Institut für Software & Systems Engineering
Universitätsstraße 6a D-86135 Augsburg

Certificate-based Resource Alteration Prevention using a Public Key Infrastructure

Philip Lenzen

Masterarbeit im Elitestudiengang Software Engineering

SOFTWARE ENGINEERING
Elite Graduate Program

Institut für Software & Systems Engineering
Universitätsstraße 6a D-86135 Augsburg

Certificate-based Resource Alteration Prevention using a Public Key Infrastructure

Autor:	Philip Lenzen
Beginn der Arbeit:	12. November 2018
Ende der Arbeit:	13. Mai 2019
Erstgutachter:	Prof. Dr. Bernhard Bauer
Zweitgutachter:	Prof. Dr. Alexander Knapp
Betreuer:	Robert Freudenreich

SOFTWARE ENGINEERING
Elite Graduate Program

ERKLÄRUNG

Hiermit versichere ich, dass ich diese Masterarbeit selbständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Augsburg, den 14. Januar 2019 Philip Lenzen

Acknowledgments

This thesis was written in cooperation with Secomba GmbH, and my thanks go to Robert Freudenreich for supervising it. I would also like to thank Prof. Dr. Bernhard Bauer, who agreed to be the first examiner of my thesis, and the second examiner Prof. Dr. Alexander Knapp. Furthermore, I thank Boris Schlosser for the critical discussions and comments, as well as Nina Jameson, Johannes Jungbluth, Ralph Reithmeier and especially Dominik Klumpp for proofreading and feedback. Last but not least, I would like to express my deepest gratitude to Carola Gajek, who never hesitated to spend hours discussing, and who put up with me throughout the entire period of work.

Abstract

The shift from purely static server-side websites to rich web services in the modern World Wide Web entailed so-called single page applications. These are JavaScript programs that run mostly server independent in the client's browser and dynamically change the visible elements, requesting additional resources only when they are needed. Unfortunately, security issues were only of secondary importance in this evolution, leading to numerous client-side attacks nowadays to steal sensitive user data such as credit card data or even passwords. In addition to well-known cross-channel attacks such as cross-site scripting or cross-side request forgery, this also includes resource tampering, which was originally more ascribed to server-side attacks, but which explicitly enables client-side attacks through the distribution of script resources by third party content distribution networks. While existing work deals basically only with attacks by malicious third parties, we assume in this thesis that the publisher of the application can be in cahoots with the adversary. The resulting attack, targeting a specific or small group of clients, is called a *resource alteration attack*, as the application files can be legitimately modified to reveal user information.

In this work, we present a novel approach based on a public key infrastructure to prevent such attacks, even if multiple parties within our system have been compromised. Hash values of application resources are embedded in certificates, which are validated by certificate authorities. Thereafter, they are stored on one or more synchronizing log servers in an authenticated data structure, the Merkle Tree [33]. Such a tree is able to generate unforgeable proofs regarding its content and consistency of insertions, whereby the existence of a certificate can be proven beyond doubt and verified by the client. We modify and extend an existing protocol [34] that allows certificate authorities to monitor modification and query processes, signing the proof responses on approval. This allows any malicious party to be identified and held accountable in the event of inconsistencies. We also present an efficient monitoring procedure that does not require a Merkle Tree to be built and maintained locally, so that instead of only dedicated monitors, the client can also ensure global consistency of multiple log servers. An analysis of our proposal shows that the level of security scales in the number of certificate authorities involved, and in the case of a secure connection between publisher and client, both the former and at least one log server must be compromised in addition to the authorities for a successful attack.

Furthermore, we encourage security experts to publish their analysis results for an application in our system with a certificate as well. Since we expect casual users to have not taken far-reaching security precautions against common client-side attacks, experts can additionally specify assertions in the analysis certificates to provide additional protection and confidence. These are then verified by our browser extension, which already queries certificates and verifies associated proofs.

Contents

1 Client-Side Security in Single Page Applications

The modern World Wide Web has come a long way: while static websites dominated in its early days, server-side programming languages such as PHP led to a shift towards interactive server-side applications. At the latest since the birth of Web 2.0 in 2003, more and more parts of the application logic have shifted to the client side – supported by JavaScript, which is still the prevailing programming language for client side web applications today.

Since its peak in 2009 with 98.3%, the percentage of websites available via the Internet Archive's Wayback Machine that use JavaScript has not changed significantly [1]. However, in addition to the complexity and number of statements, the number of third-party scripts from remote origins involved has increased: in 2016, each site used external scripts from an average of 12 different domains [1]. Besides analytics or advertising libraries, a common representative is jQuery [2], whose presence grew after its introduction in 2006 to over 65% in 2011 and which was used by 77.9% of the top 1 million websites at the end of 2018 [3]: This provides, among other things, functionality for page navigation, asynchronous HTTP requests and direct manipulation of the *Document Object Model* (DOM) tree – and may therefore be considered one of the reasons for the rise of so-called *Single Page Applications* (SPA).

Rather than downloading entirely new pages from the server after a user interaction or navigation command, they dynamically update the content of the current page, while additional resources are reloaded and embedded into the page only when needed [4]. This allows a fluid and fast transition between different pages and views. Furthermore, they also enable richer applications on the client side by moving traditional desktop functionality to the browser, since resources as well as a complex application state can be effectively cached locally. This implies at least limited offline functionality and the opportunity to reuse backend code for mobile applications. Therefore concerns and corporations such as Google, Facebook and Twitter have realized their products as SPAs, but also smaller companies chose this design pattern for complex applications. To implement these, they can draw on powerful frameworks such as React, AngularJS or Vue.js [5].

However, SPAs also have some drawbacks compared to conventional Multi Page Applications (MPA) [4], [5]: besides the more difficult optimization for search engines and analytics, their initial loading takes much longer due to the resources that have to be downloaded to start the application. Although subsequent page loads are usually faster than MPAs, implementation bugs can cause a drop in performance. Even worse, their strong dependence on the uncompiled programming language JavaScript (which therefore cannot be deactivated in the browser, insofar as the application should be executable) opens the door to attackers. But it is not JavaScript alone that has led to new web-specific vulnerabilities. These have mainly arisen from the fact that security issues have been grossly neglected during the evolution of the web, which is why some experts call "Web

security" an oxymoron [1].

While there is an overwhelming amount of attacks on web applications in general and on SPAs in particular, we would like to focus on those that aim at unauthorized access to sensitive user data. Banking transactions are carried out online, private files and company secrets are stored in the cloud, correspondence is increasingly based on e-mails or chats – especially the fact that more and more critical services are being moved to the web provides tempting incentives for attackers: credit card numbers, company secrets, private addresses or simply passwords to access and exploit services without any further effort offer a lucrative prospect. Therefore, a detailed investigation of such attacks and the development of effective security countermeasures are necessary.

1.1 Client-Side Attacks and Mitigation Techniques

In general, attacks on web applications can be categorized as server-side or client-side. While the former aim at gaining access to the server and thus the backend of the application, the latter intend to control or manipulate the JavaScript code in the user's browser. Of course, both can be used to steal user data. But since attacks on the server have been dealt with extensively in literature [6], [7], we limit ourselves to only consider client-side attacks that lead to violation of privacy and confidentiality.

Cross-Channel Attacks One of the most common and serious client-side attacks is *Cross-Site Scripting* (XSS) [8]: Due to insufficient or incorrect validation of user input, an attacker can force the execution of malicious scripts injected into a web application through user input forms. It is usually separated into three categories: Reflected, Stored and DOM-based XSS. The first two target the server, which executes the malicious code either transiently or persistently and reveals information, while the latter refers to the execution in the client's browser by DOM manipulation. This may result in cookie stealing to impersonate the user to the server by hijacking the session, in phishing of credentials via manipulated DOM input fields or even key logging, or in the exposure of private information displayed on the website [9], [10]. Note that it is also possible to execute JavaScript code received from other sources during runtime by commands such as `eval` without involving the DOM, which can therefore be considered a fourth category of Client-Side XSS [11]. Cross-Site Scripting attacks have already been known for nearly 20 years [1], and due to their importance there are a number of proposals for detection and prevention. In general, they use techniques such as static and dynamic analysis, modeling, policies or programming guidelines – however, this is a widely explored research area, and therefore we just refer to [10] for a detailed overview.

Another attack that targets the user, but has received less attention in the literature than XSS, is *Cross-Site Request Forgery* (CSRF) [12]: a malicious website forces the browser to send HTTP requests in the name of the user – if they are already authenticated on a website, arbitrary actions such as transactions can be performed and information available in the account can be accessed. In contrast to XSS, the malicious code runs on third-party sites, which means that CSRF attacks are not dependent on JavaScript. This implies that

sites which are vulnerable to XSS automatically allow CSRF – however, even effective protection against XSS is no guarantee of being immune to CSRF [13]. Nevertheless, such attacks usually exploit poorly secured authentication mechanisms, such as Basic Authentication [14] or session cookies, which is why simple means such as secret tokens or development guidelines can often provide a remedy [12], [13]. More sophisticated solutions use, for example, an HTTP proxy on the client side to examine outgoing requests [15], or a browser extension to prohibit non-compliant messages [13].

We end the overview of cross-channel scripting attacks with *Cross Frame Scripting* (XFS) [16]: If the user has been tricked onto a malicious website, an attacker can integrate the actual web application using an HTML `iframe` element – by listening to events leaked by the browser, for example, key logging is possible. Furthermore, such attacks may enable XSS or CSRF, but must be clearly distinguished from them. In contrast to CSRF, the attack is executed inside the visited website, but does not force a script execution as during XSS attacks. Rather, XFS can be compared to phishing or spoofing schemes, but it does not install a vulnerability in the copied target web site, it exploits one of the browser. Developers can easily prevent such attacks through Frame Busting, i.e. prohibiting the inclusion of their website in a frame [17].

Resource Tampering Although there are more client-side attacks like web cache poisoning or response splitting and associated mitigation techniques [18], we would just like to introduce another attack traditionally considered to be server-side: *Defacement* [19] refers to the malicious modification of page content by an intruder, i.e. the insertion of provocative or offensive data. To ensure the integrity of website resources, watermarks have been suggested [20], [21]. Based on message authentication methods or simple hash algorithms, watermarks related to the page content are generated and embedded in the website through upper- and lowercase letters of HTML tags. Their locations are determined by secret keys, which in turn serve to extract a watermark from a page – if this does not correspond to the originally generated one, the attack is discovered. Other suggestions achieve this by periodically matching a checksum of the website with an already stored one [19], [22], which can also be handled by a browser extension on client-side, or by log file analysis [23]. As an alternative approach, [24] makes use of genetic programming: Using observations of the target page and a sample set of attacks during its learning phase, the generated algorithm does not rely on domain-specific knowledge, but is still able to detect suspicious modifications and raise alarms.

The reason we mention this classic server intrusion attack pattern is the trend towards rich applications whose script resources are retrieved from multiple origins other than the server. By defacing them, an attacker can bypass the mentioned security mechanisms against cross-channel attacks, since a properly integrated but maliciously modified application script is not cross-channel despite having the same effect on client-side. This means that the attacker can "legitimately" access user information, as this seems to be part of the application – making cross-channel attacks obsolete. Since such an attack is not a classic defacement, we will use the term *Tampering* for altering resources of arbitrary type in the following. The mentioned defacement mitigation techniques have only limited

effect in this case, since they are either designed for purely static content located on the application server, or are based on previous observations, which can be compromised from the beginning. Therefore, various approaches to mitigate tampering attacks by malicious *Content Distribution Networks* (CDN) or third parties in general have been proposed:

In [25], mutation events are used to enforce security policies before scripts are executed. The policies specified by the application publisher can, for example, restrict script access or only allow scripts white-listed by their hash value. *Firecoral* [26] provides a browser extension that allows the client to participate in a peer-to-peer CDN to share resources with other clients and thereby validate them. A trusted signing component validates the authenticity of the resource checksum a client claims to have received from the server. The signed checksum is then deposited with the peer discovery component so that other clients can subsequently request the same resource from the client. *Stickler* [27], on the other hand, does not require an extension, but uses a bootloader script embedded in the HTML page of the application. This contains functionality to load the required application resources and verify if their checksums match those stored in a separate manifest file. Alternatively, the verification can only check whether the resources have been signed by the publisher. The development platform *Mylar* [28] protects data from attackers with full access to the servers involved by encrypting it on client-side. The developers of an application can determine which data can be encrypted and who can access it. Further, Mylar ensures their integrity through a browser extension that verifies checksums of resources. Finally, *Subresource Integrity* [29] allows the publisher to specify checksums of a resource in the HTML page of the application, which is verified when loaded by the browser. Furthermore, the check can be enforced by specifying an appropriate policy. Although this feature is already available in most browsers, its status is still "recommendation" [30].

Unfortunately, all these proposals assume that the publisher's server is not compromised or at least the developers are not malicious. Only the peer-to-peer solution Firecoral could continue to ensure integrity in this scenario – but demands that other trusted parties the client relies on can never be compromised by an attacker.

1.2 Adversary Model

As it already became clear at the end of the last section, we assume in this work that there is a powerful adversary who attacks the client by manipulating the resources of a single page application. However, they are not only able to infiltrate CDNs, but also to compromise or even convince the publisher of the application to cooperate with them. Strictly speaking, it follows that it is no longer a tampering attack, since the application owner is providing a new version that is vulnerable – so we call this attack a *Resource Alteration Attack* instead. In summary, the adversary has full control over:

- the application server and the resources on it
- CDNs that provide additional resources for the website
- the uncompressed source code and the build process
- the network connection and communication
- an arbitrary number of members that could be relevant for security systems

The first two points allow all resources to be modified at will, no matter where they are stored. The third point allows these changes to be legitimately made, i.e. without inconsistencies in checksums or log files. In addition, the attacker is able to monitor arbitrary connections, read messages, modify them and, if necessary, repeat or insert them. But the last point has the greatest influence: We allow the adversary to control not only servers that are involved in the delivery of the application, but also those network members that are supposed to ensure integrity and thus protection against resource alteration. Note that these do not necessarily have to be objects: The adversary can also convince developers and organizations to participate in the attack. However, we demand that the attacker can neither control the user nor the computer and browser they use, and thus still assume the necessity of web-based attacks.

Scenario We assume in this thesis that the adversary does not attack all users of an application at the same time – if they did, there would be no way to expose resource manipulation as a malicious action. Instead, it would have to be interpreted as a simple software update, except that the new version has vulnerabilities. Depending on the type, these can be detected by the mitigation techniques described above or by security experts or tools that analyze the application using a wide variety of methods [31], [32]. Therefore, we expect the resource alteration attack to target a specific or a small group of users, and only they will receive the malicious version of the single page application.

Furthermore, we need to make restrictions in order to have any chance at all against the adversary. Otherwise, in the worst case, they would be able to compromise the whole web, if all members could contribute to our protection. Note that we are explicitly talking about network members rather than just servers, as other clients in decentralized or distributed systems can also be in cahoots with the attacker. For this reason, we limit the attacker so that they can control some, but not all of the members – however, we do not determine their exact type and number.

Finally, we demand that all application resources required for execution in the browser are static files that are not generated specifically for a client upon request. This makes the website itself explicitly not static – during runtime, its content can still change and contain client-specific information dynamically queried from the server. However, it excludes techniques such as server-side rendering, where the page is loaded with the necessary information on the server before delivery. But since this requires that the application server, and thus the adversary, already has access to sensitive user data, purely static resources do not represent a significant restriction.

We want to emphasize that all proposals against tampering from the previous section fail in this scenario: Policies can't work with reasonably sophisticated and intentional vulnerabilities without significantly restricting the functionality of other, honest websites. Subresource Integrity and Stickler only protect against third parties, not against the publisher who provides the necessary checksums. In Mylar, developers determine what information is encrypted, which checksums are used for integrity verification, and who has access to data – but they can easily be convinced to support the attack based on our

adversary model. Even Firecoral falls victim to the power of the adversary: once the peer discovery service and the publisher are in cahoots, they can provide the client and the signing component with tampered resources. These obviously cannot be obtained from any other client and are henceforth considered valid new resources. Since the Peer Discovery component keeps the client's message about the new resource visible only to the client and the signing component, the attack is not noticeable.

1.3 Contribution

In this master's thesis we present a new scalable approach to prevent targeted resource alteration attacks even in case of multiple compromised entities within our system. For this purpose, we enable the publisher of a security-critical single page application to wrap hash values of resources that serve as fingerprints into certificates and register them in a public key infrastructure. This consists of log servers, which store the certificates in authenticated data structures, the Merkle Trees [33], and thus can generate unforgeable proofs about the existence of a certificate as well as consistency between several versions of the tree. These are then verified by certificate authorities who supervise the registration process and forwarded to both the publisher and the client. After registration, the certificate is publicly visible and cannot be removed from the data structure, preventing hidden attacks on a client. In addition, security experts can publish analysis results in the same way, informing casual users of any vulnerabilities in websites, whether or not they have been tampered with. In detail, the most notable features of our system are:

Multi-Resistant Integrity Protection Collision-resistant hash values, which are stored in certificates that are made publicly visible, are used to prove the integrity of resources beyond any doubt. We modify and extend protocols of an existing public key infrastructure [34], which ensure that all parties involved monitor each other during both registration and query, so that the client can request verifiable proofs for existing certificates. Moreover, efficient consistency checks can be performed by both the client and dedicated monitors. A security analysis shows that a scalable number of certificate authorities (all those involved in the process) must be compromised in order to deliver a bogus certificate to the client. If the connection between application server and client is secure and the log servers are immune to Denial of Service attacks, the publisher and all participating log servers must also be malicious.

Authenticity and Accountability Public key cryptography and digital signatures make it possible to ensure authenticity of a certificate and each action of every participant. Since with every modification of stored certificates the actions themselves, but also the approval by supervising parties must be authenticated by signature, the malicious entities can be identified and held accountable in case of inconsistencies (which most likely indicate an attack). These in turn become visible at the latest through the append-only property of the Merkle Tree logs, which can be verified by all parties.

Distribution of Security Analysis Results Although our system guarantees that all users receive the same application resources, it provides no information about vulnerabilities to traditional client-side attacks in those resources. But since we do not assume that casual users will review the resources or even use analysis tools to identify threats, we offer security analysts a way to register the results of their analysis and optionally resulting assertions for an application by using special certificates – these can then be dynamically verified by the client. Because they follow the same processes as the resource certificates, they cannot be inserted or modified unnoticed.

Efficient Verification and Deployability We implemented a lightweight browser extension for the client that automatically downloads certificates and verifies the associated proofs and assertions. Since due to the use of Merkle Trees, the former require only hash calculations to be verified and their length remains logarithmic in the number of all registered certificates, they do not neither require high computational effort nor notable network consumption. For the other participants we provide virtualized containers, which offer the functionality intended for them – especially for developers it is thus possible to integrate our system into their existing deployment process without much configuration effort by just a single line.

1.4 Outline

We start with an introduction to the concept of public key infrastructures in Chapter 2: after a brief explanation of basic cryptographic concepts on which they are based in Section 2.1, we examine the different categories and corresponding proposals in Section 2.2. It turns out that neither authority- nor client-centric architectures would be particularly suitable for our scenario, but domain-centric concepts based on log servers are. Therefore, we compare the different approaches in this category with respect to several features, including security, at the end of this section.

Since most log-based public key infrastructures make use of the authenticated data structure Merkle Tree, we introduce it in Section 2.3: Following on from basic terminology, we show how it allows to prove that an entry is actually in the tree. Furthermore, two different variants of Merkle Trees are presented, each with its own proof capabilities, which together form the data structure that we build on later. Finally, we look at applications of this data structure in the literature and productive systems.

Subsequently, we start with the description of our system in Chapter 3 by explaining the basic design decisions and defining the requirements for our infrastructure in Section 3.1. We not only consider the necessary security criteria, but also quality attributes that have to be met on order to gain acceptance by users and developers. From now on, we focus on the different participants in Section 3.2 and the individual data structures, i.e. the different certificates and their attributes, in Section 3.3. The interaction between participants as well as the creation and usage of certificates is defined by the protocol in Section 3.4: we begin by describing the modifications to the public key infrastructure that inspired it, followed by the protocol steps of generating certificates, modifying Merkle Trees,

and verifying resources and proofs. Afterwards, an extension of the original protocol is presented, that is monitoring without keeping a copy of the data structure. The subsequent security analysis in Section 3.5, in which we consider both external and internal attacks, as well as combinations thereof, reveals the resilience of our architecture. After briefly discussing the implementation of the prototype in Section 3.6, we conclude the chapter with a discussion of, for example, improvements or economic incentives in Section 3.7.

