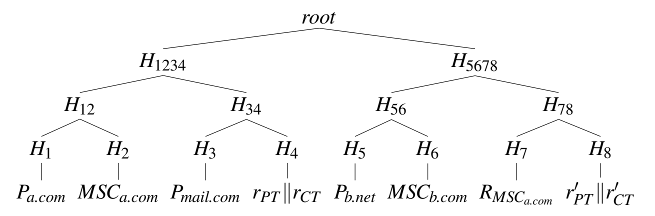
If a domain such as `b.a.com` does not have a SCP, then the log sends one of three classes of proofs to show policy compliance:

1. if any subdomain of `b.a.com` has its own policy then log shows that $P_{\texttt{b.a.com}}$ is empty.

2. if no domain at the same domain level (e.g., `x.a.com`) has an SCP, then the log shows that $S_{\texttt{a.com}}$ is empty.

3. if there are SCPs at the same domain level, then the log sends a proof starting with the two nearest domain names on either side of `b.a.com` (e.g., `a.a.com` and `c.a.com`).

Both the certificate and the policy tree represent only the set of *currently valid* certificates. In order to prove the consistency of a log's database over time, the log maintains an append-only MHT called a *consistency tree*. The consistency tree contains all SCP and MSC registrations, updates, and revocations in chronological order. Additionally, upon each update the log appends the concatenation of the root hashes of the current certificate and policy trees to the consistency tree. The log then provides a proof showing that the root hashes are the most recent ones in the consistency tree, and this proof is sent with the appropriate proofs from the certificate and policy trees. The example of consistency tree is shown in Figure 5.



root
H_{1234} H_{5678}
H_{12} H_{34} H_{56} H_{78}
H_1 H_2 H_3 H_4 H_5 H_6 H_7 H_8
$P_{a.com}$ $MSC_{a.com}$ $P_{mail.com}$ $r_{PT}\|r_{CT}$ $P_{b.net}$ $MSC_{b.com}$ $R_{MSC_{a.com}}$ $r'_{PT}\|r'_{CT}$

Figure 5: Example of a consistency tree. At every update roots of policy and certificates tree are appended as the last value.

6. PROTOCOL DESCRIPTIONS

We now describe in detail the steps by which a certificate is created, registered, and validated in the PoliCert architecture. This process takes place through SCP registration, MSC registration, and client validation. We then describe how certificates and policies can be updated or revoked.

6.1 Policy Registration

Before a domain can register a certificate, it must first create and register a subject certificate policy. A domain A at `a.com` creates its own policy by specifying any parameters it wishes (described in §5.2) and then obtaining CA signatures on the policy. Recall that only the policy version is required in a valid SCP, so A can choose to only specify a version number if it does not want to enforce any policy on its certificates. A then obtains a number of CA signatures on the policy to create a signed SCP.

Once A has a valid SCP, it can register the policy at one or more logs by sending a registration request containing the SCP. Logs must globally coordinate to ensure that two different SCPs for A are not active simultaneously (to a given time tolerance). If the registration is successful, the logs send to the domain signed registration receipts containing the policy and a time after which the SCP is guaranteed to be recorded in the log's database. This receipt serves as a temporary proof of presence for the SCP.

Because each domain must have an SCP (in order to be protected by the PoliCert) and these policies can greatly impact a domain's certificate usage, policies can only be updated, not revoked, and must meet several criteria to be accepted by the logs. In particular, the new policy must be signed with the policy private key of the old policy, as well as by a threshold number of CAs (`UP_CA_MIN`). To handle the loss or compromise of the policy private key, we also allow domains to create an *unlinked* policy not signed with their policy private key. However, in this case, the domain may need to obtain a larger number of CA signatures (`UP_CA_TH`) and a "cool-off" period is enforced as in AKI [20]. During this cool-off period, the new policy is visible but not active. The domain can query the logs for any policies for its name in a cool-off period in order to detect fraudulent update attempts. Therefore, even if an adversary manages to get a fake version of the domain's policy accepted by the logs, the true domain will have time to contact its CAs and logs to resolve the issue.

To update its policy, a domain A sends an *update request* containing P'_A. The log receives this request, verifies the signatures on the new policy P'_A, and checks that its policy version number is one more than that of P_A. The log must then check whether its policy update meets the above criteria, such as being signed by at least $min($`UP_CA_MIN,UP_CA_TH`$)$ CAs from `CA_LIST` (of the *old* SCP). If not, the logs enforce the cool-off period as specified in the SCP.