2 Public Key Infrastructure Architectures

A *public key infrastructure* (PKI) [35] describes several roles, policies and services, which allow users to securely exchange private data inside an insecure public network like the Internet. Every participant owns a digital certificate for identification, which can be obtained through a trusted *certificate authority* (CA). An additional *registration authority* (RA) ensures validity and correctness of a registration request and authenticates the requesting entity. Afterwards, a generated certificate is stored in one or more *secure directories* and made publicly available to other parties through a *certificate management system*. The certificate itself contains a unique key used for authentication and encryption based on *public key cryptography*: Every participant owns a cryptographic key pair consisting of a secret *private key* only they know and the *public key* included in the certificate. If this pair satisfies the property that a message encrypted with one of them (which we call *cipher*) can only be decrypted by the other, it can be used for secure communication – by encrypting a message with the public key of the recipient, a receiver cannot decrypt it unless they have the corresponding private key. Further, it allows to create *digital signatures* for a message by encrypting it with the private key and sending the resulting cipher (or, more precisely, the *signature*) together with the original message. Then, anyone can authenticate the sender by decrypting its signature with the corresponding public key obtained through the certificate and comparing the result with the message. A formal description is given in section 2.1, but for now, this explanation suffices to illustrate five fundamental security requirements a PKI satisfies:

- *Non-Repudiation*: Since a PKI requires each message to be digitally signed, the sender can neither deny the existence nor the origin of a message.
- *Privacy*: A PKI enables private communication based on public key encryption – any party that is not an intended receiver cannot decrypt and read a message.
- *Integrity*: Besides its importance for non-repudiation, a digital signature proves that the message has not been tampered with if it is equal to the message when decrypted.
- *Accountability*: A participant can be held accountable for their messages, as their identity can be verified through digital signatures in combination with certificates.
- *Trust*: A PKI creates trust between all parties involved based on the security properties above – however, each participant has to trust the issuing CA.

Regarding secure web-applications, the *Transport Layer Security* (TLS) ecosystem protects the majority of financial and commercial transactions [34] as well as the communication between web server and client as part of the HTTPS protocol [36]. It employs a PKI for X.509 certificates (*PKIX*) [37], the standardized format for a TLS certificate, using a repository for valid certificates and one for *certificate revocation lists* (CRL). The architecture is illustrated by Figure 2.1: Whenever a domain owner wants a X.509 certificate to be approved, they contact a CA either directly or through a RA. On approval, the CA signs

this certificate and publishes it on the repository (the latter may alternatively be performed by a RA). This workflow also applies to certificate revocation, except that a CA updates the list of revoked certificates and publishes it on the CRL repository. To establish trust between CAs, they build a hierarchical CA certificate chain using *cross-certification* (i.e. signing the public key of another CA), which allows to verify signatures of any participating CA by using the public key in the topmost certificate referred to as *root certificate*.

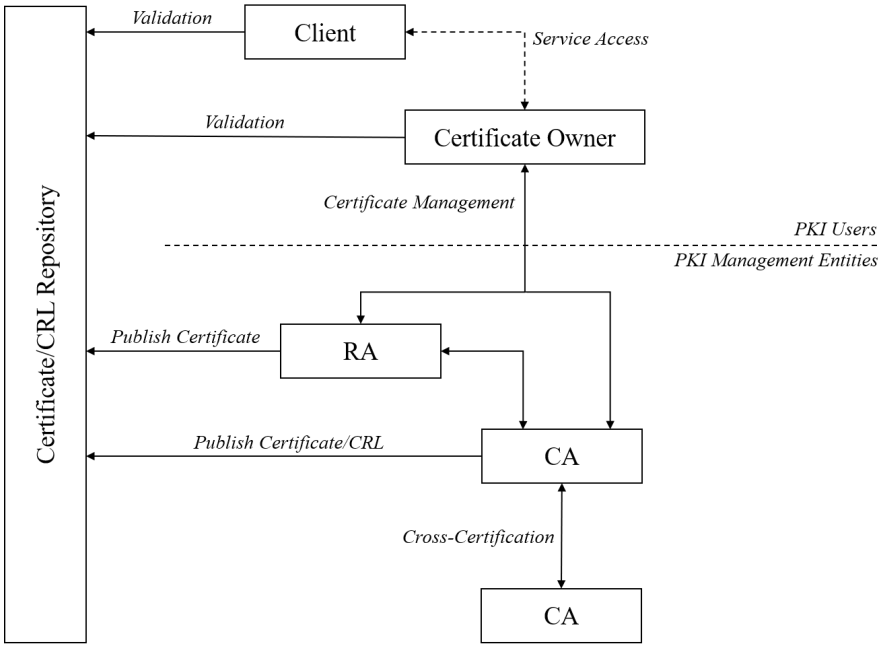


Figure 2.1: *Entities and communication in the public key infrastructure for X.509 certificates [38].*

When accessing a website or executing a transaction, the communication is encrypted using the public key of the target server, which is included in its certificate. The CA signature should prove the correctness of the public key and identity of its owner, and is verified using a root certificate built into the browser (e.g., Mozilla Firefox has about 100 in its database [39]). Unfortunately, a single compromised authority can issue certificates containing fake keys for any domain which are still accepted by the browser [40]–[44]. In fact, this can result in *Man-in-the-Middle* (MitM) attacks, i.e. a third party can read and possibly alter the communication while remaining undetected [45].

For this reason, numerous new architectures have been proposed to mitigate the impact of untrustworthy authorities – they can be roughly classified as *CA-*, *client-* or *domain-centric* [34], and we take a closer look at the notable work in section 2.2. Since it turns out that domain-centric approaches using log servers are suited best for our purposes, section 2.3 introduces the data structure *Merkle Tree*, on which most of them depend.

2.1 Cryptographic Preliminaries

Although cryptography has long been recognized as the art of writing or solving codes (as defined by the Concise Oxford Dictionary 2006), modern cryptography is essentially the science of techniques for securing digital information, distributed computations and transactions [46]. Fortunately, this shift led to new methods accompanied by formal definitions and security proofs, which are not only intended for encrypting messages. Since PKIs security characteristics are based on cryptographic operations, this section briefly introduces their fundamental building blocks.

2.1.1 Hash Functions

A basic method for ensuring integrity in networks and especially for digital signatures are hash functions. Informally, they compress an (possibly infinite) input string into a string of fixed length which can effectively serve as a fingerprint for the input.

Consider two alphabets M and D. A *hash function* $H : M^* \to D^k$ compresses a *message* $m \in M^*$ of arbitrary length into a *digest* $d \in D^k$ of fixed size k deterministically and in polynomial time. Members of M^* are also commonly called *preimages*, while a member of D^k is named *hash value* or simply *hash*. Since D^k is finite but M^* isn't, H cannot be injective and applying H to two different $m, m' \in M^*$ may result in a *collision* $H(m) = H(m')$. Thus, in context of cryptography, the following security requirements are commonly considered [46]:

1. *Preimage resistance*: Given a digest $d \in D^k$, the probability of "guessing" a value $m \in M^*$ such that $m = H^{-1}(d)$ is negligible for every probabilistic polynomial-time inverting algorithm (which we will further denote as *computationally hard* [47] to find)
2. *Second preimage resistance*: Given a fixed $m \in M^*$, it is computationally hard to find $m' \in M^*$ such that $m \neq m'$ and $H(m') = H(m)$
3. *Collision Resistance*: It is computationally hard to find any distinct $m, m' \in M^*$ such that $H(m) = H(m')$

Note that collision-resistance implies second preimage resistance, and we obviously want to use a collision-resistant hash function in our system. Ideally, it behaves like a *random oracle* [48]: on the same input, the output is repeated, but for any new message a random string is returned (which implies that even a small change in the message is likely to result in an entirely different digest). Of course, such an oracle does not exist – yet, hash functions should not be distinguishable from it. However, popular representatives like SHA-256 suffer from a vulnerability called *length-extension attack* [49], which can be exploited if an attacker knows the length of a message m and its digest $H(m)$: Due to their construction, an attacker can choose a message m' depending on the length of m and compute $H(m||m')$ without knowledge of m, where $||$ denotes the concatenation operation. Therefore, hash functions vulnerable to length extension are clearly distinguishable from a random oracle, even though this has no impact on their resistance.

2.1.2 Digital Signatures

As described above, a *digital signature* [50] guarantees integrity, authenticity and non-repudiation of a message [51]. Based on public key cryptography, it uses a secret private key and a public key accessible to the recipient, who is therefore able to verify the mentioned characteristics of the signature. In general, it is described by a mathematical scheme given by three probabilistic polynomial-time algorithms $(\texttt{Gen}, \texttt{Sign}, \texttt{Verify})$ which satisfy the following [46]:

1. `Gen` generates two keys pk and sk (the *public* and *private key*, respectively) for a security parameter λ^n
2. Given a message m and a private key sk, `Sign` returns a signature σ
3. The deterministic algorithm `Verify` computes a bit VALID or INVALID using a public key pk, a message m and a signature σ

As an example, consider a signature scheme based on RSA [52]: $\texttt{Gen}_{RSA}$ returns a figure $N = p \cdot q$ with p, q prime as well as two integers e, d with $ed = 1 \bmod \phi(N)$. Then, the private key is $\langle N, d\rangle$ and the public key is $\langle N, e\rangle$. Using such a private key and a message $m \in \mathbb{Z}_N^*$, $\texttt{Sign}_{RSA}$ computes $\sigma := m^d \bmod N$. Finally, $\texttt{Verify}_{RSA}$ returns VALID if and only if $m \stackrel{?}{=} \sigma^e \bmod N$ for a public key $\langle N, e\rangle$, a signature s and a message m with $\sigma, m \in \mathbb{Z}_N^*$.

However, this signature scheme is insecure: Given two messages $m_1, m_2 \in \mathbb{Z}_N^*$, and corresponding signatures σ_1, σ_2, then $\sigma := \sigma_1 \cdot \sigma_2$ is a valid signature for $m = m_1 \cdot m_2$ because $\sigma^e = (\sigma_1 \cdot \sigma_2)^e = (m_1^d \cdot m_2^d)^e = m_1^{ed} \cdot m_2^{ed} = m_1 \cdot m_2 = m \bmod N$. Thus, an adversary can create a *forgery*, i.e. a valid signature for the message m although they do not know the private key $\langle N, d\rangle$. Therefore, we demand that a secure digital signature is *existentially unforgeable under an adaptive chosen-message attack* [46]: It has to be computationally hard to find a valid signature without knowledge of the private key even if the attacker can obtain valid signatures for an arbitrary number of messages of their choice. In case of RSA, this can easily be achieved: `Sign` typically hashes a message m with an at least preimage resistant hash function H before computing the signature. This ensures that it cannot be forged due to the multiplicity of RSA, since $H(m_1 \cdot m_2)$ and $H(m_1) \cdot H(m_2)$ are totally different digests. Moreover, $H(m)$ is of fixed size and usually much shorter than the arbitrary large message m, which is why most signature schemes sign the hash value instead of the plain message as this also results in a speed benefit.

In the following, we denote the encryption of a message m with a public key K by $\{m\}_K$, whereas a message signed with a private key K^{-1} (i.e. m together with its signature) is written as $\{m\}_{K^{-1}}$.

2.2 Enhanced PKI Architectures

While public key infrastructure architectures have been developed for various fields [53]–[57], we focus on proposals designed specifically for X.509 certificates [37], the standardized format for TLS. Due to the enormous impact of the weaknesses of PKIX, the work in this

area covers almost all relevant concepts regarding PKIs. As stated in [34], the proposed architectures can be divided into three categories: In *CA-centric* PKIs, authorities control whether a certificate is valid or not. By contrast, *client-centric* approaches enable clients to actively select and maintain mechanisms for certificate validation by themselves. *Domain-centric* architectures share the permission to define which certificates are valid between CA and domain owner, thereby allowing the latter to actively control (and protect) their domain certificates. Figure 2.2 provides an overview of this landscape we survey in the following sections.

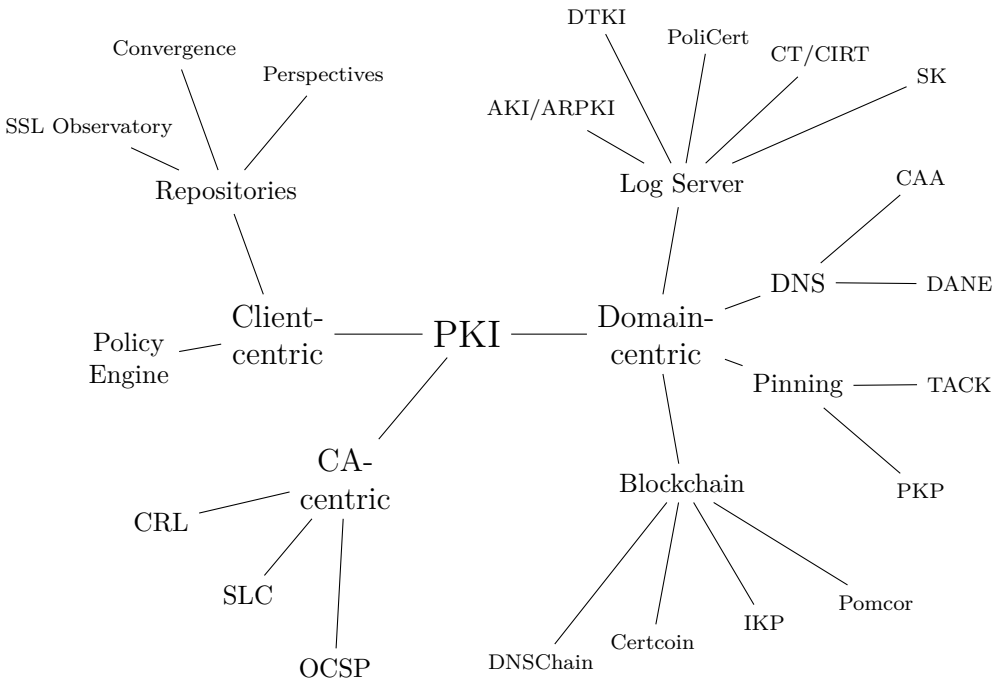


Figure 2.2: *Classification of enhanced PKI architectures.*

2.2.1 CA-Centric Architectures

We have already encountered a CA-centric enhancement: the CRLs in PKIX extend the standard PKI model by providing an authority-managed source for looking up revocations. Unfortunately, clients have to download them continuously to make sure that a certificate is not revoked – as CRLs can get quite large, this is highly impractical during connection setup.

The *Online Certificate Status Protocol* (OCSP) [58] is designed to resolve these issues. This standard requires CAs to run external OCSP servers which are always up-to-date with the certificate repository of the CA. Therefore, clients are not required to download CRLs but can query these servers for a particular certificate. Although some successive extensions [59], [60] have been proposed to allow secure status checks in real-time, implementations still suffer from performance and security flaws [61].

By restricting the time-to-live of a certificate to several hours up to a few days, *short-lived*

certificates (SLC) [62] eliminate the need for revocation and therefore make CRLs as well as OCSP obsolete. However, this results in quite a long attack window if a fake certificate is accepted.

In conclusion, SLC and other CA-centric architectures do explicitly not protect against certificates issued by a compromised CA, making this category unsuitable for our scenario, as a single party can always be compromised by the adversary.

2.2.2 Client-Centric Architectures

A naive approach to client empowerment is to let the client simply define what trust means to them. Thus, the *Policy Engine* [63] offers support in designing custom security policies (e.g. "certificate keys shorter than n bits are untrustworthy"), which can also include historical observations. The engine transforms these rules into predicate functions for certificate validation.

However, this system is purely local – in contrast, systems like *Perspectives* [64] and *Convergence* [65] use public repositories called notaries which connect to the service a client queries and log the certificate it offers. Based on a policy selected by the client, a particular voting rule is applied between several notaries to validate the result. To detect malicious participants, the notaries share their databases among themselves and sign them in case of consistency, so that a client can cross-validate the reply.

The largest repository was created by the Electronic Frontier Foundation: They collected all X.509 certificates visible on the Internet (IPv4) and stored them in the *SSL Observatory* [66] for research purposes. Unfortunately, the data is not updated and cannot be queried online.

Finally, Perspectives and Convergence are powerful architectures against attacks even if a large proportion of notaries is compromised (and are, in fact, relatively similar to some proposals in the next section). Yet, we would like to demand even more: Consider a MitM attack on the service itself, such that the attacker behaves as a proxy for incoming requests. To both clients and notaries, the attacker could present a consistent view regarding their fake certificate, and the only party that would be able to detect the attack (i.e. the publisher) is not involved in the voting process. This motivates our investigation in the following category.

2.2.3 Domain-Centric Architectures

PKIs in this area are designed to reduce the inevitability of blindly trusting authorities by taking statements of the owner into account. They can be subdivided into approaches based on *pinning*, *DNS*, *blockchains* and *log servers*.

Pinning Representatives of pinning-based proposals are *Public Key Pinning* (PKP) [67] and *Trust Assertions for Certificate Keys* (TACK) [68]. The former allows servers to attach several X.509 certificate chains to a response which the client "pins" on first service access for a limited period. Alternatively, the latter defines designated TACK-keys which are processed in the same way. If on a subsequent request, a certificate occurs that is not

included in the pinned list for this service or, respectively, a TACK-key that is not equal to the pin, the client might be under attack. However, these systems do not protect on first access and also decline acceptance of legitimately changed certificates until the pin expires.

DNS *DNS Certificate Authority Authentication* (CAA) [69] allows service providers to specify which CAs should issue their certificates inside resource records served by the Domain Name System. Before accepting a certificate, the CA performs a lookup for the corresponding record and ensures that it is included therein. Correct behavior of the authority can be monitored by third parties (or, although not recommended, directly by clients).

DNS-based Authentication of Named Entities (DANE) [70] enables domain owners to specify constraints on the acceptance of certificates in DNS records. These and advise clients to allow only a particular root-authority, to accept the certificate if already a given segment of the CA certificate chain is valid, or to match against a particular certificate. Note that the last option enables the use self-signed certificates by skipping cross-validation. Unfortunately, both rely on uncompromisable DNS servers which therefore become a single point of failure.

Blockchain The core idea behind blockchain-based PKIs is to decentralize the certificate repository, so that consensus about the validity of a certificate must be reached between the parties involved in the registration process. In general, a *blockchain* [71] is a list of records called *blocks* that contain *transactions* (i.e. an actual data value) stored in a hash-based data structure which serves as digital signature over the dataset. Besides some other values, the *block header* contains this signature and a hash value of the previous block that marks the current one as its successor. Both signatures together allow to verify that a blockchain includes a particular value and each block is an extension of its tail. Since each node in a blockchain network is considered untrustworthy, several approaches exist in order to reaching consensus – for example, *Proof-of-Work* (PoW) requires each node to compute hash values of the block header using a varying *nonce* value (i.e. a number only used once) as additional header attribute. If a digest fulfills predefined requirements, the other nodes verify its correctness and append the new block to their local blockchains. The computing node gets a reward for being the first to find a proper hash.

The decentralized PKIs *DNSChain* [72] and *Certcoin* make use of the cryptocurrency *Namecoin* [73] which primarily acts as decentralized DNS but supports X.509 certificates to be additionally included. However, DNSChain introduces central servers to remove the need for clients to download the whole blockchain in order to verify a transaction, which in turn leads to attacks if these servers are compromised. Certcoin has no identity validation, which implies that an attacker can easily register certificates for a domain they do not own.

Other approaches like *IKP* [74] or the PKI developed by *Pomcor* [75] rely on CAs responsible for the prevention of identity theft as well as for certificate registration and monitors to validate their publications to the blockchain. Further, IKP combines this with authorization criteria embedded in policies a publisher defines and reactions specified by the CA which apply if a unauthorized certificate is issued. Both of them require the client

to download only the chain of block headers without the actual block data, making the verification process more efficient.

Nevertheless, all proposals suffer from problems typical for blockchains [76], [77]: First of all, most blockchains do not scale very well. Since the block size is fixed by design, it can only contain a limited number of transactions – therefore, only so many can be processed at once. Secondly, performance issues arise especially from the consensus mechanism, as all nodes have to verify and update their blockchains which may take hours to complete. Finally, blockchains are based on the assumption that an attacker cannot gain control over at least 51% of the nodes responsible for hashing. Otherwise, they would have conquered the whole network and be able to reverse and suppress transactions from other participants.

Log Server Instead of decentralized repositories, several architectures let domain owners register their certificates on public log servers to make CAs accountable for their actions. *Soverein Key* (SK) [78] requires any X.509 certificate to be cross-signed by an additional public key which is registered on append-only "timeline" servers for the respective domain. Usually, a client queries a mirror server that keeps a full copy of the timelines for the sovereign key of a domain, and declines any certificate that is not signed by it – however, this is based on trust, since a compromised mirror could simply answer with a fake key that does not exist in the timeline.

In contrast, *Certificate Transparency* (CT) [79] uses an authenticated data structure as chronological append-only log which is capable of creating cryptographic proofs regarding the inclusion of its content. Clients can then easily verify that a particular certificate is indeed included and the current log is an extension of another one observed earlier (typically, the latter is checked by dedicated monitor servers).

Yet, CT does not allow to prove that a certificate has been revoked, which is why the *Accountable Key Infrastructure* (AKI) [80] relies on a verifiable data structure called LexTree that is ordered lexicographically rather than chronologically. Certificates for a domain can be directly accessed in such a key-value store and simply removed on revocation. Unfortunately, consistency between two observed versions cannot be proven like in CT, as this requires non-modification. Therefore, AKI establishes a "checks-and-balances" concept between CAs, log servers and validators to make sure that all parties behave correctly by monitoring and reporting each other.

Inspired by this, the *Attack Resilient Public-Key Infrastructure* (ARPKI) [34] addresses weaknesses of AKI through extensive protocols for maintaining the log. Certificates are registered or revoked by creating a new entry which is synchronized between log servers. Further, a domain owner can actively select the service providers involved in the process, which then verify and cross-sign their actions for accountability. Additionally, the security property of ARPKI has been formally proven using the protocol verification tool *Tamarin Prover* [81].