After a successful update the log confirms the presence of the new policy by returning a proof of presence to the domain. The domain must then update its own MSCs to signal the use of the new policy. Since as shown in Equation 1 the policy binding in each MSC contains the policy number and is signed by the policy private key, the domain does not need to contact other CAs in order to update its MSCs. After updating, it simply submits its new MSCs to the appropriate logs.

6.2 Certificate Registration

To create a multi-signature certificate, a domain first creates a keypair with which it will establish TLS connections to clients. It then obtains standard X.509 certificates from CAs, and combines them along with a policy binding (signed by its policy private key from the previous section) into an MSC. The domain then sends a *certificate registration request* to its trusted logs. In contrast to the SCP registration process, no global coordination of logs is required. Each log server receiving the registration request validates the certificate and ensures that it is in the `LOG_LIST` field of the domain's SCP. If so, the log returns a *registration receipt*, a signed statement containing the certificate that was registered and a time by which the certificate is guaranteed to be in the log.

As stated in §5.1, every MSC can be revoked by its owner, by a threshold number of CAs, or by a parent domain (if the policy allows). In order to revoke given MSC, one of the above three sends to all logs on the domain's `LOG_LIST` a *revocation request*, which contains the certificate and the appropriate signatures. If a log receives a revocation request and does not have the corresponding certificate, it adds the MSC along with the revocation request to its certificate tree. The log must not discard the revocation request without this step because otherwise an adversary could register the revoked MSC at a log server that previously had not recorded the certificate and use it as a valid MSC. Once the log has processed the revocation request, it returns a signed *revocation receipt* with the certificate along with a time after which the revocation is guaranteed to be present in the log's certificate tree.

6.3 Certificate Validation

Before an MSC can be validated, a client must first read the parameters in the domain's SCP P_A which contain the list of trusted CAs and logs needed for the rest of the validation process. Recall that policy fields can be inheritable (§5.2) and that a proof for a domain's policy contains all policies of parent domains from which fields can be inherited (§5.3). Therefore, we can extract from this proof a list P_{list} of the parent domains' SCPs. However, since almost all fields in an SCP are optional, it is possible that some policy fields will not be specified by any applicable policy. In this case, the browser will determine a default value for any unspecified parameters as browsers do today. We call this "default policy" $P_{browser}$.

Once we have the above we can determine the appropriate parameters for each TLS connection. We achieve this through Algorithm 1, which returns the set of parameters as a dictionary. The algorithm treats each policy as a dictionary in which values corresponding to a key can be accessed as $P[key]$. The dictionary that will eventually be returned starts as $P_{browser}$, which usually contain the least conservative parameter values. These default values are then overwritten by the fields specified in P_A. Next for each policy in P_{list}, the client checks whether the fields are inheritable (i.e., the corresponding bit in the SCP's inheritance mask is set to 1). If so, the inherited value overwrites the current one, but only if the new value is more conservative. For example, if a domain's policy specifies a high security level for TLS connections, and a parent domain's policy specifies a medium level that is inheritable, then the parent domain's security level is not inherited because it could result in a lower security level in the domain's TLS connections.

Algorithm 1: Determine appropriate SCP parameters from the SCP hierarchy.

```
$P_A$ - policy of domain contacted by a browser
$P_{list}$ - list of policies of parent domains (with respect to $P_A$)
sorted by level (from the most specific domain name to TLD)
$P_{browser}$ - default browser policy

policyInheritance($P_A, P_{list}$):
  $d = P_{browser}$
  for attr $\in P_A$
    $d[\texttt{attr}] = P_A[\texttt{attr}]$
  for $P \in P_{list}$
    for attr $\in \{$LOG_LIST, CA_LIST$\}$
      if $P[\texttt{attr}].isInherited$
        $d[\texttt{attr}] = d[\texttt{attr}] \cap P[\texttt{attr}]$
    for attr $\in \{$CERT_TH, CERT_SEC, TLS_SEC,
              FAIL_CERT_TH, FAIL_TLS, FAIL_EXP,
              FAIL_POL, FAIL_*, UP_CA_MIN, UP_CA_TH,
              UP_COP_UNTRUSTED, UP_COP_UNLINKED$\}$
      if $P[\texttt{attr}].isInherited$ and $d[\texttt{attr}] < P[\texttt{attr}]$
        $d[\texttt{attr}] = P[\texttt{attr}]$
    for attr $\in \{$LOG_TIMEOUT, MAX_PATH_LEN,
              MAX_LIFETIME$\}$
      if $P[\texttt{attr}].isInherited$ and $d[\texttt{attr}] > P[\texttt{attr}]$
        $d[\texttt{attr}] = P[\texttt{attr}]$
    for attr $\in \{$EV_ONLY, WILDCARD_FORBIDDEN$\}$
      if $P[\texttt{attr}].isInherited$ and $P[\texttt{attr}]$
        $d[\texttt{attr}] = P[\texttt{attr}]$
  return $d$
```

Once the policy parameters have been determined, the domain's MSC must be "pre-validated." The client checks whether the X.509 certificates within the MSC are issued for the correct domain and whether the certificates all authenticate the same public key. The client browser also checks that the version number of the policy obtained from Algorithm 1 matches the version number in the MSC's policy binding, and that the hash of each certificate appears in the `CERTS` field of the policy binding. The client browser then verifies the signature on the policy binding, which is signed by the domain's policy private key.

With pre-validated MSC and SCP parameters, the client browser can then validate the MSC by following Algorithm 2. The most important parameter for this validation is `CERT_TH`, which describes how many standard certificates must be valid in a multi-signature certificate in order for the MSC to be valid. A certificate is classified as valid and counts toward `CERT_TH` if 1. its signature is successfully verified, 2. the private key used to sign the certificate has not already signed another certificate counting towards `CERT_TH`, and 3. the certificate meets the constraints set by the SCP parameters. If `CERT_TH` is met, then the client browser proceeds to validate the log proofs for the MSC.

6.4 Log Proof Validation

After a successful SCP or MSC registration, the log returns a registration receipt promising that the certificate or policy will be added to its database within a certain amount of time. This registration receipt can be used as a short-term confirmation that an SCP or MSC is in the log, but log proofs are more commonly used for this purpose. To successfully establish a connection to the domain, the client requires proofs that the policy is registered, as well as proofs that the MSC is registered and not yet revoked.

Algorithm 2: MSC validation.

```
$d$ - dictionary generated by $policyInheritance()$ execution
$isLegacyValid()$ - standard validation for single certificate

$\textbf{\textit{isMSCValid}}(d, \{Cert_A^{CA_1}, Cert_A^{CA_2}, ..., Cert_A^{CA_N}, Cert_A^{P_A}\})$:
  $S = \{\}$
  for $Cert_A^{CA_x} \in \{Cert_A^{CA_1}, Cert_A^{CA_2}, ..., Cert_A^{CA_N}\}$
    if ($CA_x \in d[\texttt{CA\_LIST}]$ and $isLegacyValid(A, Cert_A^{CA_x})$)
      if ($getCertSec(Cert_A^{CA_x}) < d[\texttt{CERT\_SEC}]$) or
        ($getPathLen(Cert_A^{CA_x}) > d[\texttt{MAX\_PATH\_LEN}]$) or
        ($getLifetime(Cert_A^{CA_x}) > d[\texttt{MAX\_LIFETIME}]$) or
        (not $Cert_A^{CA_x}.isEV$ and $d[\texttt{EV\_ONLY}]$) or
        ($Cert_A^{CA_x}.isWildcard$ and $d[\texttt{WILDCARD\_FORBIDDEN}]$)
          continue
      $S = S \cup \{CA_x\}$
  return $|S| \geq d[\texttt{CERT\_TH}]$
```

While anyone can request such proofs from a log, proofs are often periodically retrieved from the log by the domain and stapled to the MSC and SCP during connection setup. To request a log proof, the domain sends a *proof request* to the log containing a hash of its MSC. The log uses this hash to locate the appropriate leaf node in its certificate tree and generates a proof of presence or absence (as in Equation 2) for this node.[2] The log also produces a proof of presence for the domain's policy (following Equation 9), as well as a proof that the policy and certificate trees' root hashes is the most recent one recorded in the consistency tree. The log then sends these three proofs along with a signed root hash of the consistency tree to the domain. The domain can pass these proofs and hashes on to the client.

There is also a possibility that the log does not have a proof for an SCP or MSC. It may be the case that the MSC, SCP, or both does not have a corresponding log proof because the log has not yet updated its database to reflect a registration. In this case, a registration receipt from the log suffices as a proof of presence so that domains who newly register a certificate and policy can begin serving customers as soon as possible. It may also be the case that the domain has not yet adopted PoliCert. In this case, the client can request a proof of absence for the domain's SCP from one or more of the log servers. This prevents an adversary from obtaining a bogus certificate for a domain and suppressing the log proofs to make it seem as though the domain has not yet deployed PoliCert. When requesting a proof of absence, the client may want to proxy the request through another log [1] or request several decoy proofs to preserve the privacy of its queries.