Built on top of ARPKI, *PoliCert* [82] adds separate certificate policies defined by domain owners which contain configuration parameters for the PKI protocol and security parameters for certificate validation. The selected CAs sign a bundle of these policies so that they can be published to the LexTree along with TLS certificates. However, PoliCert

relies heavily on auditor servers responsible for log monitoring, which are queried by clients to approve the proof result.

Certificate Issuance and Revocation Transparency (CIRT) [83] combines an append-only log like in CT with a LexTree to build a PKI specifically for end-to-end encryption in messaging or email systems. While the LexTree contains certificates like in AKI, the log includes an entry for each certificate together with a fingerprint of the LexTree at the time of insertion, which enables participants to verify (non-)inclusion as well as consistency between two versions of the log. Nevertheless, third-party auditors still have to ensure that none of the data structures exclusively contains a value by keeping a full copy of both and verifying each operation.

As an improvement, the *Distributed Transparent Key Infrastructure* (DTKI) [39] reuses the idea of combining a chronological with a lexicographical data structure to provide the same proofs as CIRT, but lets clients verify random parts of the log and "gossip" the results among themselves for collective monitoring of the logs. Further, it employs two types of log servers: multiple certificate log maintainers hold distinct subsets of the certificates, whereas a single mapping log maintainer is responsible for associations between domains and the logs containing the corresponding certificates. Since none of them needs to be trusted, this partitioning prevents an oligopoly of a single participant. Like ARPKI, the security property of DTKI has been formally proven.

In summary, domain-centric architectures serve as a useful basis for our system, as they also allow domain owners to have a say in the verification process. However, pinning- and DNS-based approaches have several shortcomings regarding security, like unprotected first access or, even worse, reliance on a trusted party. Blockchains seem to be a good choice regarding decentralization and the associated empowerment of clients. Yet, we would have to balance serious performance limitations and fundamental security concerns depending on the number of nodes. Therefore, we take a look at PKIs using log servers in the following section, since they promise accountability and non-repudiation for claims of the CAs and logs, as well as higher security in case of compromise.

2.2.4 Comparison

As observed above, domain-centric approaches based on log servers are the means of choice in our scenario. However, a closer look reveals that this is only partially true: some architectures have weaknesses that even make them worse than proposals from other categories. Nevertheless, we identify some suitable candidates below based on several characteristics summarized in table 2.1 by consulting analyses of multiple researchers [34], [39], [84], [85].

Security With regard to security of the proposals, we consider the ability of a PKI to mitigate a MitM attack. Since SK requires a certificate to be signed by a sovereign key, a bogus certificate would not be accepted, unlike in CT, where any logged certificate is considered valid (and remains in the log until it expires even on detection by the domain owner). CIRT enables the revocation of mis-issued certificates and provides proofs for

detecting them, but still fails in prevention like SK and CT, i.e. it cannot detect the registration of such a certificate by a malicious CA. Instead, AKI, ARPKI and PoliCert establish additional protocols for revoking a certificate and monitoring actions of CAs and logs by other components while DTKI allows clients to verify the actions of any participant, which makes them all robust in preventing MitM attacks.

This is also reflected by the number of participants that must be compromised in order to carry out a successful attack. In SK, CT and CIRT it is enough to control a single log server. The protocols in AKI and PoliCert ensure that the attacker has to compromise at least a log and a validating service, since CAs are not actively involved in the verification process. Only ARPKI requires that n servers behave maliciously to launch a successful attack (where n is a scalable system parameter), which has also been formally proven. Likewise, a formal proof for DTKI states that if the domain owner registered and verified a so-called master certificate before the attack, all parties can be compromised and the system would still be secure – however, in general, this constraint and also the assumption that clients communicate via a (unspecified) gossip protocol may not be fulfilled, which is why an attacker has to gain control over only one log server.

Further, neither CT nor CIRT support recovery if a domain owner loses their private key and DTKI supposes that this (as well as key compromise) cannot happen, whereas all the other architectures can handle key loss. From a client's perspective, SK, CIRT and DTKI do not preserve privacy, as they require the client to actively query a log server that can learn about the visited domains. For the other approaches, proofs are handed over to the domain owner who forwards them to the client. The other way round, a domain can configure security policies like involved parties in AKI, ARPKI and, even more, additional properties for the verification of a certificate in PoliCert.

Although being a desirable feature for enhancements of PKIX, we consider the support for multiple certificates of a domain as a vulnerability regarding our use case, since this undermines our method of determining integrity. Nevertheless, CT allows an arbitrary number of certificates by design, SK and DTKI enable multi-certificate registration through the sovereign and master key, respectively, and PoliCert binds multiple certificates to a single subject certificate policy defined by the domain owner.

Proofs The mentioned proposals behave transparent, i.e. they make the stored information publicly visible. Unless SK, the PKIs offer verifiable proofs that allow other parties to verify correctness of an operation. Naturally, this can be the modification and consequential presence or absence of a certificate. In case of append-only logs, they support proving that the current database version is a consistent extension of an earlier one.

Since CT uses such an append-only log, it can provide efficient proofs for included certificates as well as for consistency. However, proving that a domain is absent or querying the most recent certificate for a particular domain would result in an exhaustive search on all certificates contained within the log. Instead, the LexTree used in AKI, ARPKI and PoliCert serves as a key-value store and allows removal of a certificate. This implies that consistency proofs are impossible, but besides presence and absence, a query about the currently unrevoked certificates can also be answered efficiently in a verifiable manner.

Finally, the combination of these two data structures in CIRT and DTKI also results in a union of the proof capabilities, since the LexTree reflects the validity of certificates and the chronological log describes their evolution over time.

Deployability Finally, we emphasize some deployability concerns that are especially important for a TLS infrastructure. Nevertheless, they could assist us in deciding which approaches to take inspiration from.

In SK, CIRT, PoliCert and DTKI, the client is responsible for fetching the required information (the key of the domain in SK, proofs in the others) directly from an involved party, whereas CT, AKI and ARPKI provide their proofs to the domain owner who appends them to the client request. Therefore, the former architectures depend on extra client communication, which excludes offline verification.

Redundancy of data combined with monitoring of other parties can increase the security level – yet, CT, CIRT and DTKI specify no mechanism for synchronization between logs, while AKI, ARPKI and PoliCert embed mechanisms for synchronizing between log servers in their modification protocols.

	SK	**CT**	**CIRT**	**AKI**	**ARPKI**	**PoliCert**	**DTKI**
Security							
MitM attack mitigation	✓	×	✓	✓	✓	✓	✓
Attack prevention	×	×	×	✓	✓	✓	✓
Certificate revocation	✓	×	✓	✓	✓	✓	✓
Compromised parties for attack	$^1/_1$	$^1/_1$	$^1/_1$	$^2/_3$	$^n/_n$	$^2/_3$	$^1/_1$
Formal Security Proof	×	×	×	×	✓	×	✓
Domain key recovery	✓	×	×	✓	✓	✓	×
Client connection privacy	×	✓	×	✓	✓	✓	×
Configurable security policies	×	×	×	✓	✓	✓	×
Multiple certificates supported	✓	✓	×	×	×	✓	✓
Proofs							
Certificate present	×	✓	✓	✓	✓	✓	✓
Certificate currency	×	×	✓	✓	✓	✓	✓
Certificate absent	×	×	✓	✓	✓	✓	✓
Consistency between logs	×	✓	✓	×	×	×	✓
Deployability							
Extra client communication	✓	×	✓	×	×	✓	✓
Synchronization required	✓	×	×	✓	✓	✓	×

Table 2.1: *Comparison of domain-based PKI architectures using log servers [34], [39], [84], [85].*

2.3 Merkle Trees

The domain-centric architectures compared in section 2.2 use log servers based on a data structure called *Merkle Tree*, originally introduced as digital signature for large amounts of data [33]. A Merkle Tree allows efficient membership proofs for included data based on hashes, and the security of a proof reduces to the collision resistance of the hash function used for construction. We will give a rather informal description in this section which suffices for our purposes – however, we refer the interested reader to the formalizations of Yu et al. [39] which serve as a basis for the following.

2.3.1 Terminology

Before defining a Merkle Tree, we need to introduce some basic notions [86]: A directed graph $T = (N, E)$ consisting of a set of *nodes* N and *edges* $E \subseteq N \times N$ is called a *tree* if N is empty or contains a node *root* without incoming edges (i.e. there's no edge $(n, root)$ in E) and every other node in N has exactly one incoming edge and can be reached from *root*, i.e. there exists a *path* $root \to n_1 \to n_2 \to \cdots \to n_{n-1} \to n_n$ defined by edges $(root, n_1), (n_1, n_2), ..., (n_{n-1}, n_n) \in E$ for every node n_n. Any node from which a node n is reachable is called *ancestor*, any node which can be reached from n is called *descendant*. Furthermore, the direct ancestor of a node is its *parent*, direct descendants are called *children*. Nodes without children are referred to as *leaves* and nodes with the same parent are said to be *siblings*. Note that, for any node n, $N' \subseteq N$ containing all descendants of n and n itself together with the corresponding edges $E' = E \cap (N' \times N')$ define a tree $T' = (N', E')$ which is called a *subtree* of T with root n.

Moreover, the length of the longest path from node n to a leaf is the *height* of n, while *depth* references the path length from *root* to n. The set of nodes with same depth d is called a *level*, the number of levels define the height (or depth) of the tree. While both level and height of the root are 1, its depth is 0.

Every node of a *binary tree* has at most two children, which are additionally marked as `left` or `right` child and ordered by a total relation. If each non-leaf node has two child nodes, the tree is *full*. To be *complete*, the depth of each leaf must be either the height h of the tree or $h - 1$. Finally, a full tree is *perfect* if all leaves have the same depth.

Basically, a *Merkle Tree* M is a full binary tree whose nodes are labeled by bit strings [33]. While the leaves "contain" the actual data as label, any non-leaf node is marked with the digest $H(\mathcal{L}(c_{\texttt{left}})||\mathcal{L}(c_{\texttt{right}}))$ of the labels of its left and right child $c_{\texttt{left}}$ and $c_{\texttt{right}}$, respectively. For every node n, these are ordered by an *in-order tree traversal* relation $\prec_M$ such that $c_{\texttt{left}} \prec_M n \prec_M c_{\texttt{right}}$. Further, M is always filled "left-to-right" – the left subtree of any node in M must always be perfect and at least as high as the right one. Thus, one can easily prove that the sequence of data in the leaves completely defines the tree; even further, two Merkle Trees are equal if and only if they contain identical sequences of data. This motivates to declare the *size* of a Merkle Tree as the number of its leaves.

2.3.2 Data Verification

As previously mentioned, Merkle Trees allow one to generate a proof that data is included, which is why we call them *inclusion proofs*. Of course, such a proof only exists if a node n_n labeled by b_n is actually a leaf of a Merkle Tree – if so, its inclusion proof is defined as $(p, [b_1, \ldots, b_k])$ where p is the *position* of n and $[b_1, \ldots, b_k]$ is a sequence of node labels of length k, which equals the depth of n in the tree.

The position of a node n in a Merkle Tree is typically a representation of the path to n, but for further use, we will use it as a synonym for the index of the corresponding label in the stored sequence of data. However, given the tree size N, an index can easily be transformed to a path representation indicating for each node with which child to continue:

$$path(p, N) = \begin{cases} \varepsilon & , \text{ if } p = N = 1 \\ \texttt{left} \cdot path(p, 2^{k-1}) & , \text{ if } p \leq 2^{k-1} < N \leq 2^k \text{ for a } k \in \mathbb{N} \\ \texttt{right} \cdot path(p - 2^{k-1}, N - 2^{k-1}) & , \text{ if } 2^{k-1} < p \leq N \leq 2^k \text{ for a } k \in \mathbb{N} \end{cases}$$

The label sequence effectively contains necessary information to compute all hash values which occur on the path $root \to n_1 \to ... \to n_n$: Starting with n_1, the values b_i are labels of the sibling of each node n_i in the path. Note that even for (very) large trees, the proof size remains logarithmic in the size of the Merkle Tree and therefore in the amount of stored data.

The attentive reader may have already noticed that the information an inclusion proof provides is enough to recompute the label of the root node, which we call *root hash*. Assuming a collision-resistant hash function, the root hash can indeed effectively serve as an integrity warranty for the tree contents, as two Merkle Trees only contain the same data if their root hashes are equal. Hence, a valid inclusion proof $(p, [b_1, \ldots, b_k])$ for a bit string b, a root hash h and a tree size N has to fulfill three conditions:

1. p is a valid leaf index for a Merkle Tree of size N, i.e. $1 \leq p \leq N$
2. k equals the number of levels except the root level in a Merkle Tree of size N
3. h is equal to $H_{chain}(path(p, N), b, [b_1, \ldots, b_k])$, which is defined recursively as

$$\begin{array}{lcl} H_{chain}(\varepsilon, b, []) & = & b \\ H_{chain}(\rho \cdot \texttt{left}, b, [b_1, b_2, \cdots, b_k]) & = & H_{chain}(\rho, H(b||b_1), [b_2, \cdots, b_k]) \\ H_{chain}(\rho \cdot \texttt{right}, b, [b_1, b_2, \cdots, b_k]) & = & H_{chain}(\rho, H(b_1||b), [b_2, \cdots, b_k]) \end{array}$$

Besides the logarithmic size of the proof, note that the computation of H_{chain} and therefore the proof verification also requires at most $\lceil log_2(N) \rceil$ hash computations.

For example, consider the Merkle Tree in Figure 2.3. The inclusion proof for the node with label b_4 is $(4, [b_3, H(b_1||b_2), H(b_5||b_6)])$, represented by gray nodes. To verify the inclusion of b_4 in a tree of size $N = 6$ with root hash $h = H(H(H(b_1||b_2)||H(b_3||b_4))||H(b_5||b_6))$, we have to compute $path(4, 6)$, which yields $\texttt{left} \cdot \texttt{right} \cdot \texttt{right}$.

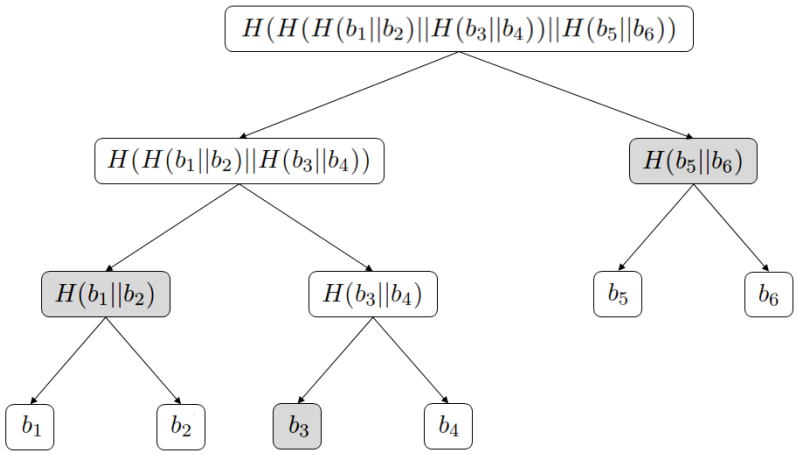


Figure 2.3: *A Merkle Tree of size 6 with highlighted inclusion proof for the node with index 4 and label b_4.*

Now, we can compute the hash chain and compare it to h:

$$
\begin{aligned}
&H_{chain}(\texttt{left} \cdot \texttt{right} \cdot \texttt{right}, b_4, [b_3, H(b_1||b_2), H(b_5||b_6)]) &&= \\
&H_{chain}(\texttt{left} \cdot \texttt{right}, H(b_3||b_4), [H(b_1||b_2), H(b_5||b_6)]) &&= \\
&H_{chain}(\texttt{left}, H(H(b_1||b_2)||H(b_3||b_4)), [H(b_5||b_6)]) &&= \\
&H_{chain}(\varepsilon, H(H(H(b_1||b_2)||H(b_3||b_4))||H(b_5||b_6)), []) &&= \\
&H(H(H(b_1||b_2)||H(b_3||b_4))||H(b_5||b_6)) = h \quad .
\end{aligned}
$$

As stated above, the collision-resistance of the hash function H is obviously critical for the security of an inclusion proof. Unfortunately, implementation-specific data encoding may still result in ambiguous hash values – for example, by using an empty string for non-existing values (which are then suddenly equal to actual empty string values). Therefore, the tree structure has to be additionally included in the hash computation. This is typically achieved by adding constants [87], e.g. by appending a prefix bit to distinguish between leaves and interior nodes [79].

2.3.3 Extension Verification

While inclusion proofs create confidence in log servers regarding authenticity and integrity of data, they reveal no information about the chronological evolution of a Merkle Tree. For example, a malicious server could publish a root hash and tree size from an entirely different (but valid) tree containing a value that should not exist; even if a client already has information about a previous version of the original tree, they would still have to accept this value based on its inclusion proof. We address this problem by prohibiting the deletion of a value in the tree (i.e., the log becomes *append-only*) which allows one to construct a *consistency proof* for verifying that a tree root is indeed an extension of a root that is already trusted.

Consider two Merkle Trees M and M' with root hashes h and h', respectively. Further, let N and N' be the tree sizes of M and M' such that $N < N'$. An efficient construction

of a consistency proof makes use of the fact that, if a binary tree is append-only, perfect (sub-)trees do not change – since a new value always gets the next larger index, it must be added as the new rightmost leaf. This implies that if $N = 2^k$ for a $k \in \mathbb{N}$, it is enough to verify an inclusion proof for h in M'. Otherwise, we know by definition that the left subtree of any node is perfect and can focus on the root node b_p of the highest perfect subtree P_M in M which contains the rightmost leaf of the whole tree: Since P_M is perfect and therefore is not changed, it has to occur somewhere in M'. Moreover, the left siblings of both P_M and its ancestors (which effectively form the inclusion proof sequence) are already perfect and consequently also in M'. Hence, it suffices to prove that M' contains P_M and to construct a valid proof of presence for M based on the proof for M'. As a result, a consistency proof for M and M' is defined as $[b_p, b_1, \ldots, b_k]$ where $[b_1, \ldots, b_k]$ denotes the inclusion proof sequence for b_p in M'. If M itself is perfect, b_p is omitted from the proof.

The depth of M equals $log_2(N)$ if it is perfect, which implies that any imperfect tree contains a perfect left subtree with depth $\lfloor log_2(N) \rfloor$ or, in other words, $N - 2^{\lfloor log_2(N) \rfloor}$ leaves in its right subtree. Thus, we can compute the depth of P_M using the formula

$$d_P(N) = \begin{cases} log_2(N) & , \text{ if } N - 2^{\lfloor log_2(N) \rfloor} = 0 \\ d_P(N - 2^{\lfloor log_2(N) \rfloor}) & , \text{ otherwise} \end{cases}$$

which is equal to counting the trailing zeros in the binary representation of N. By cutting off the last $d_P(N)$ words from $path(N, N')$, we receive a path representation ρ from the root of M' to b_p.

As mentioned above, if M is perfect, a valid consistency proof for M and M' is $[b_1, \ldots, b_k]$ if and only if $N < N'$ and $(N, [b_1, \ldots b_k])$ is a valid inclusion proof for the former root labeled by h in M', i.e. $H_{chain}(path(N, N'), h, [b_1, \ldots, b_k]) = h'$. Nevertheless, if M is not perfect, the consistency proof $[b_p, b_1, \ldots, b_k]$ must still fulfill $H_{chain}(\rho, b_p, [b_1, \ldots, b_k]) = h'$. However, we then have to additionally convert it into an inclusion proof to be verified for M. Since P_M contains the rightmost leaf in M, its path representation is always a concatenation of `rights` and its inclusion proof sequence only contains left children of all its ancestors. As a consequence, every `left` in ρ means that the subtree containing b_p in M' is a left child of the corresponding node, so the label of the right sibling in $[b_1, \ldots, b_k]$ can be ignored as its descending leaves have an index greater than N. Therefore, the proof of presence $(N, [b_{i_0}, \ldots, b_{i_n}])$ for b_p in M can be obtained through $decompose_{\texttt{right}}(\rho, [b_1, \ldots, b_k])$ which is defined as

$$\begin{aligned} decompose_{\texttt{right}}(\varepsilon, [b_1, b_2, \ldots, b_k]) &= [] \\ decompose_{\texttt{right}}(\rho \cdot \texttt{left}, [b_1, b_2, \ldots, b_k]) &= decompose_{\texttt{right}}(\rho, [b_2, \ldots, b_k]) \\ decompose_{\texttt{right}}(\rho \cdot \texttt{right}, [b_1, b_2 \ldots, b_k]) &= b_1 :: decompose_{\texttt{right}}(\rho, [b_2, \ldots, b_k]) \end{aligned}$$

where :: denotes the prepend operation of a single element to a sequence. Note that this verification procedure for the extension of two Merkle Trees requires a proof which is still logarithmic to the number of values in the extended tree. Also, only a maximum of $2 \cdot \lceil log_2(N') \rceil$ hash computations are required, since at most two inclusion proofs have to be verified.