6.5 Connection Establishment

The client initiates a TLS connection with a domain using Algorithm 3. In the first *ClientHello* message, the client browser sends the latest seen version numbers of domain's policy and all parent domain policies it has from the previous connections. The domain then sends its multi-signature certificate, subject certificate policy (if the browser does not have the latest version), and the appropriate log proofs or registration receipts showing that the MSC is valid and compliant with appropriate policies. The browser validates the proofs, determines the policy parameters, and validates the domain's MSC. The browser negotiates the TLS connection with the appropriate security level and, if all other operations are successful, accepts the connection.

[2]If a revocation request for an MSC has been accepted but the log has not yet updated its database, the log returns the revocation receipt for the certificate.

Algorithm 3: TLS connection establishment.

```
$preValidation()$ - pre-validates MSC, policies, and
proofs (§6.3). If pre-validation fails and the browser has a
stored policy for the domain, then FAIL_POL and FAIL_LOG
from this policy can be applied in the appropriate scenario.
Otherwise, it hard fails.
$getSec()$ - evaluates security level of TLS parameters
$fail(S)$ - if $S \neq \{\}$ fails with $max(S)$ failure scenario ($0$ - $soft$
$fail$, $1$ - $hard fail$) and shows all occurred errors to client

Client                                A's Server          Log
                                      proofs request  $\longrightarrow$
                                      (every          $\longleftarrow$ proofs
ClientHello                            LOG_TIMEOUT)
(indicates stored policies)   $\longrightarrow$
                              $\longleftarrow$  $MSC_A, P_A, P_{list}, proofs$
$S = \{\}$
$preValidation(...)$
$d = policyInheritance(P_A, P_{list})$
if not $isMSCValid(d, MSC_A)$
  $S = S \cup \{d[\texttt{FAIL\_CERT\_TH}]\}$
if $getSec(TLSParams) < d[\texttt{TLS\_SEC}]$
  $S = S \cup \{d[\texttt{FAIL\_TLS}]\}$
if $proofs\ are\ expired$
  $S = S \cup \{d[\texttt{FAIL\_EXP}]\}$
if $Log \notin d[\texttt{LOG\_LIST}]$
  $S = S \cup \{d[\texttt{FAIL\_LOG}]\}$
$fail(S)$
```

7. IMPLEMENTATION AND EVALUATION

In order to evaluate the deployment feasibility and performance of PoliCert, we implemented each of the parties in the architecture. The client-side code, which includes Algorithms 1 and 2 as well as part of Algorithm 3, was implemented by extending the Chromium web browser. We deployed our domain on both Apache and Nginx HTTP servers, which were equipped with special scripts to send proof requests and periodically process responses from the log. The domain sends these proofs to clients during the TLS handshake protocol. Because CAs have a similar role in PoliCert as they currently do, we used standard tools such as OpenSSL to handle CA certificate operations, and created multi-signature certificates and policies with several trusted CAs of our making. We used elliptic curve cryptography for our keypairs, with ECDSA [18] as our signature scheme. We selected the elliptic curve *secp521r* [5, 7] from OpenSSL 1.0.1f, and also used this version for all cryptographic operations. We implemented our log servers in C++ (gcc 4.8.2) using SHA-512 as the hash function for the Merkle-hash trees. Auditors compare the signed root hash value from the logs with those they have stored to detect potential misbehavior.

For our evaluation we deployed three machines running Linux 3.13.0-24-generic x86_64, representing a log (Intel i5-3210M, 2.50 GHz, 4GB of RAM), a domain/server (Intel i5-3210M, 2.50 GHz, 4GB of RAM), and a client browser (Intel i5-3470, 3.20 GHz, 8GB of RAM). Since the log serves many types of requests, we sent 500 of each type of request (policy registration, policy update, certificate registration, certificate revocation, and proof request) to the

Action	Avg.	Median	Min.	Max.
Policy Registration	10.02	9.79	6.50	16.33
Policy Update	10.75	10.27	7.70	14.52
Certificate Registration	7.35	6.73	5.84	12.58
Certificate Revocation	4.90	4.57	2.69	9.37
Proof Request	8.99	8.58	5.64	18.00

Table 1: Log's processing time (in *ms*) for different requests.

Action	Avg.	Median	Min.	Max.
MSC Pre-validation	0.79	0.78	0.75	1.40
SCP Processing (Alg. 1)	0.50	0.49	0.45	1.05
MSC Validation (Alg. 2)	0.60	0.59	0.55	1.22
Proof Validation	1.45	1.44	1.39	1.78
Complete Validation	3.34	3.32	3.17	4.84

Table 2: Browser's processing time (in *ms*) in details.

log. We show the average, median, minimum, and maximum processing times for each request type in Table 1.

The synchronization protocol, required for global logs' coordination in the case of policy registration and update, was realized with a two-phase commit protocol [3], where all messages are signed by participants.

To evaluate the computational effort required by the browser, we executed the browser's side of verification 500 times. In this scenario the browser was connecting to `localhost.net` serving an MSC, its own policy, and a policy of `.net`. Each multi-signature certificate consisted of three standard certificates. The total time taken by the browser was divided into several categories as shown in Table 2: pre-validation of MSCs, SCP parameter processing, MSC validation, and log proof validation.

The MSC with policies was sent in the *Server Certificate* message of the TLS Handshake, allowing us to deliver the multi-signature certificate and policies without any changes to the browser. The log proofs were sent to the browser via the OCSP Stapling extension [6], saving the client the need to fetch the proofs separately. Since the applicable policies were already sent for SCP processing (Algorithm 1), the SCPs of the domain and its parent domains do not have to be included with the proof. Rather, for better efficiency the domain can omit this information and instead have the client browser fill in the gaps with the policies, significantly decreasing message overhead.

However, the highest message overhead is due to the structure of MSCs, which contain multiple X.509 certificates for the same key. Because of this structure some fields are duplicated, once for each certificate in the MSC. However, we can decrease this overhead by compressing the certificates during transmission.

Our results show that PoliCert introduces a small overhead in logs, around 9 ms per proof request (the most common request sent to logs). This means that even on standard hardware the log can handle about 111 proof requests per second. However, we expect that servers will only infrequently query logs (e.g., every few hours), since log proofs can be stored and reused for some time. Notice that in interactions between the client browser and the domain, the overhead is 3.3 ms on average, which is short enough to be unnoticeable by users [29].

8. POSSIBLE ENHANCEMENTS

We now present an overview of a possible security enhancement to PoliCert which combines our architecture with another proposal called ARPKI [4]. In ARPKI, a domain interacts with the PKI, employing n trusted entities, and the system prevents attacks even when an adversary controls $n-1$ of these entities. In our proposal, we define $n \geq 3$ to be a system parameter, where the log contacted by the domain is one of the n trusted parties along with $n-1$ auditors. We show an overview of the architecture and general message flow in Figure 6. The logs' operation is similar to that described in §6, but their correctness is asserted by additional trusted parties (auditors). Auditors[3], as in ARPKI, can detect log's misbehaving and disseminate that information among each other.

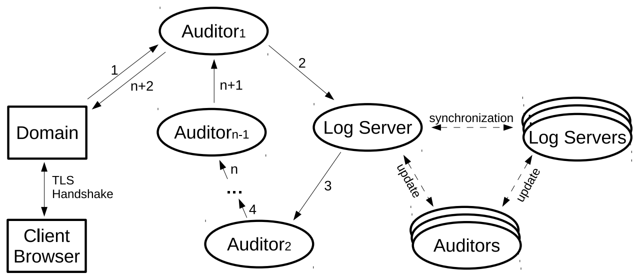


Figure 6: Overview of architecture of PoliCert in combination with ARPKI.

As in PoliCert, the first step for a domain is is **policy creation and registration**. We introduce new SCP's parameter `AUDITOR_LIST`, which defines at least $n-1$ auditors trusted by the domain. The domain creates an SCP and a registration request along with a list of auditors to confirm that registration. The request, however, is sent first to the first auditor in the list, who then passes it to the log. The log performs the same checks and synchronizations as in §6.1, but the registration receipt is then returned to the second auditor, who checks that the registration was carried out correctly. This auditor then signs the receipt and passes it to the next auditor. Each of $n-1$ auditors performs similar checks and verifies whether the log indeed appended the SCP in the next update period. Finally, the domain receives a receipt confirmed by $n-1$ trusted auditors. For SCP updates the message flow is identical, and the auditors verify whether or not the log correctly updated policy (e.g. if a potential cool-off period was satisfied).