As an example, we want to verify that the tree in Figure 2.4 is an extension of the Merkle

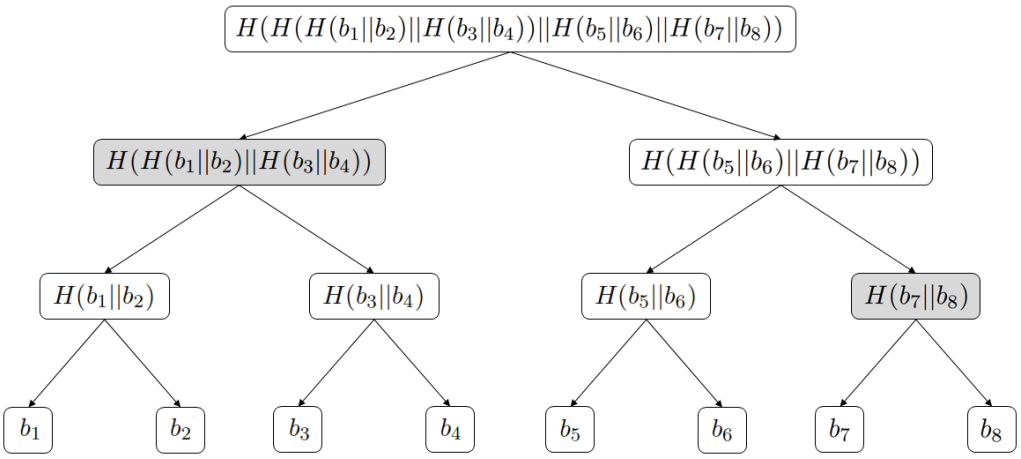


Figure 2.4: *A Merkle Tree of size 8 which is an extension of the tree in figure 2.3 with highlighted inclusion proof for the perfect subtree with root label $H(b_5||b_6)$ containing the node b_6.*

Tree in Figure 2.3, which we will denote as M' and M, respectively. Obviously, M isn't perfect, but there exists a perfect subtree containing the rightmost node b_6 – we receive it by computing $d_P(6) = 1$ and slicing the last word off $path(6, 8) = \texttt{right} \cdot \texttt{left} \cdot \texttt{right}$, which results in $\rho = \texttt{right} \cdot \texttt{left}$ and the corresponding node $H(b_5||b_6)$ of M'. Hence, the consistency proof is $[H(b_5||b_6), H(b_7||b_8), H(H(b_1||b_2)||H(b_3||b_4))]$.
The verification of this proof consists of proving inclusion of $H(b_5||b_6)$ in M' as well as in M. The former is the confirmation that

$$\begin{aligned} &H_{chain}(\texttt{right} \cdot \texttt{left}, H(b_5||b_6), [H(b_7||b_8), H(H(b_1||b_2)||H(b_3||b_4))]) \\ &\quad = H(H(H(b_1||b_2)||H(b_3||b_4))||H(H(b_5||b_6)||H(b_7||b_8))) \end{aligned}$$

which we generously leave to the reader; the latter requires decomposing the inclusion proof for M' which lets us prove that $H(b_5||b_6)$ is in M:

$$\begin{aligned} &decompose_{\texttt{right}}(\texttt{right} \cdot \texttt{left}, [H(b_7||b_8), H(H(b_1||b_2)||H(b_3||b_4))]) \\ &\quad = decompose_{\texttt{right}}(\texttt{right}, [H(H(b_1||b_2)||H(b_3||b_4))]) \\ &\quad = H(H(b_1||b_2)||H(b_3||b_4)) :: decompose_{\texttt{right}}(\varepsilon, []) \\ &\quad = H(H(b_1||b_2)||H(b_3||b_4)) :: [] = [H(H(b_1||b_2)||H(b_3||b_4))] \end{aligned}$$

Because $path(6, 6)$ yields $\texttt{right} \cdot \texttt{right}$, we know that the path to $H(b_5||b_6)$ in M is only $\texttt{right}$ (because of $d_P(6)$) and can verify that it is indeed a member of M:

$$\begin{aligned} &H_{chain}(\texttt{right}, H(b_5||b_6), [H(H(b_1||b_2)||H(b_3||b_4))]) \\ &\quad = \cdots = H(H(H(b_1||b_2)||H(b_3||b_4))||H(b_5||b_6)) \end{aligned}$$

2.3.4 Sparse Merkle Trees

While Merkle Trees can guarantee the presence of data by inclusion proofs, querying particular values is inefficient because it requires one to iteratively visit each leaf until the data is found. Even worse, proving absence of a value is basically recomputing the

whole tree and therefore requires all leaves with their corresponding inclusion proofs to be transferred. Therefore, Yu et al. describe a compact data structure that allows addition and deletion of key-value pairs, as well as generating proofs of presence and absence using an AVL-Tree [86] as LexTree in which each node contains a hash of its children in addition to data [39]. However, we can achieve the same behavior using Merkle Trees [88].

Let S be a perfect Merkle Tree of size $2^{|H|}$, where $|H|$ denotes the fixed length of the outputs a hash function H yields. We introduce a constant ε denoting an empty leaf value – hence, S can contain a sequence of key-value pairs with size $n \leq 2^{|H|}$, as the remaining leaves are empty. Further, we associate a key-value pair with a particular leaf: As we "read" the hash value of the key from left to right, we replace each 0 and 1 by `left` and `right`, respectively, allowing us to follow the resulting path from the root to a leaf with the hashed key as index. Thus, addition and deletion of a value is simply re-labeling the corresponding leaf with either the value or ε. Finally, S can provide a proof of presence for a key-value pair by generating an inclusion proof of size $|H|$ for the associated leaf with the actual value – to prove absence, the leaf must be labeled with ε.

Considering a real-world hash function like SHA-256, a perfect Merkle Tree would have 2^{256} leaves – it is therefore obviously not computable. Nevertheless, Laurie and Kasper observed that in a *Sparse* Merkle Tree (or SMT), almost all leaves are empty, which means that their parents are labeled by $H(\varepsilon||\varepsilon)$, their parent nodes by $H(H(\varepsilon||\varepsilon)||H(\varepsilon||\varepsilon))$, and so on [88]. We denote these digests that only derive from empty leaves as *(empty) default hashes*. Since they are identical for most nodes and can be cached, it suffices to compute digests individually only for non-empty leaves and their ancestors. Even more, the tree can be represented efficiently by caching interior nodes whose children are both labeled by non-default hashes, so-called *branches* [89]. The branch cache usually contains $N-1$ nodes for N values stored in the SMT, but depending on memory consumption, it can be shrunk by probabilistically removing branches or extended by additionally caching the children of a branch. Regardless of these adjustments, the time required for both modification and proof construction scales as $\mathcal{O}(\log N)$ per leaf.

In terms of security, a Sparse Merkle Tree has a major disadvantage compared to an append-only Merkle Tree, that is *history independence*: Consistency proofs cannot be created, as they rely on extension of an unmodifiable tree. Hence, we have to consider attacks in a multi-instance setting due to incomparability of several SMTs (e.g. resulting from a modification). In [53], Melara et al. present a collision-resistant hash strategy H_C for this scenario: A node n with children $c_{\texttt{left}}$ and $c_{\texttt{right}}$ is hashed by

$$H_C(n) = \begin{cases} H(k_{empty}||k_t||i||d) & \text{, if } n \text{ is labeled by an empty default hash} \\ H(k_{leaf}||k_t||i||d||v) & \text{, if } n \text{ is a non-empty leaf with value } v \\ H(H_C(c_{\texttt{left}})||H_C(c_{\texttt{right}})) & \text{, otherwise} \end{cases}$$

where H is a hash function, d denotes the depth of n and i is its unique index (for an interior node, i is the d-bit prefix of the leaf indexes reachable from n). Both d and i ensure that preimages cannot be valid across multiple locations in a tree, while the tree-specific unique constant k_t protects against second-preimage attacks using multiple trees. Without them, an adversary could search across an arbitrary number of SMTs and locations in

parallel to find a hash collision [89]. While in logs, one has to prefix a bit before hashing to differentiate nodes and leaves, we can safely ignore this for sparse trees as a proof of presence is always of fixed length – however, it is necessary to distinguish empty leaves and zero-length values by using constants k_{empty} and k_{leaf} because otherwise, we would lose the benefits of the sparseness property and had to calculate all $2^{|H|} - 1$ labels on every change. Yet, even the hash strategy H_C cannot protect a client against a malicious SMT server which changes a particular leaf the client requested, but resets this update if anyone else queries the value. As a solution, root hash changes of an SMT could be logged in a verifiable manner so that they become publicly visible [88].

2.3.5 Verifiable Log-Backed Map

As described in [90], a *Verifiable Log-Backed Map* is a Sparse Merkle Tree combined with an append-only Merkle Tree as operations log. The SMT is populated by applying the sequence of mutation operations stored in the log leaves, and the resulting root incorporates the size and root hash from the underlying log (in addition to the root hash). If further, a separate verifiable tree stores a history of the map size together with the corresponding root hash, any client can verify that the current view is consistent with previous versions of the map by a consistency proof of the underlying log and that it is indeed the latest version by an inclusion proof of the history log. Moreover, they can trust the proofs obtained from the server, because other parties auditing the history log would observe malicious changes to the log-backed map. Furthermore, an auditor could replay the underlying log to verify that the server behaved correctly over time. Note that we use the term auditor rather than client, since this requires downloading the entire dataset and only a small number of dedicated clients is expected to do so. For this reason, we make the same distinction in Table 2.2, which compares the data structures append-only Merkle Tree (*Log*), Sparse Merkle Tree (*Map*) and *Log-Backed Map* regarding computation time and proof size. *Efficiently* means that exactly one proof is required, whose size and time are logarithmic in the number of values stored and therefore can/should be performed by clients, whereas *Audit* denotes recomputing the root hash using the downloaded dataset. Finally, any proof that is either not bounded logarithmically in time and size, or depends on multiple proofs for at least some other entries in the tree is considered *Impractical*.

2.3.6 Fields of Application

Several public key infrastructure architectures developed for TLS security use append-only Merkle Trees. CT signs and stores TLS certificates in this data structure, which allows CAs to provide non-repudiable proofs to domains. These can pass them on to a client when establishing a TLS connection, and auditors can validate the consistency between versions of the logs. To handle the revocation of such certificates efficiently, the supplementary system *Revocation Transparency* [88] uses a Sparse Merkle Tree indicating whether a particular certificate has been revoked or not, but still requires full audits to verify correct behavior of the log and map. As an alternative, AKI introduces a variant of Merkle Trees called LexTree that contains certificates sorted lexicographically by their domains and is also used

	Log	Map	Log-Backed Map
Prove inclusion of value	Efficiently	Efficiently	Efficiently
Prove non-inclusion of value	Impractical	Efficiently	Efficiently
Retrieve provable value for key	Impractical	Efficiently	Efficiently
Retrieve provable current value for key	Impractical	No	Efficiently
Prove append-only	Efficiently	No	Efficiently*
Enumerate all entries	Audit	Audit	Audit
Prove correct operation	Efficiently	No	Audit

*Although append-only can be proven efficiently for the underlying log, verifying the correctness of an append-only operation requires an Audit.

Table 2.2: *Comparison of append-only Merkle Trees (Logs), Sparse Merkle Trees (Maps) and Log-Backed Maps according to [90].*

in ARPKI. CIRT adapts this idea and combines it with a traditional append-only Merkle Tree to persist combinations of a certificate and the corresponding root hash, whereby (non-)inclusion and consistency can easily be verified. However, inconsistencies between these trees can only be detected by third-party auditors that mirror both and verify all modifications. PoliCert adapts concepts of ARPKI and CIRT by using an append-only Merkle Tree and a LexTree together with heavy protocols for modification, so that in addition to presence and absence proofs of certificates, consistency proofs between both of them as well as their earlier versions are possible. As mentioned before, DTKI uses a special AVL-Tree as LexTree to implement a data structure equivalent to Sparse Merkle Trees. This tree is combined with append-only Merkle Trees to maintain associations between top-level domains and logs containing the corresponding certificates.

Because Merkle Tree proofs include very little information about other values stored in the tree, they are also a means for privacy in transparent logging. The provably secure authenticated data structure *Balloon* [91] combines an append-only Merkle Tree as event log with a lexicographically-sorted treap used like a Merkle Tree [87] as key-value store to provide efficient (non-)membership proofs. Unfortunately, treaps store values in each node, which requires information about key-value pairs of ancestors to be leaked in a proof for recomputing the root hash [51]. *CONIKS* [53], a privacy-preserving key verification service developed for end-to-end encrypted communication systems, allows clients to manage and monitor their key-bindings efficiently using a Merkle Prefix Tree, which is a dynamically sized SMT where empty subtrees are represented by empty nodes. Indexes are obtained by transforming user names with a verifiable random function before hashing to prevent collision attacks based on index prefixes. A protocol for lookups and monitoring hides the total number of registered bindings.

Blockchain-based PKIs like *Claimchain* [92] use lexicographically-sorted Merkle Trees in each block to provide efficient verifiable lookups for domain certificates, whereas *Chainiac* [57] uses an append-only tree to compute a signature over binary software resources.

Originally developed as a cryptographic signature for static datasets, Merkle Trees have also been investigated in the broader field of *incremental cryptography* [93] for dynamic data. Li et al. [94] apply the structure and mechanisms of a Merkle Tree to a B-Tree [95] which allows them to authenticate index structures for outsourced databases. The characteristics of incremental Merkle Trees have been formalized and proven in [96].

In live systems, Merkle Trees are used from distributed systems to decentralized networks where authenticity and integrity are essential. Besides public-key infrastructure architectures, the fields of applications include peer-to-peer networks (e.g., used for crypto currency), NoSQL databases, and distributed file and revision control systems. The blockchain-based peer-to-peer networks *Ethereum* [97] and *Bitcoin* [98] use Merkle Trees for storing transactions in the hash chain. Each block in the chain contains a tree of transactions to verify local copies. Nevertheless, if a client wants to check only a specific transaction, they can just download the chain of block headers that contain the root hash and an inclusion proof instead of the whole tree [99]. Some NoSQL database systems like *Apache Cassandra* [100] and *Amazon Dynamo* [101] rely on Merkle Trees to enforce the synchronization of replicas on nodes. Each node exchanges a tree of the values it stores with the requesting node, which then compares the root hashes to decide whether it has to synchronize with a node or not [101]. Distributed file systems such as *IPFS* [102] or *ZFS* [103], as well as revision control systems like *Git* [104] and *Mercurial* [105] build specialized Merkle Trees containing file content hashes. They are used as an identifier for content addressing, as a checksum for tamper resistance and as a mean for deduplication – further, they allow a simple distribution of version changes [102].

3 Certificate-based Resource Alteration Prevention

In order to build a system for effectively defending clients against targeted attacks based on tampering, we have to answer fundamental questions regarding its functionality and design. First, we have to ask how alteration can generally be detected – since several proposals dealing with third-party adversaries [27], [29] suggest verifying the resource hashes, we take up the idea. This also seems to be the most straightforward approach, as result checking [106] or similar methods would require a full specification and implementation of verification procedures for any part of the software. However, one needs digests to compare the computed resource hashes to, which raises the question of how they can be delivered to clients with regard to our adversary model. In this respect, a centralized architecture with a single server on which application developers can publish the hashes is obviously out of the picture. Hence, there are two options: We could either have clients communicate directly with each other and exchange the hash values, or we could install a public key infrastructure with multiple server components that can guarantee integrity of the digests through verifiable proofs and security protocols.

In case of a compromised party, both need a mechanism to achieve consensus. If we demand that at any time, all participants have to agree on the hash value of a particular resource, the former solution implies that each client has to know everybody else and any resource digest of any web application that another random client could visit. Therefore, they must have been "registered" by their publishers – which in turn requires a process for establishing trust, so that a client can check the origin of the hashes. For example, a digital signature could provide a remedy, but only works if clients have the publishers' public keys built-in (including update capability if it gets lost or compromised). Further, we would need a kind of update procedure for new resource versions. Since all clients must agree, each one would have to be permanently online in case of an update or be left out when voting for the correct digest until they have synchronized with the others. For this reason, we could weaken this criterion of strong consistency such that only a quorum of clients is necessary for believing in the integrity of a hash value. Unfortunately, the adversary could compromise some of them as well, and would win if they control at least the quorum on average.

On the other hand, the latter option promises the properties of PKIs described in chapter 2 and shifts the obligation to store all the information from clients to servers. Because these are continuously available, consensus can be broken down to monitoring the synchronization process, which allows publishers to easily register and update their resource hashes by the use of efficient protocols. If on top, verifiable proofs can be generated for any action of the servers, neither clients nor publishers have to trust them blindly. Nevertheless, we must be careful – some PKIs no longer offer protection if just a single server has been compromised, while others also fail once a part of the infrastructure has been successfully attacked. Thus, they provide even less security than the quorum approach discussed above.

A decentralized PKI still seems more promising than a fully distributed system, if we manage to avoid pitfalls like failure on compromise of a single party described previously. Although the desired properties for our scenario might differ in part from those that typically apply to TLS infrastructures, we could at least take our inspiration from the architectures compared in section 2.2.4, which, for example, already allow updates and deletions by certificate revocation and can prevent MitM attacks. However, we have to figure out which characteristics should be considered important to design a PKI for our needs. Hence, the following section 3.1 clarifies on which requirements we focus for our approach. Afterwards, section 3.2 gives an overview of the participating entities we consider in our system, while section 3.3 defines the formats that we use to structure necessary information. Bringing it all together, we describe a communication protocol that ensures the required level of security in section 3.4 and analyze it in section 3.5. Subsequently, section 3.6 explains some aspects of our prototypical implementation and we finally conclude this chapter with a discussion about possible improvements and economic considerations in section 3.7.

3.1 Requirements

Of course, we want to build a system that makes it almost impossible to successfully attack a specific client through altered resources. Therefore, the following focuses on major security properties our approach must fulfill. Nevertheless, quality is taken into account as well – besides the efficiency of our protocols, an uncomplicated deployment is essential for being accepted by both publishers and clients.

3.1.1 Security Attributes

Naturally, our main focus lies on detecting resources that have been tampered with. Yet, we would like to satisfy additional security requirements: If an adversary launches an attack, it should become permanently visible, regardless of whether it is a MitM attack or executed by a compromised party. Of course, this requires that we know the identity of each participant including publishers. Our system must therefore legitimate registrations, updates and deletions of identities and application resources.

Tamper Protection This is the core property we want to satisfy. If a client downloads an application resource from a particular address that differs from the resource other clients receive from the same link, this must be detected even on first access. As we solve this through a public key infrastructure, this implies that the resource hash and the corresponding registered hash must be identical.

Attack Prevention Obviously, tamper protection can only be achieved if the hash provided by our PKI is correct. Thus, we have to detect and prevent attacks on participating entities. An adversary must not be able to impersonate a participant in our system through a MitM attack, nor to foist bogus digests on a client if they infiltrated some of them.

Note that we explicitly demand resilience as long as not all involved parties have been compromised.

Operation Legitimization In order to prevent an attacker from simply publishing hash values for modified resources, the involved participants must ensure that a publisher's identity is known to the system and validate that they actually own the domain of the application they want to register. Further, updating requires checks of additional meta information, e.g. of an incremented software version number to indicate an intentional resource change.

Attack Visibility If an adversary ever succeeds in compromising entities or registering a bogus value, the attack must be publicly visible to other parts of the system, as well as to publishers and clients. Then, anyone willing to monitor or audit the operations and stored hashes is able to detect it and can identify both the resources which have been tampered with and the responsible party behind it.

Scalability Despite the fact that scalability is commonly considered a quality attribute, we have to look at it in terms of security. As mentioned above, some decentralized PKI architectures fail if a fixed number of servers has been infiltrated. Hence, we prefer a protocol which ensures that the number of compromised parties required for a successful attack scales with the overall number of participating servers, as this results in a more resilient system regarding security concerns.

3.1.2 Quality Attributes

Although the functional requirements above would suffice for our purpose, we should also consider ease of use. Since users might despise security solutions because of poor usability [107], it is worth taking care of simplicity to lay the foundation for productive use.

Simple Deployment We do not want to force developers to change their deployment process, nor browser vendors to implement our protocol. Ideally, publishers should be able to easily integrate ready-to-use components in their scripts that require only minimal configuration effort and deal with the detailed communication steps for registrations, updates and deletions. Moreover, it should be possible for interested users to download and install a lightweight browser extension that ensures tamper protection without further actions.

Efficiency When accessing a web application, the verification process should not block the page load. This isn't even crucial, since browsers typically sandbox web pages by default [108], [109] and prevent the application from accessing sensitive information without the user's permission. However, verifying both the resource digests and the actions of other participants must not cause unnecessary network consumption or waste computational resources. Otherwise, this could lead to a significant delay the user won't tolerate. In

contrast, publishing new hashes might take some time if it fits seamlessly in the (usually time-consuming) deployment process.

3.2 Participants

Now that the requirements for our system have been defined, we can move on to assigning the resulting tasks to participants. As we already know from section 2.2.3, a domain-centric architecture is the most suitable because it also allows the application owner to check actions of other parties against their policies. Therefore, we follow the separation of concerns proposed in several of the log-based approaches [34], [39], [80], [82], so that we can benefit from the transparency and security concepts they provide. Consequently, we consider the following entities as participants in our public key infrastructure:

Publisher As the creator of the web application, a Publisher is responsible for registering and updating the resource hashes of the current version. They have to introduce themselves to other parties and to reveal the domains they own, so that none of them is able to deceive clients with bogus digests for applications of others. Further, clients should receive a fingerprint of the current hash values when accessing their service, since this helps in preventing attacks by a compromised party.

Certificate Authority Due to the need to verify information about publishers, we introduce an independent Certificate Authority (CA) to legitimate their actions. Initially contacted by a publisher who transmits their domains, it certifies the ownership by signing them together with the publisher's public key. In order to register application resource hashes, the digests have to be signed by this key to prove the publisher's identity. Hence, a CA supervises the modification process and monitors the operations of other authorities and log servers for correct behavior.

Integrity Log Server Because each CA can only have knowledge of the publishers and applications it accepted, our system relies on parties named Integrity Log Servers (ILS) to store the hash values. Each of them maintains a verifiable log-backed map described in section 2.3.5 to save publisher keys and resource hashes. Integrity can therefore be verified through inclusion and consistency proofs which are accessible to all other participants. On modification, ILSes synchronize autonomously between themselves and provide signed proofs to CAs which hold the old Merkle Tree roots to verify correctness of the operations.

Monitor Optionally, anyone willing to audit the actions of CAs and ILSes can act as a Monitor. This party replays the modification entries in the append-only log of an ILS and ensures that the resulting map is equal to the snapshot provided by the log server. Since CAs offer signed tree roots to compare the ILS roots to, a monitor can verify global consistency among CAs and ILSes by verifying all inclusion proofs.

Client When accessing a website, the browser extension installed on the client's computer requests the resource hashes together with the publisher's key from a CA. After forwarding the request to an ILS, each CA verifies and signs the returned consistency and inclusion proof before responding the latter to the client. After verifying the signatures of each participant and checking against the fingerprint provided by the publisher, the extension compares the digests with hash values it computed for received resource files to detect tampering. Typically, global consistency is not verified by the browser extension because this might be quite time-consuming. Still, clients can explicitly take on the monitor role if they want to.

Auditor Since our overall objective is to design a security system for web applications, we provide a mechanism to include trustworthy results of code analyses. We do not restrict the method to use, so it can range from simple code reviews to formal verifications. However, the analysis has to be carried out by an approved expert that is called Auditor. They can announce their public key just like the publisher and register assertions for an application which are built into and verified by the browser extension. Nevertheless, such analysis results are only considered valid if the publisher provides a corresponding fingerprint to the client.

Although auditors and their assertions are a useful means to inspire client confidence in the application, they are not essential for protecting our system against tampering. In fact, the combination of CAs, ILSes and publishers alone guarantees that the core property can be verified by clients. However, auditors are indeed among the entities that can create, update or delete entries. But, since we assume that the private keys of at least CAs and ILSes are never accessible to any third party, we can ensure authenticity by committing them to sign each modification and enforce privacy through public key encryption. As CAs additionally legitimate and verify every operation of all other participants, our architecture effectively prevents MitM attacks and even invalid modifications by one or more compromised parties. Further, such attacks are always visible: The append-only log of the log-backed map allows anyone to prove consistency and inclusion of any action ever made. Even if all CAs and ILSes are compromised, clients can still match possibly forged digests against signed fingerprints from the publisher.

Obviously, this brief description is not enough to convince skeptical readers of the security features we claim. That is why we go into detail in the following sections. Before defining the actual protocol, we have to deal with how to structure the information that is needed in addition to hashes in a meaningful way.

3.3 Certificates

As the name certificate authority suggests, we use a simple certificate format to bundle entity specific attributes. For example, publishers and auditors must identify themselves with both their public key and their domains in order to create application- or audit-specific

certificates – hence, they are combined in a Publisher Certificate that is signed by the CAs. Figure 3.1 gives an overview of the different types that we consider in our system.

Besides domain attributes, the certificates contain further parameters for configuring the registration process and attesting its correctness, which are similar to those proposed in ARPKI. This allows participants to actively select who they want to include in the publishing process. The `cas` attribute specifies an ordered list of CAs that are contacted sequentially to review and sign the certificate, `ilses` contains a sequence of log servers where the accepted certificate is stored and `caMin` indicates the minimum number of CAs required to consider a subsequent certificate valid on update. Note that we demand that at least 2 CAs are involved as the lower bound for this parameter. Since these attributes are configurable by publishers and auditors, this might be a security risk: Using two disjoint subsets of both CAs and ILSes, an adversary could register two certificates for the same application. However, we do not want to restrict the freedom of an honest publisher if they do not trust a party. Further, imagine an existing system would require all authorities and log servers to be involved: if it was extended by new CAs or ILSes, certificates which are already registered would become invalid since they would have to be respectively signed by or stored in all of them. Thus, we accept this vulnerability and rely on monitors to verify global consistency. Nevertheless, we still strongly recommend to register all currently available CAs and ILSes in a publisher's and auditor's configuration. After the specified CAs verified the configuration and domain attributes of a raw certificate and ensured that the `validity` does not start in the past, they sign it for the publisher who then combines their `signatures` in the same order as the CA identifiers to obtain the final registrable certificate.

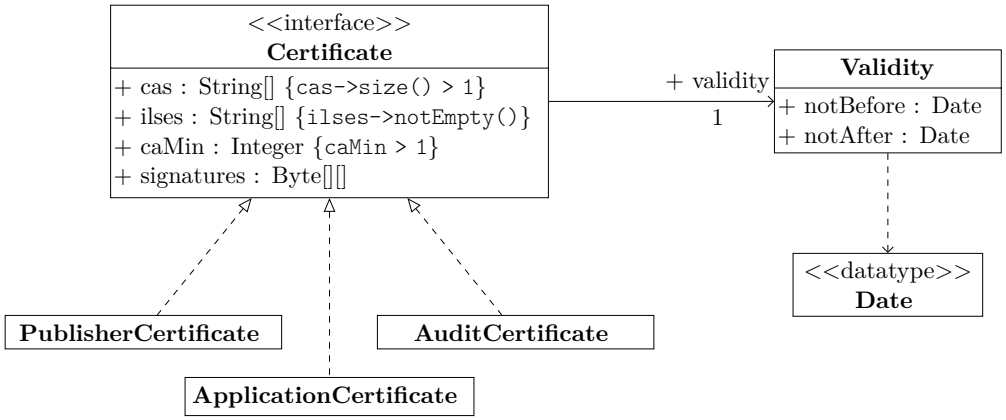


Figure 3.1: *The common and process related certificate attributes. All of them are read-only, but the annotations were conveniently omitted.*

So far, we have not addressed the questions of what the registration process of a certificate looks like and how it can be confirmed to the publisher on success. While we leave the former to the next section, the latter is a matter of signature structuring. Since each CA listed by the publisher must approve the modifications of ILS entries, we could broadcast

the proofs they are supposed to sign and respond with a bunch of signatures. However, if we contact only one of them which confirms the operation and subsequently contacts the next one (and so on), we reduce the network traffic and achieve that the actions of each are verified. To indicate that a CA accepted a certificate and verified the behavior of all previous ones, we build a signature chain like in ARPKI that we call `MultiSignature`. After the ILS created a `signature` of the certificate, each CA signs the previous multi-signature on approval - the old one is then stored in the initially empty `chain`. Finally, it is handed over to the publisher as `acceptanceConfirmation` of a `RegisteredCertificate` (see Figure 3.2), who can recursively verify that the process has been executed correctly. In addition, they can sign it as well and provide the result as fingerprint to the client.

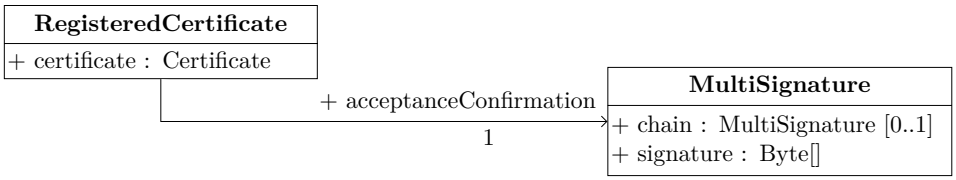


Figure 3.2: *The registered certificate containing the confirmation signature.*

3.3.1 Publisher Certificate

Both publishers and auditors own a Publisher Certificate, as they are able to create, update or delete entries. Of course, the CAs could simply verify each Application or Audit Certificate manually by checking the domain ownership and contacting the requesting party. However, we expect resource changes due to version updates to be much more frequent than key loss or compromise which implies an enormous effort for the authorities. This encourages one to adopt the idea of master keys as used in DTKI because certificates signed by such a key can be verified automatically.

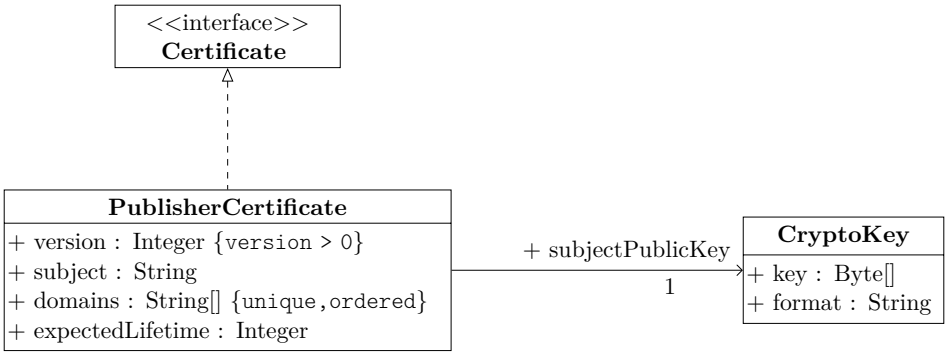


Figure 3.3: *The domain-specific attributes of a Publisher Certificate.*

As shown in Figure 3.3, our equivalent is the `subjectPublicKey` field of a Publisher Certificate. Each CA verifies the unique `subject` name of a publisher or auditor and the `domains` they claim to own, so that afterwards, it can automatically accept all Application or Audit Certificates signed by the corresponding private key.

Although auditors typically won't deploy web applications under the domains they specified, this field must still be provided in order to be contactable by others. Because we consider the key loss of a master key in contrast to DTKI, Publisher Certificates have a `version` number that must be incremented for every new one. For simplicity, we only include the encoding `format` for public key bit strings instead of fully specifying the algorithm with optional parameters like in X.509 certificates.

The purpose of `expectedLifetime` is to help clients in noticing timed-release attacks. An infiltrated publisher could analyze the time frame in which a client usually accesses their service. Afterwards, they publish a proper certificate for the modified resources they want to attack with that is valid only for the observed time window and immediately replaced by the "secure" Application Certificate when it expires. This makes it very likely that the target trusts and uses the tampered resources, while it nearly prevents third parties from effectively analyzing the resources for vulnerabilities. On the other hand, a malicious auditor could prevent a specific client from using a service by publishing a bogus Audit Certificate the same way. Therefore, we demand that a Publisher Certificate contains the approximate duration of the release cycle which conforms to the expected lifetime of an Application or Audit Certificate. As we consider significantly shorter values as suspicious, such attacks can be detected by monitors and exposed by analyzing the resources listed in the short-lived certificate. Since the latter requires actual source files, publishers must ensure that they are still available on request for at least one year, if they do not want to be listed as untrustworthy.

3.3.2 Application Certificate

With a registered Publisher Certificate, the domain owner is able to publish digests for a web application. The corresponding Application Certificate includes a set of `resources`, each described by its `resourceHash` and the `contentUrl` it is received from. The application itself can be identified by a unique `applicationUrl` that must be a subdomain (with an optional path suffix) of some registered domain listed under the `publisher` field. This also contains the public key so that CAs and clients can verify the included publisher's `signature` to ensure that the attributes are authentic. The `deploymentVersion` is incremented on each update to indicate an intended change of the resource digests by the publisher. Note that the remaining properties shown in Figure 3.4 are similar to those of a Publisher Certificate. In fact, the registration process for each of our three certificates does not need to distinguish between them.

3.3.3 Audit Certificate

We want to encourage domain owners to have their applications analyzed for security vulnerabilities by third parties. Since some may not want to disclose their source code to the

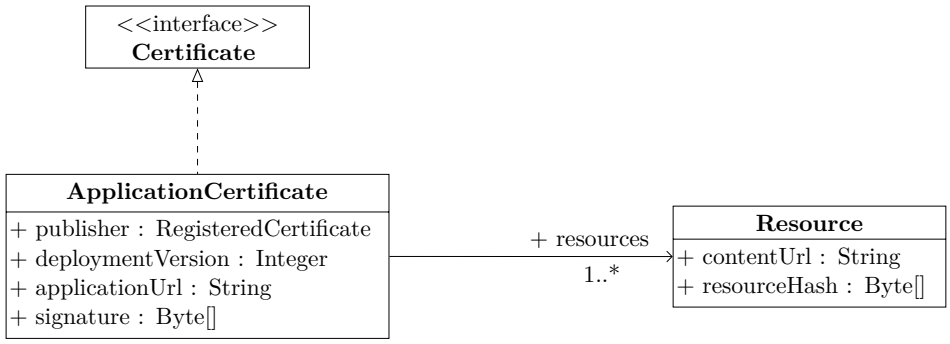


Figure 3.4: *The structure of an Application Certificate to describe resource digests.*

general public, publishers can contact an `auditor` they trust and who is registered in our system. After the analyst applied analysis `methods` of their choice, they publish an Audit Certificate shown in Figure 3.5 for an `applicationUrl` and fixed `deploymentVersion`. This ensures that each resource digest update has to be reviewed so that applications cannot pretend to be audited based on outdated certificates. One might wonder why we do not just use the Application Certificate that already contains the necessary information – the reason is that the listed `resources` may differ. While the application resources have to include all files transmitted to the client's browser, the Audit Certificate usually specifies only a subset of them. For example, font definitions, broadly used style sheets or third-party libraries are not likely to be analyzed separately in each review.

The results of the analysis are stated in audit `properties`. Besides an expressive `name`, they provide a `description` comprehensible for end users. Furthermore, they refer to `assertions` built into the extension by unique identifiers – however, since we may not be able to design a verification algorithm for all properties that come to an auditor's mind, this field remains optional.

Finally, the analyst affixes their `signature` to the mentioned audit attributes together with the configuration parameters used for registration. In the following, we take a detailed look at the communication flow and the individual protocol steps, which guarantee us the required security and quality features of our system.

3.4 Protocol

We now have all the building blocks to proceed with specifying the protocol. Based on the comparison of domain-centric public key infrastructure architectures using log servers in the last chapter, we choose ARPKI to take inspiration from – the attentive reader will have guessed it, since we have already introduced some ARPKI-oriented certificate fields that a publisher can configure. The main reason for this choice is simple: ARPKI is the only PKI that is formally proven to offer resilience until all certificate authorities and log servers have been compromised. As we know from section 3.3, the publisher can specify which of them are involved in the registration process and thus determine the level of security. However, we believe that a domain owner willing to participate in our system has

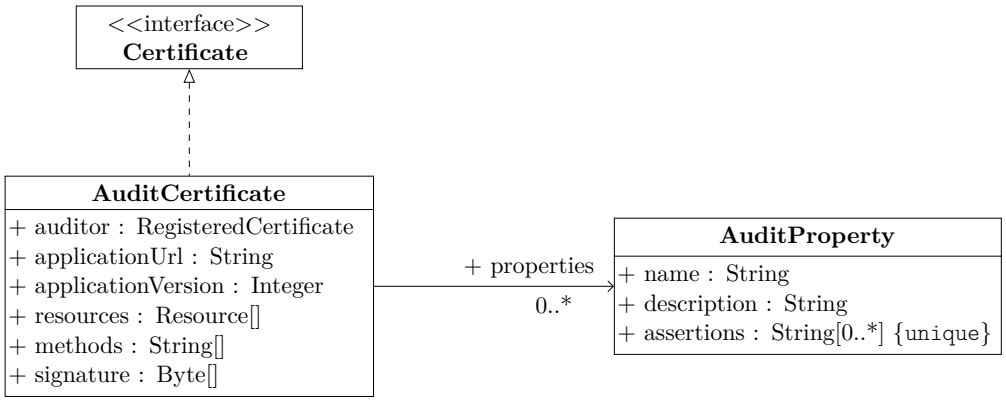


Figure 3.5: *The Audit Certificate holding the results of a code analysis.*

a great interest in providing as much security as possible and includes all available CAs and ILSes in their certificates. Of course, this is not the only feature that makes ARPKI the best choice. Besides attack prevention and a recovery mechanism, it guarantees operation legitimization and attack visibility as defined in section 3.1. It also fulfills a requirement we have not yet considered in detail. Because values can be appended to the configuration attributes of a certificate on each update, the resilience of the architecture scales with the number of authorities and log servers.

Although we can benefit from some core concepts of ARPKI, there are open issues that we need to address. Since it was designed for TLS, the trusted party is the publisher, not the client as in our case. Therefore, we need an additional query that allows the client to trust the publisher's behavior. Further, additional auditors must be included. The biggest change, however, is due to the fact that ARPKI forces CAs to maintain their own Merkle Tree. To verify a proof of presence or absence generated by the ILS, they have to synchronize their trees periodically with the log by replaying all modifications – if afterwards, the root hashes are equal, the proof can be validated. This is a disproportionate effort to ensure consistency as it requires additional requests, more processing power and potentially enormous storage capacity. Hence, we adopt some ideas of DTKI (whose properties have been formally proven) in our protocol and equip ILSes with verifiable log-backed maps instead of lexicographically sorted trees. As a result, a log server can create consistency proofs that can be verified by other participants if they have stored only a root they trust. Since this eliminates the need for CAs to operate a separate tree, it reduces the required computing and storage capacity. Further, the proofs are of small size and can therefore be directly included in the modification process of a certificate, making additional requests unnecessary.

Figure 3.6 illustrates the message flow of our protocol that results from these modifications. Because monitoring can be performed by any participant, separate third-party monitors are omitted for the sake of clarity. The steps (1) - (5) describe the registration process for the publisher, which also applies to an auditor. Afterwards, the publisher can query the audit confirmation (6) which they include together with the application fingerprint in the

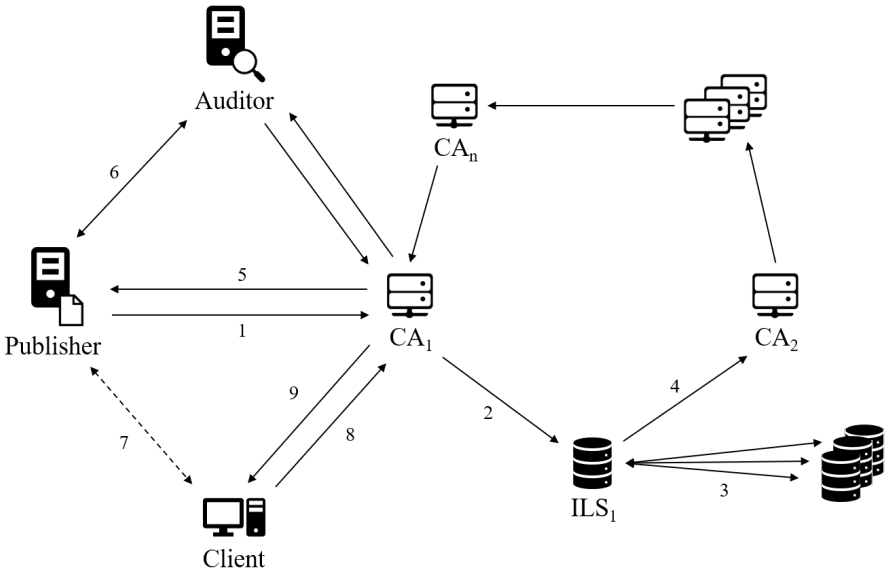


Figure 3.6: *The communication flow of the proposed architecture.*

web page served to the client (7). Finally, the end user requests the Application and Audit Certificate (8) - (9) to verify resource integrity. The individual steps will be examined in more detail in the following sections.

In general, the communication between two participants is always encrypted with the recipient's public key that is either known in advance or included in the request. Each message sent by a party who is responsible for a registered certificate (i.e. a publisher, auditor, CA or ILS) is additionally signed by its sender and, if different, the original creator. Since this requires that the public keys of these parties are known for authentication, we demand that at least the keys of CAs and ILSes are known to publishers, auditors and clients in advance. This actually seems to be enough to prevent impersonation attacks: Messages from both of them contain critical information like certificates and corresponding proofs which must be authentic, whereas the communication between publishers and auditors or clients basically yields fingerprints for comparison that are in turn signed by CAs and ILSes. However, an adversary could still succeed with a replay attack, e.g. to feign an invalid certificate as still valid. In contrast to ARPKI that cannot prevent such attacks, we include a nonce that is defined by the initiator of a request in each associated message and signed by every receiving party on response.

In the remainder of this section, we assume that all participants rely on the same collision-resistant hash function, existentially unforgeable digital signatures, secure encryption and decryption schemes, and that clients, publishers and auditors use the same encoding for resource digests. Further, CAs and ILSes are supposed to immediately stop the protocol run on failure (e.g. due to a lost message or an invalid verification result). Additionally, the latter have to revert any persistent change if subsequent parties in the run fail. Finally, we expect a CA to efficiently protect itself against Denial-of-Service attacks, while we do not demand this of ILSes. Nevertheless, they should also have a large bandwidth available.

3.4.1 Certificate Generation

As stated in section 3.3, a publisher must have their certificate signed by the CAs listed in the corresponding field to obtain a certificate that they can publish. We may have swept that under the table in the communication flow above, since this begins with the actual registration – so we catch up on the description here. Of course, this and the following procedures also apply equally to auditors – however, we only focus on the publisher in our explanations.