For **certificate registration** the message flow is the same as previously. The log operates as presented in §6.2, and the only difference is that log sends the receipt to an auditor instead of to the domain. At the end the domain obtains a registration receipt confirmed by auditors, and the auditors again checks log's content after the next update for the registered MSC. In the case of a certificate revocation, auditors similarly confirm that the certificate was revoked in the log.

As in PoliCert, domains periodically send proof requests and receive from the log proofs of their SCP and MSC's presence. However, these proofs are additionally confirmed by $n-1$ auditors. **Browser-based validation** is almost the same as presented in §6.3. The only extra step is that the browser also checks if the proofs (or receipts) are confirmed by $n-1$ auditors from the domain's `AUDITOR_LIST`.

9. SECURITY ANALYSIS

We now conduct an informal security analysis of PoliCert. In this analysis we assume that a domain A has correctly registered its policy and MSCs at the logs. We consider an adversary who is able to capture trusted elements of the system (logs, CAs, and domains private keys) and whose goal is to impersonate A's website.

[3]In ARPKI CAs take the auditor's role.

Our first claim is that *an adversary without A's policy private key cannot create a valid MSC for A*. Constructing a multi-signature certificate (Equation 1) requires a policy binding which combines a set of X.509 certificates into one logical MSC. Because the policy binding must be signed by *A*'s policy private key, an adversary without that key cannot create any valid MSC.

Even if we assume that the adversary has access to the policy's private key, then we can show that *the adversary cannot impersonate A without compromising at least* `CERT_TH` *of A's trusted CAs (from* `CA_LIST`*)*. This is due to the MSC validation process (Algorithm 2), which requires a valid MSC to contain at least `CERT_TH` valid X.509 certificates. This threshold is also a lower bound, since the domain may inherit a higher threshold from its parent domain's SCP.

An adversary who has compromised the required number of *A*'s trusted CAs and *A*'s policy private key can impersonate *A* by creating a malicious MSC and serving it to the clients. However, to mount this MitM attack the adversary must receive confirmations (a registration receipt or log proof) from the log. This requires first registering the malicious MSC, which would make the fraudulent certificate publicly visible. The adversary could also attempt to update the SCP itself, but this would require compromising at least `UP_CA_MIN` CAs, which may be more than `CERT_TH`.

If we assume that logs are not malicious, then all of the above attacks can be detected since all of the adversary's actions would become publicly visible. If we assume a worst-case scenario where the adversary has compromised at least `CERT_TH` of *A*'s trusted CAs, *A*'s policy private key, and one or more of *A*'s trusted logs, then the adversary could forge an MSC, and the necessary log proofs or registration receipts. However, even in this case the MSC would have to comply with *A*'s SCP, constraining the malicious MSC. Additionally, while the adversary could send the client registration receipts and never add the fraudulent MSC to the log, this action would also be detectable after some time, as eventually the registration receipts would expire and anyone querying the log after the receipts' expiration would find that the MSC was not in the log.

Security level can be increased by contacting number of auditors, that could confirm log's actions. In §8, we propose such architecture. The achieved property is that with successfully registered SCP, an adversary even with $n-1$ parties compromised, cannot launch impersonation attack undetectably, as n parties are actively involved in asserting correctness of SCPs and MSCs.

10. INCREMENTAL DEPLOYMENT

The PoliCert infrastructure is designed to be interoperable with, and incrementally deployable alongside, the current TLS PKI. One important feature of our architecture is that CAs act no differently than they currently do, allowing them to preserve their existing business model. Because an MSC is mostly made up of a series of X.509 certificates, it can be implemented and validated using currently available tools and methods, and served in the standard TLS handshake. Moreover, PoliCert even works with legacy software, since all major browsers only validate the first certificate they receive from a domain. Thus for a legacy browser, only the first certificate in the MSC needs to be valid, and the rest of the MSC will be ignored.

During incremental deployment of PoliCert there is the possibility that an adversary may attempt a downgrading attack. In this attack, the adversary impersonates a domain and claims that it has not yet deployed PoliCert and hence has no proofs for its certificate or policy. In this situation a PoliCert-enabled client browser can obtain a proof of absence for the domain's SCP as discussed in §6.4. Because the logs synchronise SCPs globally, any log should be able to return a proof of presence or absence for the domain's policy. In fact, a proof of absence for a domain's SCP allows a client to establish a TLS connection with a legacy domain, further illustrating the interoperability of PoliCert with the current PKI.