Regardless of its type, the publisher creates the certificate they want to register based on a static configuration file containing the ARPKI specific attributes. Obviously, the `signatures` field remains unset. Afterwards, they contact each CA listed in the corresponding sequence using a `GenerateRequest` depicted in Figure 3.7. After automatically validating the process related properties, they have to verify the domain attributes – while this is a manual step for Publisher Certificates, the other types can be verified and accepted automatically due to the publisher's signature. Finally, each CA signs the certificate in a `GenerateResponse` that is encrypted using the `publicKey` provided in the request. If the nonce and the corresponding signature are correct, the publisher combines all `certSignature` fields and appends them to the certificate.

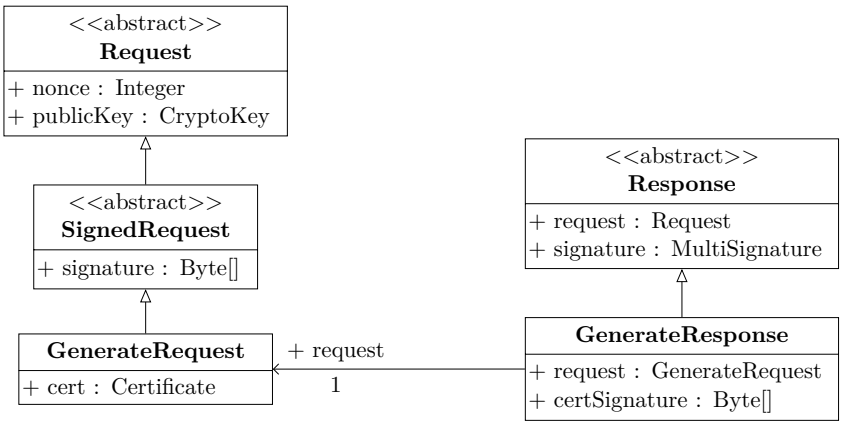


Figure 3.7: *The GenerateRequest and GenerateResponse message classes. All attributes are read-only, but the annotations were conveniently omitted.*

3.4.2 Log Modification

Now we are ready to follow the communication flow as shown in Figure 3.6. At first, the publisher has to decide what they want to do with their generated certificate – register it initially, update an existing one or even delete the current entry. Accordingly, they send one of the inheritors of a `ModificationRequest` depicted in Figure 3.8 to the first CA listed in the certificate field as the first step (1). Afterwards, the next stages of the process can be roughly divided into *Initialization, Validation, Synchronization, Modification, Proof Generation* and *Verification*. Note that, although both the auditor and publisher contact

the same CA_1 in illustration 3.6, each of them could have chosen any other CA arrangement.

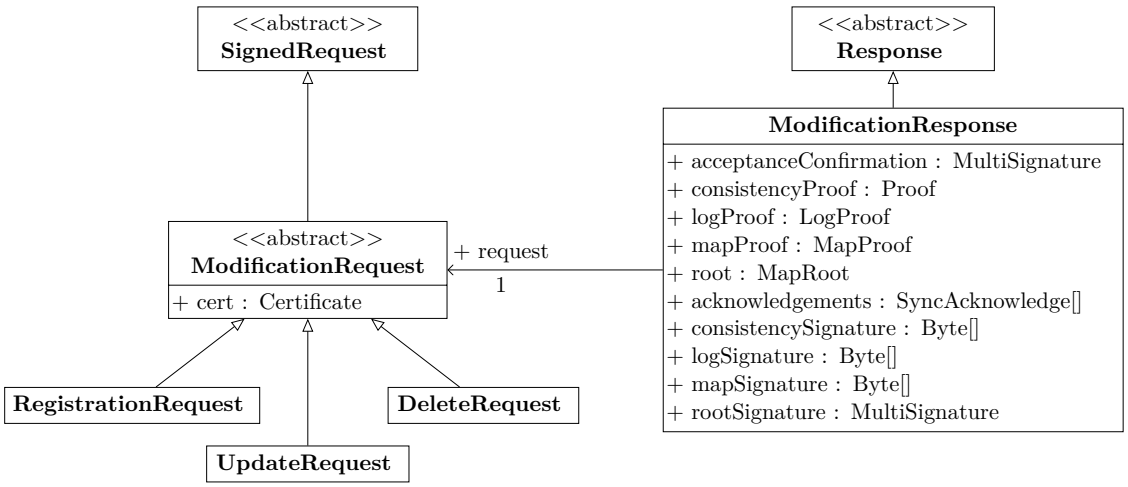


Figure 3.8: *The ModificationRequest and ModificationResponse message classes used for registration, update and deletion of a certificate.*

Initialization When CA_1 receives the message, it performs some verification steps to check whether the request is valid: First, it verifies the publisher's signature of the message, as well as of the certificate to authenticate the sender using the public key contained in the Publisher Certificate. Second, it examines the participants of the process: The `cas` sequence of the certificate must contain at least `caMin` ≥ 2 different entities, one of which is CA_1, and naturally the `ils` field must not be empty. Actually, this is enough to continue with modification, as the configuration fields of the certificate have already been accepted during generation. Therefore, CA_1 appends the request to a pending requests queue, signs the message from the publisher and forwards it to the first element of the `ils` list, which is ILS_1 in our case (2).

Validation Initially, the ILS behaves very similar to a CA when processing a modification request. It verifies the signatures and checks the fields for validity. Afterwards, however, it performs a lookup for the publisher's certificates of the same type and domain as the request certificate: For registration, there must be no entry in the log, whereas one has to be present for update or deletion (which is basically an update operation). Further, an Application Certificate requires the associated Publisher Certificate to be registered regardless of the operation, and an additional Audit Certificate needs both of them. The term *domain* may sound like nonsense in connection with a Publisher Certificate – still, we use it to indicate the unique key for the corresponding certificate. Application and Audit Certificates are naturally identified by the `applicationUrl` they were issued for, and the publisher can be identified by its `subject` name. If the validation steps have been completed successfully, the ILS creates the basis for the fingerprint that is later passed on

to the publisher, the `acceptanceConfirmation`. At this early stage, however, this only includes the signature created by signing the certificate digest.

Synchronization Before the ILS can proceed with executing the requested change, it has to check whether the publisher listed more than only one ILS in their certificate. If so, the modification should be synchronized beforehand, because it is a rather bad idea to modify the log first: Imagine that one of the ILSes to synchronize with declines the modification due to a (non-)existing certificate - if ILS_1 and, consequently, also all other contacted ILSes have already executed the operation, preserving consistency becomes a difficult task if we take network reliability into account, not to mention the huge amount of additional requests that would be necessary. Thus, our main ILS_1 sends a `SyncRequest` given in Figure 3.9 to the other ones which verify the request in the same way as ILS_1 earlier. On success, each of them stores the request in a queue and responds with a `SyncResponse` containing the request hash to inform the ILS that triggered the synchronization about its acceptance. Note that this digest is not accompanied by a signature, since in the direct communication between two ILSes, the overall message signature is enough for authenticity. Once ILS_1 has collected such responses from all the others, it notifies them sequentially with a `SyncCommit` that they can remove the queued request and make the actual change. This message contains the `acceptanceConfirmation` that is subsequently signed by each receiving ILS: After creating the `signature`, the contacted log server wraps the old confirmation in the `chain` field of the `confirmation` in its `SyncAcknowledge` response. As a result, ILS_1 receives a set of acknowledges, each consisting of a new tree `root` and the associated proofs. Obviously, the last `confirmation` is then signed by all ILSes that participated in the synchronization. For simplicity's sake, we grouped the requests and responses in step (3) of Figure 3.6.

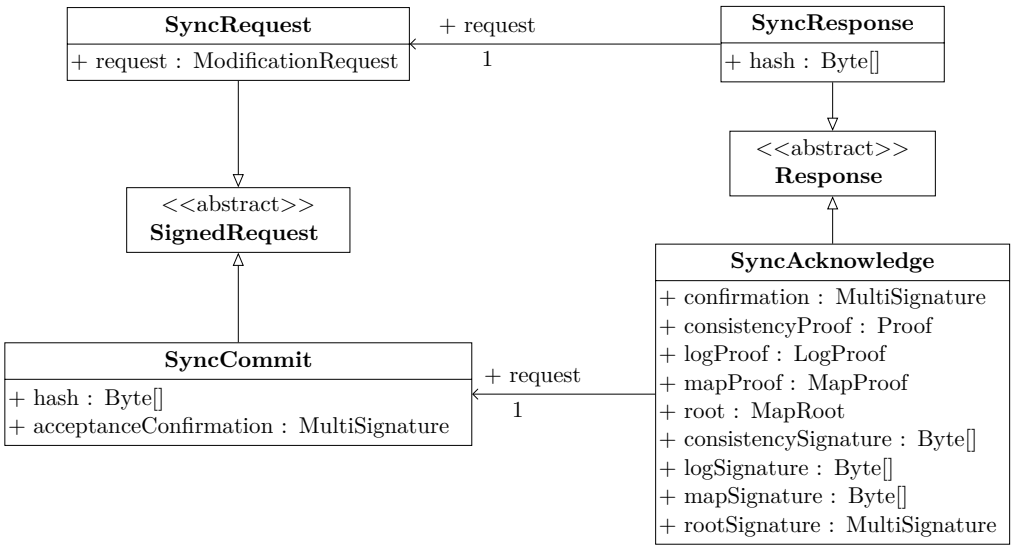


Figure 3.9: *The message classes used for synchronization between an arbitrary number of ILSes.*

Modification After the other ILSes completed their modifications and therefore the synchronization, ILS_1 performs the operation based on the data structures in Figure 3.10 as follows: It encapsulates the certificate with its domain in a `LogEntry` that serves as label of a `LogLeaf` for the append-only Merkle Tree described in section 2.3. Since this log is supposed to record modifications on the map, we additionally include the respective `Operation` in each entry. On insertion, the log server recomputes all intermediate node hashes up to the root and creates a new `LogRoot` instance for the incremented tree size. Afterwards, the corresponding map leaf can be changed. However, we could get into trouble here: As for example Application and Audit Certificates share the same domain, we recommend using a separate log-backed map for each of the certificate types to avoid unexpected domain collisions. As a consequence, this involves that CAs store the tree roots for each of them. Assuming that we evaded colliding domains, the ILS can create an `Entry` for the map which again consists of the domain and a certificate – in contrast to a `LogEntry`, the latter may be the empty constant ε instead of the certificate upon deletion. Using this as label, it builds a `Leaf` that is added to the Sparse Merkle Tree. Its index (i.e. the associated key) is simply the digest of the certificate domain. Then, the root hash can be calculated and combined with the `LogRoot` in a new `MapRoot` revision.

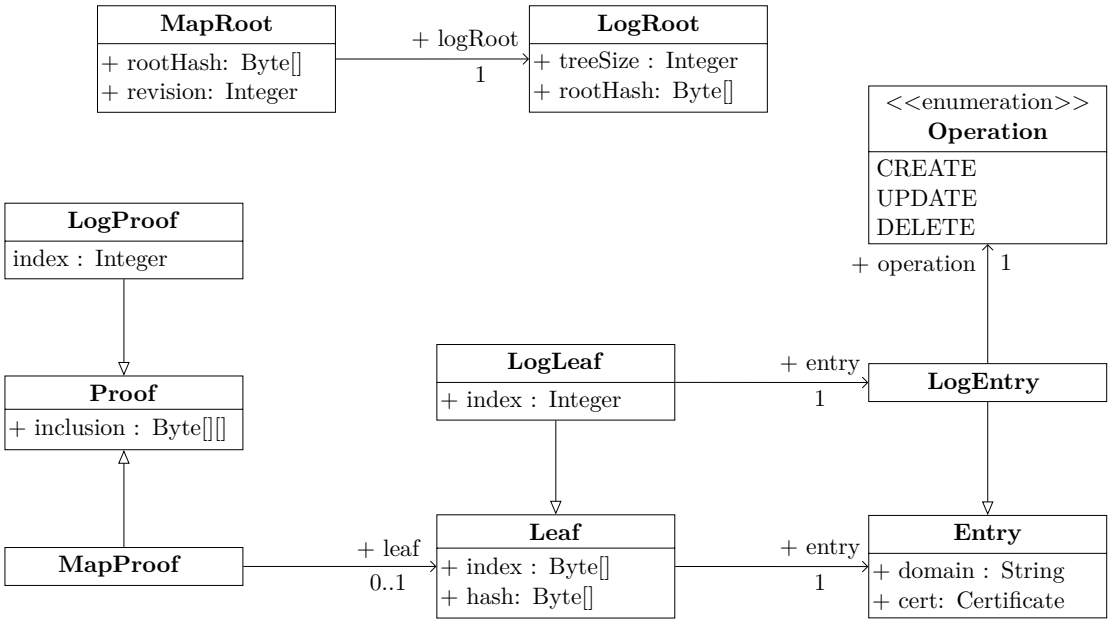


Figure 3.10: *Data structures provided by a log-backed map for verifying inclusion and consistency.*

Proof Generation Finally, ILS_1 can construct the `Proof` instances for verifying consistency between the last and current `LogRoot`, as well as inclusion of the certificate in both the log and map. Based on the findings in section 2.3, we know that a consistency proof is a sequence of digests that prove the existence of a known former root in the tree, while

inclusion proofs additionally include the index of a leaf. However, remember that we aim at passing on the proof of inclusion of an entry in the map to the client as well. Hence, a `MapProof` contains the whole leaf instead of just the index if it exists – as stated in the last chapter, we have to distinguish between empty and non-existent leaves, which is why they could be absent from the proof in the latter case. Nevertheless, the proof structure remains the same for the underlying log: Extension is proven by a `Proof` instance, whereas a `LogProof` allows to verify the existence of an entry.

For its final `ModificationResponse` (see Figure 3.8), the ILS signs each of the proofs and the new `MapRoot`, as the synchronization acknowledgments of other ILSes are already signed by them. Further, it includes their final `confirmation` (or only the initial `acceptanceConfirmation` if there were no ILSes to synchronize with) and embeds the root signature in a `MultiSignature` to guarantee that a malicious CA cannot fake the root and proofs, because such signatures have to be verified and signed by all CAs. Instead of responding back to CA_1, the response is sent to the next authority in the corresponding certificate field, indicated by step (4) in Figure 3.6.

Verification Now it is time for CA_2 to check the actions of the previous participants. First, it performs the same steps as CA_1, except that it examines the response from ILS_1. This implies that each signature in the response has to be verified - especially every `MultiSignature` recursively. Second, it builds the `LogLeaf` that is expected from the modification and verifies the proofs for each ILS: After ascertaining that the claimed new root is indeed an extension of the one already stored by the CA, the leaf is used for ensuring that the requested operation has been executed and the correct entries are included in the log-backed map. On success, it saves the new root for future use and approves the request by singing and therefore extending each `MultiSignature` just like the ILSes described above. Then, it forwards the updated `ModificationRespone` to the next CA (which can actually be CA_1 if only two authorities are listed in the certificate).

Since the next CAs in the ring topology behave exactly like CA_2, they verify the actions of their predecessors. After CA_1 approved, it can remove the request from its queue and respond to the publisher with the final `ModificationResponse`, represented by step (5) in our communication flow. As both the root and confirmation are signed by each participant, the requester knows that the modified entry matches the requested modification and the proofs are authentic - by verifying them, they can be sure that the process has been executed correctly.

However, we have omitted one detail here: The publisher needs an already saved root for consistency verification – unfortunately, this must be the same as the previous revision of the ILS, because the corresponding proof only guarantees consistency between the last and current root. A closer look reveals that this even applies to CAs: If some of them have not participated in a modification, they cannot verify that their version and the provided new root are consistent. Of course, CA_1 could ask every CA listed in the certificate field for their current revision and forward them to the ILS in step (2), so that a proof for each of them can be included in the response (4). Nevertheless, we are optimistic: While we

suggested above for security reasons to list as many CAs as available in a certificate, we renew this recommendation at this point to need as few requests as possible. If just a very small number of them is not involved in most modifications, then a large majority of them will hold the current root afterwards. Therefore, only the CAs that have not participated need to ask for it during the next run – obviously, in the worst case, this results in the same amount of requests as the prior collection of all revisions – if our suggestion bears fruit, there will be far fewer. Thus, when receiving a `ModificationResponse`, each CA holding an older root sends a `RootRequest` depicted in Figure 3.11 for the `type` of the modified certificate to obtain the last version before the modification and a consistency proof from the `ils`. Just like for the hashes exchanged during ILS synchronization, there is no additional signature required for the root and proof, since the overall message signature suffices in direct communication. After verifying the proof and updating its root, the CA can continue with the protocol steps - checking correct execution of the operations that led to this revision is not necessary because the `cas` listed in the response have supervised the process and can witness the digest.

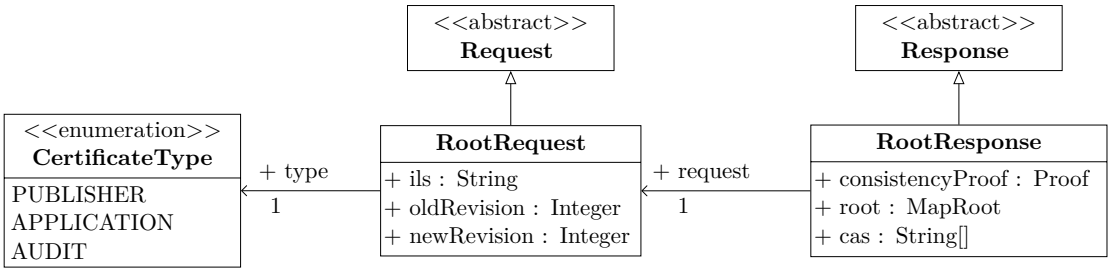


Figure 3.11: *The RootRequest and RootResponse messages to verify and update to a new root.*

In summary, these are quite a lot of steps to publish or delete a certificate. To illustrate the steps and to avoid confusion, a sequence diagram for Log Modification can be found in Appendix A. We consider a publisher *pub* with public key P that wants to register a certificate with n CAs with keys $C_{1...n}$ and m ILSes with keys $I_{1...m}$. For simplicity, we denote the validation of process related certificate attributes as `verifyParties` and the verification of participant signatures as `verifySignatures`. Further, we omit verification steps of message signatures as well as decryption. On that basis, Appendix B also includes a sequence diagram showing the following resource verification process initiated by a client *clt* with public key K.

3.4.3 Resource Verification

After the publisher generated and registered an Application Certificate, they must pass the fingerprint on to the client. This is rather simple: they sign the `acceptanceConfirmation` received from CA_1 and append the result together with the lists of CAs and ILSes to the

index HTML document of their application. If an auditor has already created an Audit Certificate for the application, the client must also be informed. Therefore, the publisher requests the signed `acceptanceConfirmation` that the auditor received on registration using a `SignatureRequest` (see Figure 3.12) shown by step (6) in Figure 3.6. The response content is added to the HTML document in the same way as the Application Certificate fingerprint.

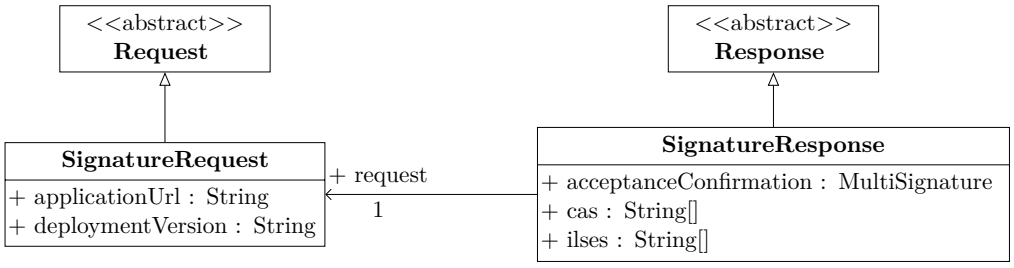


Figure 3.12: *The SignatureRequest and SignatureResponse message classes to query the signed confirmation of an AuditCertificate.*

Proof Retrieval After retrieving the web application files by a common HTTP request in step (7), the client starts with querying the appropriate Application Certificate from the given CAs. For their `GetRequest` depicted in Figure 3.13, they randomly select an ILS from the list provided by the publisher (coincidentally, it is ILS_1) or, if they want to, all of them in a row. Then, they contact the first CA in the sequence (8) – it is the same as the one contacted for registration, namely CA_1.

The CA stores the request in a queue and forwards it directly to the ILS. From now on, we benefit greatly from the heavy verification during log modification. As mentioned in section 3.2, the client usually only checks the inclusion of a certificate in the map, which is sufficient since the CAs have already taken care of consistency and traceability by verifying the modification process. Therefore, ILS_1 generates only one proof of presence or absence, depending on whether a certificate is currently registered or not. In the former case, it must additionally verify that the CAs specified in the request and in the certificate match. If not, this is an indication of an attack, as is the latter case with a non-existent certificate. Afterwards, the ILS signs both the proof and the current root and sends them all in a `GetResponse` to CA_2, because all authorities involved in the registration must also verify the result.

This is also straightforward: Each CA verifies the signatures in the response and simply compares the stored root to the received one. If they are equal, the certificate must be valid because the current root has already been verified during modification. Note that we do not have to verify any proof here – fortunately, since we would need the log inclusion and consistency proof to verify anything at all. If the roots differ, the CA updates its copy using a `RootRequest`, which only requires consistency to be proven. Since we assume that nearly all CAs are involved in most modifications, such requests are not only rare, they

also remain small in size as the last observed revision will not be too long ago on average. Thus, the actual verification of the `GetResponse` will be quite fast. After signing each `MultiSignature`, the response is passed on to the next CA in the ring and finally to the client (9).

Note that the request contains a `type` attribute. In fact, this procedure applies to any certificate type and is hence used by CAs as well: When validating domain attributes during certificate generation, they have to ensure that the version is an increment of the already registered certificate, or that the specified Publisher Certificate is indeed the one included in the current map. Therefore, they make use of the described method for proof retrieval to securely query the certificate from an ILS.

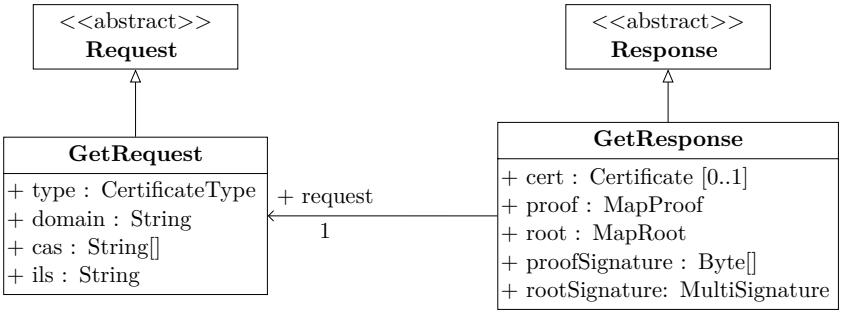


Figure 3.13: *The GetRequest and GetResponse messages to obtain a certificate.*

Verification Upon receiving the response, the client verifies the included signatures first. Subsequently, the certificate is cross-validated by verifying the `acceptanceConfirmation` using the publisher's public key contained in the certificate and the built-in keys of the participating ILS and CAs. Finally, they verify the map inclusion proof to ensure that the certificate indeed exists under the root that has been considered valid and therefore signed by all participants. After validating the certificate attributes, the only thing left to do is compare the resource hashes with those calculated from the downloaded files – if successful, the client can securely trust the participants that the resources have not been altered for a targeted attack (assuming that at least one party has not been compromised). The procedure must only be executed when the client initially accesses the application since a certificate can stay valid until the session ends. Nevertheless, keep in mind to verify resources that are downloaded during runtime when implementing the protocol.

To provide a higher level of security, we included Audit Certificates. To obtain the associated one, the client performs the same steps for proof retrieval and verification as described before, with the difference that this time, it is based on the auditor's fingerprint included in the HTML index file. Thereafter, one can trust the security properties stated in the certificate and specified assertions can be verified (continuously, if required).

3.4.4 Monitoring

Like in ARPKI, we rely on monitors to verify global consistency among all ILSes. Unfortunately, neither ARPKI, nor any of the other reviewed PKIs using log servers provide a description of how monitoring works. Obviously, we need the log entries of each ILS to reconstruct its map – ideally without having to build or even store both Merke Trees. Nevertheless, we would like to incrementally verify new changes. While the former requirement reduces the necessary storage space, the latter decreases the size of network messages. However, the less we store, the more we have to ask for – still, we only persist minimal information per ILS (i.e., only the current root) and accept larger responses. Since we assume that dedicated monitor servers execute the process quite regularly to verify global consistency, and that clients are mostly willing to monitor a particular ILS instead of all at once, the response messages should remain of acceptable size.

Effectively, the monitoring request process is similar to the proof retrieval procedure described in the last section. The participating CAs only agree on the root given in the response and verify the included signatures. Therefore, the remainder of this section skips these steps and focuses on the message structures and the verification by a monitoring party. A `MonitorRequest` illustrated in Figure 3.14 contains the same attributes as a `GetRequest` in addition to the root `revision` that the monitor has stored and currently trusts. The ILS includes a sequence of all new `leaves` of the append-only Merkle Tree that have been added since this version in the `MonitorResponse`. For verification purposes, they are accompanied by the current `root` and the corresponding proofs of inclusion in the log and map.

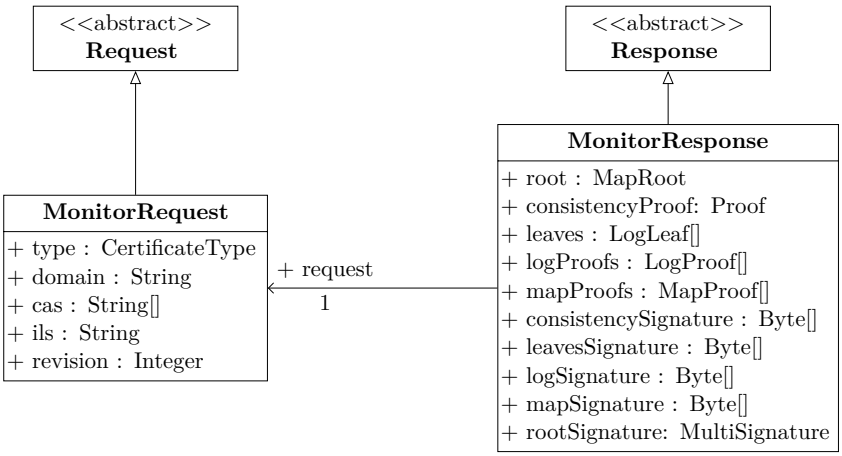


Figure 3.14: *The MonitorRequest and MonitorResponse message classes for verifying correct operation execution and (global) consistency.*

After verifying each signature in the response, a monitor checks the consistency between its current root and the one provided by the ILS. Subsequently, it starts to verify the log inclusions. Here we face a problem: proofs of inclusion guarantee that each entry we received is included – however, in general there could exist more new leaves about which we have not been informed. Hence, we extend the verification of inclusion proofs by

iteratively calculating the tree root before the insertion of a leaf. As the response contains the leaves in order, we start by "removing" each entry from the end of the list. If the ILS actually sent us all the new leaves, the computed root hash should equal the stored root that is already trusted. According to [110], we could in fact remove an arbitrary number of entries from a tree of size N and use the inclusion proof $(p, [b_1, \ldots, b_k])$ of the remaining leaf b_p with highest index p to compute the root hash of a tree M_p with equal size: Since b_p must be the most-right node in M_p due to the structure of an append-only Merkle Tree, any right sibling of b_p or its ancestors that is included in $[b_1, \ldots, b_k]$ must be ignored. Moreover, the nodes labeled by the remaining digests themselves are roots of perfect subtrees in the current tree by definition, and therefore must have already been perfect in M_p as they have an index smaller than p. Thus, we prune the inclusion proof by $prune(path(p, N), [b_1, \ldots, b_k])$ which is defined as

$$\begin{aligned} prune(\varepsilon, []) &= [] \\ prune(\rho \cdot \texttt{left}, [b_1, b_2, \ldots, b_k]) &= prune(\rho, [b_2, \ldots, b_k]) \\ prune(\rho \cdot \texttt{right}, [b_1, b_2, \ldots, b_k]) &= b_1 :: prune(\rho, [b_2, \ldots, b_k]) \end{aligned}$$

where :: and *path* denote the same operations as in section 2.3.3. As a result, we obtain the ordered left siblings of the rightmost node b_p and its ancestors in M_p, which obviously form the inclusion proof. This implies that we can compute the root hash of M_p by

$$H_{chain}(path(p, p), b_p, prune(path(p, N), [b_1, \ldots, b_k])) \quad .$$

By recursively pruning the tree by one entry, a monitor can eventually ensure that the received root is derived from the trusted one by exactly the given leaves – of course, the original inclusion proofs still have to be verified.

Based on the verified modifications described by the log leaves, one can build a snapshot of the map section that results from them. After verifying the map inclusion proofs in the response, the monitor checks that this snapshot contains only the entries specified in those.

Then, all operations that have taken place since the last call of this procedure and their results are verified – since we store the new trusted root, we reduce the response size and proof effort due to incrementally verifying all operations. However, a monitor only gains knowledge about the changed certificates in each run. Therefore, we recommend that a monitor stores the map entries locally. For dedicated monitoring servers, this helps in checking global consistency: the certificate maps of the ILSes must remain in a consistent state, i.e. a specific certificate may be present or absent among all of them, but an older version is never still present in any. Further, the maps can be merged and provided to other parties. For clients, this could enable an extended caching mechanism for certificates – if the browser periodically monitors at least some ILSes which, for example, hold certificates for applications that the client frequently uses, it could persist some of the entries as they remain valid until the `MonitorResponse` contains a new certificate for their domains. While this may cause computational overhead compared to the resource verification process described before, it could preserve client history privacy since no CA or ILS can gather knowledge about the particular domains a client visits.

3.5 Security Analysis

Since our goal was to design a security system, we will now analyze it in terms of security. But before, we should clarify what can be assumed in addition to the conditions we declared in section 3.4. First, we demand that our client is able to intercept any incoming and outgoing connection, as otherwise, neither lazily downloaded resources could be hashed, nor assertions claimed in Audit Certificates could be verified if they target network requests. Second, we expect each honest participant to adhere closely to our protocol – this also requires that it has been implemented correctly without vulnerabilities and includes a correct implementation of the log-backed map maintained by the ILS. Last but not least, we assume that all parties are informed if misbehavior is detected. As before, the introduced adversary model defines the abilities of an attacker.

Since we rely on the use of a collision-resistant hash function for computing resource hashes, we limit the attack scope to faking certificate information, as the client and their browser are not compromised according to the model. Given our attributes from section 3.1, we thus focus on attack prevention and visibility, for which legitimization is a requirement and tamper protection an implication. This results in two categories of adversaries: An *external* attacker has access to the network communication and can read or modify it. However, they are not a real participant of our infrastructure. In contrast, we consider an infiltrator as *internal* attacker, since they have knowledge about the private key of at least one participant. Of course, a party that is malicious in itself also belongs in this category. According to our assumptions, the former allows Man-in-the-Middle and Denial-of-Service attacks on participating entities, while the latter makes it possible to circumvent protocol steps or to actively alter content, so that targeted attacks are much more difficult to prevent. Note that we exclude replay attacks from both categories: Due to the nonce included in each request, neither an external adversary, nor an internal attacker is able to reuse previous responses, since they must contain a signature of the number from every participant. Moreover, we consider timed-release attacks as a subset of attacks based on invalid certificates because they are characterized by an illegal `validity` attribute with respect to the `expectedLifetime` of a certificate. Thus, we only take into account their superset and will not mention them further in the remainder of this section.

3.5.1 External Attacks

As mentioned above, the public keys of CAs and ILSes are known to publishers, auditors, monitors and clients. Therefore, we call the former *static entities*, while the latter are *dynamic entities* as they all are end users of services the core infrastructure provides. Since we expect digital signatures to be existentially unforgeable, the sender of a `ModificationRequest` can be reliably identified and the recipient can verify its integrity: if the sender is a static entity, its key is known in advance, while the public key of a dynamic entity authorized for modification is located in a signed Publisher Certificate within the request. Further, the message is securely encrypted with the recipient's public key. Thus, the communication between static entities, as well as between a static and a dynamic participant is secure against traditional MitM attacks. Regarding messages

between dynamic entities, this is not the case – if an adversary intercepts a request, they can fake the response content any way they want. Nevertheless, they cannot attack the client: Consider the `SignatureRequest` and its response between a publisher and an auditor. The adversary blocks the request and responds with a bogus fingerprint. Then, the publisher queries the inclusion proof of the corresponding Audit Certificate, which reveals the attack if such a certificate has not been registered previously. However, if a certificate exists, we assume that at least one attribute contains invalid content, for example an unfulfillable assertion. After the publisher validated the attributes and figured out which is incorrect (we assume that they know the security properties of his application), they even know the attacker's identity, since the adversary must own a registered Publisher Certificate to register an Audit Certificate. Obviously, the verification and validation can likewise be performed by the client, and the same reasoning also applies to the communication between publisher and client, since modified confirmations included in the HTML document are equivalent to faked fingerprints. In case of removed static entities provided together with the `MultiSignature` (additional ones do not weaken security), the signature check itself already fails.

The other type of attack we have identified is not aimed at disguise, but rather at disruption through tons of requests collapsing a server. However, there is no point in attacking the publisher's web server with the intent of actually attacking the client. Since an auditor is only involved by providing a fingerprint for their certificate, unavailability would only result in assertions that are not verified while resource integrity could still be ensured. Hence, we focus on static entities which provide critical functionality. First, remember that we require every CA to protect itself against DoS attacks. Under this assumption, one up to all ILSes can collapse under heavy load, whereby only the last case is interesting: the client randomly queries an ILS out of the list provided by the publisher, and retries this with another log server if it is unavailable. What remains is to specify the client's behavior if requests to all static entities (given along with the fingerprint) fail, as one CA that is not available results in the same as if all static entities deny service due to the ring topology they form when processing a request. The answer is simple: We cannot consider the application secure because we have nothing to prove it with. However, if the assumption holds such that all CAs and at least one ILS can protect themselves against DoS attacks, the infrastructure remains intact.

Apparently, our architecture protects against external adversities. Nevertheless, the analysis has not yet taken into account the fact that we may well have malicious or at least infiltrated participants. If we involve them in our considerations, the investigated attacks could result in dangerous combinations. But let us keep in line and have a look at how compromised parties affect the security of our system.

3.5.2 Internal Attacks

Due to the number of different participants and their duties, it becomes a challenging task to identify a compromised entity and prevent it from executing or enabling a successful attack. To cope with complexity, we only consider the parties necessary for satisfying our core security property. These are, besides the client, the different CAs and ILSes

that are listed in the corresponding certificate fields, as well as the publisher. Further, we split the protocol in its three main parts *generation*, *modification* and *verification*. We omit monitors here because they are not an integral part of detecting altered resources – nevertheless, since monitoring equals proof retrieval to a great extent, our analysis covers the security of this procedure.

Generation In case of only a malicious publisher, we can quickly deal with the first protocol step, as there are two possibilities: A bogus Publisher Certificate can contain invalid values like a wrong unique name or domains that the publisher does not own. However, these are identified by the CAs through manual validation. Therefore, we focus on the second one: Since most attributes of an Application Certificate are also validated by CAs, only the `resources` collection can be used for an attack. Yet, there is no way to force a targeted one on a single client, as the certificate has to be registered in order to obtain a fingerprint that is signed by all static entities. Thus, the altered resource files must be delivered to anyone accessing the application which in turn means that it can be checked by security analysts. The attacker could be identified and blamed without a doubt, and we expect them not to want that.

If one or more CAs are compromised but not the publisher, it is possible that the certificate could not be signed correctly. As this is verified by the publisher, they immediately detect the misbehavior and can identify the CA. The worst case, however, is when all CAs have been infiltrated and a third party certificate is issued for a domain that actually belongs to the publisher. Since this has to be registered to attack the client, the legitimate domain owner can query the ILS that stored the certificate in an accountable manner and detects the trespass of CAs and the third party.

Next, we proceed with the combination: Consider a malicious publisher and that all involved CAs are compromised because otherwise, at least one CA would still detect the invalid attributes. Then, a bogus certificate can be generated, and we expect the resources or the public key to be faked, as an invalid domain or subject name can be detected by the actual owner. This allows a third party to impersonate the publisher's identity, but still results in a publicly available registered entry – like the cases before, there is only a chance for the adversary if they bypass the modification or verification procedure.

Modification The participants that are actively involved in the modification process are static entities, so we start off by examining the impact of a compromised CA in the *Initialization* stage. Apart from being able to skip the validation steps, it could also simply not forward the request – even if the publisher did not verify the response, the client would at the latest notice that the required signatures are missing or the provided lists of CAs and ILSes are outside the permitted ranges. Note that the CA itself cannot alter or fake the certificate, as it must always be signed by the publisher, and even the message has to contain their signature. Therefore, we also take into account the *Verification* stage of the process and consider multiple malicious CAs. If we assume that the ILS is honest and sends a valid `ModificationResponse`, there are not many options: refusing to sign the `MultiSignatures` or interrupting the whole chain causes a verification error or a timeout

at the subsequent CAs (which can then obviously identify the dishonest party). Even if they are all compromised, the result is the same as if the first CA didn't forward the request.

Hence, we should first take a look at the effects of a malicious ILS and assume all CAs are honest for now. Theoretically, an adversary could force our system to forward a bogus certificate: imagine that faking a `GetResponse` sent to CAs would be possible, the ILS could receive such a certificate although, for example, the required Publisher Certificate is not registered. Let us put this aside and assume a valid certificate in the following, since we discuss the precondition in the next paragraph. Of course, the log server could simply create a fake certificate by itself – however, the CAs would not sign the resulting `acceptanceConfirmation` due to an invalid (because non-existent) request. Either way, the ILS obviously skips its own *Validation* phase. Note that it cannot modify or fake an honest `SyncAcknowledge` by another log server without knowledge of its private key. The only possible misbehavior regarding *Synchronization* is therefore to lie about the actual modification: If the malicious ILS receives a `SyncRequest`, it can copy the current Merkle Trees, modify the copy according to the request, respond with a valid acknowledge containing the proofs of the copied trees and discard them once the procedure is over. While registering a different bogus certificate is detected by the verifying entities in the next process stages, pretending to correctly modify results in a wrong root hash that is stored by the CAs. Due to the collision-resistant hash function, the proofs cannot be faked and the digest must be valid to be accepted. Afterwards, the misbehavior of the ILS becomes obvious with the next request, as the roots differ and cannot be proven to be consistent through a `RootRequest`. Fortunately, this explanation also covers *Modification* and *Proof Generation*, since these include the same actions as *Synchronization*.

Moving on to the combination of different malicious static entities, we know from the analysis so far that for a successful attack, an adversary needs to infiltrate the ILS and prevent it from validating a bogus certificate while forcing it to create a valid `acceptanceConfirmation`. Further, all CAs listed in the certificate must be compromised, so that they also skip the validation (and maybe even the verification) and sign the `MultiSignature` received from the ILS. Of course, this is already required for generation since they are responsible for signing the faked certificate. Together, they are able to modify entries without a security alert – however, they still need to convince the client.

Before we continue with investigating the conditions that must be fulfilled to deceive the client, we want to stress that, fortunately, the requirements for the described attack scenario conform with the security property of ARPKI. In fact, the system parameter n that we referenced in section 2.2.4 denotes the minimum number of CAs and ILSes respectively required to register and to store the certificate. While we only mentioned that one ILS must be compromised, we can easily generalize our considerations: Imagine that the system requires a fixed number of ILSes to sign a certificate. Then, the client would discard the bogus certificate since it is only signed by the malicious log server - thus, the adversary has to conquer and synchronize as much ILSes as the system parameter demands.

Verification Although we have already figured out that compromising all CAs and at least one ILS to successfully attack the core of our infrastructure, we should not let ourselves be disoriented and proceed step by step. We still cannot be sure that this always affects the client, and must examine that not even less is necessary for a successful attack. Consequently, we assume that only the publisher is malicious, but there is a valid certificate registered. Since the signature verification fails if the publisher provides a faked `acceptanceConfirmation`, they could at most omit the fingerprint completely. But then, the client considers the application not secure by default. Hence, the publisher could only launch an attack by publicly registering a certificate for the altered resources which can be detected and makes them accountable.

If the publisher is honest, but one or more CAs are not, they could only refuse to forward the `GetRequest` or `GetResponse`, respectively. After the client notified the domain owner and other parties, the misbehaving CA can be identified. If we instead assume that the ILS is compromised but not the CAs, it could either respond directly with a bogus certificate, or create a copy of the map by removing or replacing the requested one. Even if the fake certificate is signed by CAs, both would immediately be detected: the former due to an invalid inclusion proof, the latter due to a different root hash. Note that the consistency proof of a `RootResponse` alone is not enough, but since the CAs that authorized the last change must be listed, they can be queried for a confirmation. Based on the identified attack scenario, we can omit these queries if the intersection of the `cas` in both the `RootResponse` and `GetRequest` contains more than `caMin` certificate authorities. In summary, this means that a malicious ILS cannot fake a `GetResponse`, which therefore makes it impossible to force CAs to forward a bogus certificate for modification.

Finally we can pick up where we left off in the paragraph about modification. If we expect all static entities involved in the `GetRequest` to be compromised, they can obviously serve a bogus certificate or affirm that none exists. Nevertheless, once the client compares it to the fingerprint they received from an honest publisher, the attack and the conspirators are detected – therefore, also a malicious publisher is required to successfully inspire confidence. However, the ILS has to copy the tree as described and the CAs have to keep the roots of the original trees to maintain pretence. Otherwise, a monitor could detect the attack afterwards. This implies a chance for the client: If they trust one or more dedicated monitors, they can look up an "external" view when receiving a certificate, which forces the adversary to compromise them as well.

3.5.3 Combined Attacks

To conclude this section, we reconsider the possible external attacks that we already discussed together with the assumption that one or more participants are compromised. Our results are still valid for most scenarios, except for the only internal attack scenario we found: Although we stated that, for a successful attack, both the static entities and the publisher must be malicious, a MitM attack by a third party on an honest publisher suffices to pass a faked fingerprint to the client. Since the bogus third-party certificate is not necessarily registered, neither the publisher nor monitors are able to detect the attack.

Furthermore, assume that a previous version of a certificate was issued for a vulnerable

application, but has already been updated to a secure version. For security reasons, both have been stored on multiple ILSes. Now, the adversary wants to deliver the old application to exploit the known vulnerabilities. If they control the publisher, CAs but only one ILS, they are able to behave well towards other parties and to provide the fingerprint of the old certificate to the client. Nevertheless, they must rely on chance that the client chooses exactly the compromised log server to query the certificate. However, this choice can be forced: If the remaining ILS cannot respond because of DoS attacks, the client always requests the last one left.