The hierarchical structure of SCPs also allows legacy domains to benefit from the protection of PoliCert. For clients deploying PoliCert, validating a legacy domain's certificate will require fetching a proof of absence for the domain's SCP. However, this proof will also contain all SCPs of parent domains, if they exist. With wisely chosen SCP parameters, a parent domain can protect all of its subdomains by, for example, forbidding wildcard certificates or limiting the lifetime of a certificate. These parameters constrain any certificates that an attacker might try to craft for the domain, providing resilience against malicious certificates even for legacy domains.

Additionally, parent domains can leverage the hierarchical structure of SCPs to incentivise the adoption of PoliCert or higher security levels. For example, by setting `CERT_TH` to 2 and making the parameter inheritable, a parent domain can force the adoption of PoliCert for all of its subdomains. A parent domain could also set `MAX_LIFETIME` and make it inheritable to ensure that their subdomains regularly renew their certificates. From security perspective it may be worth to consider minimum/maximum values e.g. for update parameters. It requires debate however it is reasonable, especially for top level domains, whose policies may influence millions of subdomains.

A domain also has incentives to deploy PoliCert due to privacy and efficiency reasons. For a legacy domain the client must fetch a proof of absence for the domain's SCP, incurring extra latency when establishing a TLS connection to these domains and leaking privacy if the client directly contacts a log server for such a proof. Deploying PoliCert would cause the domain to staple log proofs of its SCP, saving extra round trips and preserving the client's privacy.

Moreover, PoliCert can be built upon currently deployed system like CT. That is caused by a fact that both systems employ similar data structure (consistency tree) as a core element.

11. CONCLUSION

In this work, we presented PoliCert, a comprehensive solution that addresses a range of problems with the current TLS ecosystem. PoliCert secures domains' certificates and allows domains to create policies for their certificates as well as their subdomains' certificates. Additionally, PoliCert handles all operations over a certificate's lifetime (creation, registration, validation, and revocation) in a secure and transparent manner.

By introducing long-term policies, we make a domain's security statements stable and transparent, and narrow the range of malicious certificates that an attacker can forge. These policies can be created by IT/security departments and applied to subdomains, allowing experts to easily coordinate certificate policies for their networks. Additionally, the hierarchical enforcement of SCPs allow domains to protect subdomains from human errors such as misconfiguration.

In order to evaluate the feasibility of our system, we fully implemented it and sketched an incremental deployment plan. Our implementation results show that such a system can be successfully deployed without significant influence on standard client-server connection. Additionally, we showed that PoliCert is interoperable alongside the current TLS infrastructure and can be deployed using well-known tools without breaking legacy software or protocols.

However, several challenges still remain. Global synchronization among logs is required for some actions, however infrequent they may be. In future work we plan to investigate whether or not this requirement can be relaxed while maintaining efficiency. We also plan to explore other parameters that can be specified in SCPs, and what benefits can be realized through more detailed specifications of certificate policy. However, through PoliCert we have provided an infrastructure which provides domains with more control over the security of their own TLS connections, and provided initial steps towards improving today's TLS ecosystem.

Acknowledgments

This work was supported by a Google gift. We would like to thank Lorenzo Baesso and Lin Chen for programming assistance, and Raphael Reischuk, Ralf Sasse for providing us with valuable feedback in finalizing the paper.

References

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

[2] Devdatta Akhawe and Adrienne Porter Felt. Alice in warningland: A large-scale field study of browser security warning effectiveness. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 257–272, Berkeley, CA, USA, 2013. USENIX Association.

[3] Yousef J Al-Houmaily and George Samaras. Two-phase commit. In *Encyclopedia of Database Systems*, pages 3204–3209. Springer, 2009.

[4] David Basin, Cas Cremers, Tiffany Hyun-Jin Kim, Adrian Perrig, Ralf Sasse, and Pawel Szalachowski. ARPKI: Attack Resilient Public-key Infrastructure. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, November 2014.

[5] Daniel J. Bernstein, Anna Krasnova, and Tanja Lange. Elligator: Elliptic-curve points indistinguishable from uniform random strings. *IACR Cryptology ePrint Archive*, 2013:325, 2013.

[6] S. Blake-Wilson, M. Nystrom, D. Hopwood, J. Mikkelsen, and T. Wright. Transport Layer Security (TLS) Extensions. RFC 4366 (Proposed Standard), April 2006. Obsoleted by RFC 5246.

[7] Joppe W. Bos, Craig Costello, Patrick Longa, and Michael Naehrig. Selecting elliptic curves for cryptography: An efficiency and security analysis. *IACR Cryptology ePrint Archive*, 2014:130, 2014.

[8] Jeremy Clark and Paul C. van Oorschot. Sok: Ssl and https: Revisiting past challenges and evaluating certificate trust model enhancements. In *IEEE Symposium on Security and Privacy*, pages 511–525. IEEE Computer Society, 2013.

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

[10] Scott A. Crosby and Dan S. Wallach. Efficient data structures for tamper-evident logging. In *USENIX Security Symposium*, pages 317–334. USENIX Association, 2009.

[11] II Ecrypt. Ecrypt ii yearly report on algorithms and keysizes (2011-2012). *Available on http://www. ecrypt. eu. org*, 2012.

[12] Electronic Frontier Foundation. SSL Observatory. `https://www.eff.org/observatory`.

[13] Chris Evans and Chris Palmer. Public key pinning extension for HTTP. November 2011.

[14] Adrienne Porter Felt, Robert W. Reeder, Hazim Almuhimedi, and Sunny Consolvo. Experimenting at scale with google chrome's ssl warning. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, pages 2667–2670, New York, NY, USA, 2014. ACM.

[15] Alexandra C Grant. Search for trust: An analysis and comparison of CA system alternatives and enhancements. 2012.

[16] Phillip Hallam-Baker and Rob Stradling. DNS certification authority authorization (CAA) resource record. January 2013.

[17] ISO/IEC. *ISO/IEC JTC 1/SC 27. Standing Document 12 – Assessment of cryptographic algorithms and key lengths*. ISO/IEC, 2012.

[18] Emilia Kasper. Fast elliptic curve cryptography in openssl. In George Danezis, Sven Dietrich, and Kazue Sako, editors, *Financial Cryptography Workshops*, volume 7126 of *Lecture Notes in Computer Science*, pages 27–39. Springer, 2011.

[19] James Kasten, Eric Wustrow, and J Alex Halderman. Cage: Taming certificate authorities by inferring restricted scopes.

[20] Tiffany Hyun-Jin Kim, Lin-Shung Huang, Adrian Perrig, Collin Jackson, and Virgil Gligor. Accountable Key Infrastructure (AKI): A Proposal for a Public-Key Validation Infrastructure. In *Proceedings of the International World Wide Web Conference (WWW)*, May 2013.

[21] Qualys SSL Lab. Ssl server test. `https://www.ssllabs.com/ssltest/index.html`, 2014.

[22] Adam Langley. Revocation checking and Chrome's CRL. `https://www.imperialviolet.org/2012/02/05/crlsets.html`, February 2012.

[23] Ben Laurie and Emilia Kasper. Revocation transparency. *Google Research, September*, 2012.

[24] Ben Laurie, Adam Langley, and E Kasper. Certificate transparency. *Available: ietf. org-Certificate Transparency (06.01. 2013)*, 2013.

[25] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO '87: A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, pages 369–378, London, UK, 1988. Springer-Verlag.

[26] Mozilla. Revocation plan (draft). `https://wiki.mozilla.org/CA:RevocationPlan`, 2014.

[27] Mark D. Ryan. Enhanced certificate transparency (how johnny could encrypt). Cryptology ePrint Archive, Report 2013/595, 2013. `http://eprint.iacr.org/`.

[28] Christopher Soghoian and Sid Stamm. Certified lies: Detecting and defeating government interception attacks against ssl (short paper). In *Financial Cryptography and Data Security*, pages 250–259. Springer, 2012.

[29] Niraj Tolia, David G. Andersen, and Mahadev Satyanarayanan. Quantifying interactive user experience on thin clients. *IEEE Computer*, 39(3):46–52, 2006.

[30] H. Tschofenig and E. Lear. Evolving the web public key infrastructure. `http://tools.ietf.org/id/draft-tschofenig-iab-webpki-evolution-01.html`, 2013.

[31] VeriSign Inc. The domain name industry brief. `https://www.verisigninc.com/assets/domain-name-report-april2014.pdf`, April 2014.

[32] Dan Wendlandt, David G. Andersen, and Adrian Perrig. Perspectives: Improving ssh-style host authentication with multi-path probing. In Rebecca Isaacs and Yuanyuan Zhou, editors, *USENIX Annual Technical Conference*, pages 321–334. USENIX Association, 2008.