Since the former scenario based on a MitM attack targets the publisher, our level of security is unfortunately not higher than that of ARPKI, but at least cannot drop below it. This also applies to the latter, as ARPKI does not include more than one ILS in their proof responses, and is reflected by the comparison of the three categories in Table 3.1.Nevertheless, a resource alteration attack through our architecture requires more extensive and sophisticated actions if additional common security mechanisms are applied. In section 3.7, we discuss how security can be further improved.

	External	**Internal**			**Combined**
		Generation	**Modification**	**Verification**	
Publisher	×	1	0	1	0
CA	×	c	c	c	c
ILS	×	×	i	i	1

Table 3.1: *Entities that must be controlled for a successful attack on the client, out of one publisher, c CAs and i ILSes that are listed in the certificate.*

3.6 Implementation

We provide a prototypical implementation of each participant and a small application to demonstrate the protocol on GitHub[1]. All components are written in TypeScript, and all but the client host a Node.js server. During their implementation, we have taken care to meet our quality attributes: since we wanted to enable simple deployment, the entities have been virtualized using Docker[2] containers and can therefore be configured using a few environment variables.

Of course, the client implementation is not wrapped in a container – it is currently delivered within an extension for Google's browser Chrome. Multiple extension APIs make it possible to query the content of resources for multiple frames at each tab update and incoming request. Further, the DOM and therefore the included fingerprint together with the CAs and ILSes can be obtained, which in turn allows to request the certificate in background. After proofs and fingerprint have been verified and the resource digests

[1] https://github.com/Placc/CRAP
[2] https://www.docker.com

compared, a pop-up icon that displays the current process stage enables the user to view detailed certificate information. Currently, only a simple assertion for the absence of `eval` calls is implemented for demonstration – however, any assertion is continuously validated as long as a certificate stays valid (i.e. throughout the tab session).

For the Merkle Trees of the ILS we use Trillian [110], a reference implementation of the verifiable data structures described in [90], because it supports Sparse Merkle Trees in map mode and append-only Merkle Trees as log. Thus, the desired log-backed maps for each certificate type can be composed by multiple tree instances that are maintained by the ILS. It is implemented as a gRPC[3] service and provides a rich interface definition in Protocol Buffers[4] that can easily be compiled to TypeScript. In log mode, a dedicated component periodically fetches a batch of queued leafs and appends them to the tree. While this may cause a significant delay for only a few insertions, it scales to handle a huge amount of requests. Unfortunately, only the tree service is provided as Docker container, and we had to imlpement client functionality such as verification in TypeScript by hand.

Finally, we use the most probably collision-resistant [39] hash functions SHA-256 for the calculation of resource digests and SHA-512/256 for the log-backed map, while RSA-2048 serves for digital signatures.

Deployment Aspects Note that also the publishers and auditors host a server for protocol handling in a container. We expect publishers to run this component on their build server, as this allows the protocol to be performed elegantly: After the build process of the application has finished, a one-line command suffices to execute a thin publish script (which we provide in TypeScript, but can easily be adopted for other languages) which reads a static configuration file containing the process related certificate attributes, computes the resource hashes and contacts the server component to execute the modification process with the received information. Then, the server looks up the Publisher Certificate in its local database. If there is none, the generation procedure is initiated. Otherwise, the final Application Certificate is constructed, sent to the CAs to obtain their signatures and finally forwarded to the first CA for modification. On success, the script automatically includes the `MultiSignature` together with the lists of CAs and ILSes in the specified index HTML file. If the application has been audited, it queries the auditor server and adds the resulting fingerprint in the same way. The auditor component has to execute similar steps to publish a certificate that are also initiated by a simple script, but should be always available for publisher requests. Nevertheless, the automated use of both the auditor and publisher components together with the easy configuration and setup due to virtualization satisfy the key aspect of our deployment requirement.

Application The demonstration scenario is about a publisher who wants to register their certificate with two CAs and one ILS. Therefore, each participant is visibly launched in a separate shell, and especially the publisher provides detailed explanations to the user. After they built a small web application (which contains only a simple login form that posts

[3] https://grpc.io
[4] https://developers.google.com/protocol-buffers

the hashed password on action) and registered the corresponding Application Certificate, the user can access the website in a browser with installed the extension. Further, the user can advise the publisher to alter the page resources such that the password is transferred in plain-text – then, the extension should indicate an attack. Alternatively, the publisher can contact an auditor to "audit" the application and register an Audit Certificate which is afterwards also visible in the extension pop-up and allows to automatically verify the assertion mentioned above.

3.7 Discussion

Our analysis in section 3.5 has already shown that the proposed architecture requires at least as many participants to be compromised as ARPKI for a successful attack. If we assume a secure connection between publisher and client, the former must also have malicious intentions. Finally, in contrast to other PKIs, the required level of compromise can be adjusted to the number of participants due to configurable policies like in ARPKI. However, we have neither discussed if and how our system could be improved with respect to the other characteristics that we compared in section 2.2.4, nor have we taken into account economic considerations for practical use.

History Privacy In terms of the compared features, our architecture is mainly consistent with ARPKI. We provide effective attack prevention, means for revoking certificates or updating them in case of key loss and prohibit multiple certificates for a domain. Nevertheless, there are some differences: While this work lacks a formal proof, it extends ARPKI by a more sophisticated authenticated data structure that supports proving of consistency between multiple versions. Further, we require extra client communication to query the ILS, which is not necessary in ARPKI since they trust the publisher. Unfortunately, we leak the client's browsing history through this connection. The log server is able to learn the registered applications that the client visits – although we do not regard it as relevant to achieving our goal, this weakness can easily be overcome: Instead of requesting the full domain, a `GetRequest` could include only a hash prefix of it. This corresponds to an inner node of the Sparse Merkle Tree and therefore results in multiple possible leaves which are sent back to the client. Depending on the number of certificates stored, the prefix length could be adjusted so that generally the same number of possible certificates is returned and the ILS cannot identify the actual domain.

Economic Incentives Another vulnerability we already mentioned and which also exists in ARPKI is the possibility of split-view attacks: while several compromised parties could work together to present modified copies of the log and map, our protocol even allows a publisher to register more than one certificate for an application by using two disjoint subsets of both CAs and ILSes. So far, we have tolerated this weakness because of expansion considerations – however, it could be avoided by always synchronizing modifications between all existing ILSes without a choice. The system could still be extended by new log servers or authorities, but would require each certificate to be updated. Nevertheless, there is another

reason for our decision to not include all ILSes and CAs by default: Operating an authority or a log is expensive. Since we rely on multiple of them to guarantee security, we want to offer incentives for potential commercial service providers. Because publishers usually have a great interest in assuring the client that the application has not been tampered with, they would be likely to pay if they want to include more than one ILS or `caMin` CAs to achieve a higher level of robustness and security. Note that this also implies that the minimal configuration could still be free of charge and in turn an encouragement for domain owners to register at all.

Gossip Protocols Returning to the split-view attack vulnerability, we pointed out that monitors are able to detect such attacks. However, a client would have to trust them – since they must be highly available and can therefore be easily attacked, this is rather undesirable. As assumed in DTKI, gossip protocols [111] can be integrated in our system to let clients communicate about their view of an ILS. This could be realized without any additional infrastructure or requests [112]: By piggybacking the `MapRoot` on top of the requests and responses in our protocol, misbehavior can be detected with high probability if a quorum of both clients and CAs or ILSes gossip. Although we did not implement a gossip mechanism as part of this work, it could be a trade-off between security and profitability in a live system.

Testing A well-known testing technique for web applications are A/B tests [113]: by presenting a modified version of the application with the feature to be tested to a factor of all users, while the rest of them receive the actual current version, one can evaluate the acceptance and impact of the feature with regard to the metrics used. Obviously, this might be critical in terms of security, especially since it allows a publisher to almost drown clients in an arbitrary number of versions that have not been reviewed. For this reason, we decided to allow only one certificate per application domain – while we generally consider such techniques inadequate for security-critical systems, the publisher can still use them by redirecting on different (sub-)domains.

Mobile Clients A fundamental difference to ARPKI is the use of a log-backed map and the associated proofs of log inclusion and consistency. It might be that CAs and monitors would be able to provide enough storage space and computational power to maintain their own Merkle Trees, which in turn would allow them to verify actions of an ILS without further proofs. However, clients must be able to verify inclusion as well, and ideally even consistency if enough resources are available. Assuming the case that it is a mobile client, we can benefit from common browser technology that shares the basis with desktop systems, but have to cope with limited resources and computing power. Hence, the log-backed map with additional proofs, but of fixed size, is much more suitable for these devices because we exchange a whole tree to be maintained for some hash calculations – however, remember that monitoring (which proves consistency and operation correctness) remains optional to avoid unnecessarily burdening the client's connection.

4 Conclusion

This work contributes to protecting the privacy and confidentiality of user data by making a novel proposal for protection against resource alteration attacks that are targeted at one or a group of clients. With these, an adversary is able to make changes to the source code of a single page application, possibly even with the help of its publisher, in order to gain access to the user's sensitive data. As a result, traditional client-side web attacks are no longer necessary, and the approaches proposed so far do not provide sufficient protection against them. In contrast, our system is capable of effectively protecting against such attacks, even if the adversary has compromised a certain but scalable number of components within our infrastructure. To provide more security to casual users of an application, we additionally enable security experts to pass on their analysis results to the clients.

To conclude this thesis, we will give a short summary of how we designed this system and subsequently discuss possible improvements left for future work.

4.1 Summary

The shift from static websites to JavaScript-based single page applications has enabled a client-side attack called tampering, which was previously more ascribed to the server-side. By manipulating resources, security mechanisms against classic client-side attacks can be bypassed. In this work, we assume a powerful adversary who, in addition to content distribution networks, controls the publisher of a website as well as other members of the network that may be relevant for security systems. We call their attacks Resource Alteration Attacks because they can legitimately make changes to the application. Further, we consider a scenario where they attack a specific or a small group of users of a single page application and have control over an arbitrary number of network members – but never over all, otherwise we wouldn't have a chance. Based on this, none of the existing proposals against tampering can detect the attack, and certainly not protect sensitive user data.

We propose a Public Key Infrastructure (PKI) that provides effective protection in this scenario. Certificate Authorities (CA) issue certificates containing collision-resistant hash values of application resources, which are stored by a publicly viewable append-only log server in a verifiable manner. The CAs supervise this process as well as the actions of each other, and approve them through digital signatures – thus, they can be held accountable in case of inconsistencies. The client can verify inclusion of a certificate at the log server as well as equality between the certificate resources and the ones provided by the publisher. Furthermore, we enable security experts called Auditors to deliver analysis results to the client in the same way, so that application vulnerabilities that lead to common client-side attacks can easily be indicated.

To find a suitable design, we first surveyed the PKI landscape, which can be divided into CA-centric, client-centric and domain-centric. Representatives of the former do not protect against compromised CAs, which would be easy for the adversary to control. Client-centric

approaches can only offer limited security, since even an honest publisher is never involved. We therefore took a closer look at domain-centric approaches and opted for infrastructures based on log servers, as consensus building in blockchains allows the attacker to take control of the majority. We then compared them in terms of various security features and the statements they can prove – most of these proposals are based on the authenticated data structure Merkle Tree, which can generate proofs about its construction and content.

This binary hash tree, in which each node contains the concatenation of the hash values of its children and each leaf contains the actual data, generally allows the construction of proofs of existence for a data value. These consist of the sibling nodes of all ancestors of the leaf, and therefore have a space requirement that is logarithmic in the number of data present, just as the time the verification algorithm takes. If additionally, the tree is append-only (i.e. that no elements can be deleted) one can construct a proof in a similar way that a Merkle Tree represents an extension of another. In contrast, a Sparse Merkle Tree (SMT) cannot prove consistency, but allows the implementation of a key-value store, and thus adding and removing values, by introducing an empty data element. The binary key is converted into a path to the leaf with the data element – this can be used to prove whether a particular key-value pair is included or not. To maintain a history of the changes, an SMT is combined with an append-only Merkle Tree in a log-backed map, where the latter contains the modifications of the SMT.

The Integrity Log Servers (ILS) of our public key infrastructure use such log-backed maps to store the certificates in a verifiable manner. Since the requirements for our approach include not only security but also quality attributes such as efficiency, we chose this solution instead of simple append-only Merkle Trees, as it allows to quickly prove presence as well as absence of certificates and still consistency between log versions. Besides certificates for resources and those for analyses, an ILS also holds dedicated certificates for the publisher or auditor, so that their authenticity does not have to be checked manually by the CA with every new version of the resource or analysis certificate – instead, this can be done automatically by means of digital signatures.

In order to register, modify and query a certificate, we modified and extended an existing protocol. It now consists of the following steps, which are almost completely generic for the three certificate types: To generate a registrable certificate, the publisher of a security-critical application first lists the CAs to be contacted, which verify the content and sign the certificate. Then, in step two, it is transferred to one of the listed CAs, which monitors the subsequent process. This then forwards the certificate to an ILS also specified by the publisher, which synchronizes it with other ILSes if required and sends proof of correct execution of the operation to the next CA in the list. This CA also checks the content of the certificate and verifies the proofs, which it signs as approval if successful. It also stores the root of the log-backed map, which serves as a fingerprint for all data contained. As the signed proofs are forwarded to the other CAs and finally to the initial one in a ring structure, the certificate authorities can monitor the process and each other through their signatures. This quite extensive modification procedure allows a high degree of security to be very efficiently guaranteed in the third step, the client's request for a certificate: As in the last step, the response of the ILS is passed through all CAs – however, they do not have to verify the certificate proof, but only to compare the roots and, if

they match, can directly sign and forward it. The client can then verify that everyone accepts the root, that the received certificate is really in the log-backed map and that the application resources received from the publisher match those in the certificate.

Furthermore, we have developed a procedure to verify global consistency and correctness of all log-backed maps. Since the verifying entity does not have to create and maintain Merkle Trees, this can be done by dedicated Monitors as well as by the client itself.

A security analysis of the public key infrastructure, which considers external, internal attacks and combinations of these, revealed a scalable security level for the presentation of a bogus certificate. While purely external attacks, such as Man-in-the-Middle (MitM) or Denial-of-Service (DoS), have no chance of success, even internal attacks in which one or more entities are compromised can only remain undetected if all participating CAs and ILSes, as well as the publisher itself, are malicious. If the adversary attacks with a combination of MitM, DoS and compromise, they must still have control over the CAs and at least one ILS to succeed.

After presenting basic information of our prototype implementation of all participants, we discussed aspects that are relevant for the productive use of the system. In addition, we pointed out economic incentives that, in addition to simple deployment, contribute to a rapid and large-scale commissioning.

4.2 Future Work

This work lays the foundation to increase security for the user of a single page application, as well as their confidence. Nevertheless, besides code optimization and porting, there are further technical aspects as well as theoretical and practical questions that should be addressed in the future.

Audit Properties From a technical point of view, we have not specified how more complex assertions resulting from analyses, which are delivered to the client in audition certificates, should be specified and verified. The static verification in our prototype implementation that `eval` is not used can be done relatively easily – complex assertions that rely on network monitoring or dynamic analysis, for example, require additional conception and programming effort for our browser extension. A fundamental question here is whether all audit properties should be verifiable by assertions and whether they can be at all.

Formal proof From a theoretical point of view, although it's not very common in the area of public key infrastructures, we believe that security protocols should always be formally verified – since Lowe's attack on the Needham-Schroeder protocol [114], it's obvious that informal analyses cannot suffice for more complex security protocols. Although we strongly rely on two verified infrastructures for our protocol, there should be formal proof before using the system. One approach could be not to rely on the Tamarin prover and rewriting like ARPKI and DTKI, but to model the individual participants using structured

algebraic specifications. This would make it possible to prove statements for any number of participants instead of a few concrete instances using expressive logics.

Multi Page Applications From a practical point of view, the restriction to single page applications that do not change URL paths excludes websites that use multiple URLs, especially multi page applications. However, since hybrid and some single page applications also work with different paths, it would be desirable for our system to be able to handle them. For example, one could use regular expressions to describe the corresponding URLs – but for validation you would have to prove that the intersection of this language with all other registered ones is empty. For example, the complement of the union of the complements of the corresponding finite machines could be determined for two languages each, but this might result in considerable computational effort for many registered certificates.

Subresource Integrity Last but not least, we would like to discuss Subresource Integrity, which we already presented in section 1.2 as a technique against tampering. As soon as its status changes from Recommendation to Living Standard and all browsers support it, we can benefit by not having to list all resource hashes anymore, but only one of the index HTML page in the certificate. All other resources together with their digests are stored in this page and protected by Subresource Integrity.

List of Figures

List of Tables

Bibliography

[1] B. Stock, M. Johns, M. Steffens, and M. Backes, "How the Web Tangled Itself: Uncovering the History of Client-Side Web (In)Security", in *26th USENIX Security Symposium (USENIX Security 17)*, USENIX Association, 2017, pp. 971–987.

[2] ThejQuery Foundation, *Jquery*. [Online]. Available: `https://jquery.com/` (visited on 12/30/2018).

[3] BuiltWith Pty Ltd, *Jquery usage statistics*. [Online]. Available: `https://trends.builtwith.com/javascript/jQuery` (visited on 12/28/2018).

[4] L. Gloaguen, *Single-page application, pros and cons | blog*, Nov. 2018. [Online]. Available: `https://www.spiria.com/en/blog/web-applications/single-page-application-pros-and-cons/` (visited on 12/27/2018).

[5] R. Carvalho, *Single page applications: When and why you should use them*, Nov. 2017. [Online]. Available: `https://www.scalablepath.com/blog/single-page-applications/` (visited on 12/28/2018).

[6] M. Jensen, N. Gruschka, and R. Herkenhoener, "A survey of attacks on web services", *Computer Science - Research and Development (CSRD)*, vol. 24, no. 4, pp. 185–197, 2009.

[7] X. Li and Y. Xue, "A Survey on Server-side Approaches to Securing Web Applications", *ACM Comput. Surv.*, vol. 46, no. 4, 54:1–54:29, Mar. 2014.

[8] S. Fogie, J. Grossman, R. Hansen, A. Rager, and P. D. Petkov, *XSS Attacks: Cross Site Scripting Exploits and Defense*. Syngress Publishing, 2007.

[9] S. Gupta and B. B. Gupta, "Cross-site scripting (xss) attacks and defense mechanisms: Classification and state-of-the-art", *International Journal of System Assurance Engineering and Management*, vol. 8, no. 1, pp. 512–530, 2017.

[10] I. Hydara, A. B. M. Sultan, H. Zulzalil, and N. Admodisastro, "Current state of research on cross-site scripting (XSS) – A systematic literature review", *Information and Software Technology*, vol. 58, pp. 170–186, 2015.

[11] B. Stock, S. Pfistner, B. Kaiser, S. Lekies, and M. Johns, "From Facepalm to Brain Bender: Exploring Client-Side Cross-Site Scripting", in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15, ACM, 2015, pp. 1419–1430.

[12] N. Jovanovic, E. Kirda, and C. Kruegel, "Preventing cross site request forgery attacks", in *2006 Securecomm and Workshops*, Aug. 2006, pp. 1–10.

[13] W. Zeller and E. W. Felten, "Cross-site request forgeries: Exploitation and prevention", *The New York Times*, pp. 1–13, 2008.

[14] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart, "Http authentication: Basic and digest access authentication", RFC 2617, 1999.

[15] M. Johns and J. Winter, "Requestrodeo: Client side protection against session riding", in *Proceedings of the OWASP Europe 2006 Conference, refereed papers track, Report CW448*, 2006, pp. 5–17.

[16] Guide to Cryptography - OWASP, *Cross Frame Scripting*, Jun. 2016. [Online]. Available: `https://www.owasp.org/index.php/Cross_Frame_Scripting` (visited on 12/29/2018).

[17] S. White, *XFS 101: Cross-Frame Scripting Explained*, Aug. 2010. [Online]. Available: `http://securestate.blogspot.com/2010/08/xfs-101-cross-frame-scripting-explained.html` (visited on 12/29/2018).

[18] D. Prabhakara, "Web Applications Security - A security model for client-side web applications", Master's thesis, Norges teknisk-naturvitenskapelige universitet, Trondheim, Norway, 2009.

[19] T. Kanti, V. Richariya, and V. Richariya, "Implementing a web browser with web defacement detection techniques", *World of Computer Science and Information Technology Journal (WCSIT)*, 7th ser., vol. 1, pp. 307–310, 2011.

[20] X. Long, H. Peng, C. Zhang, Z. Pan, and Y. Wu, "A Fragile Watermarking Scheme for Tamper-Proof of Web Pages", in *2009 WASE International Conference on Information Engineering*, vol. 2, 2009, pp. 155–158.

[21] X. Liu and H. Lu, "Fragile Watermarking Schemes for Tamperproof Web Pages", in *Advances in Neural Networks - ISNN 2008*, Springer Berlin Heidelberg, 2008, pp. 552–559.

[22] M. Masango, F. Mouton, P. Antony, and B. Mangoale, "Web Defacement and Intrusion Monitoring Tool: WDIMT", in *2017 International Conference on Cyberworlds (CW)*, IEEE, 2017, pp. 72–79.

[23] R. Kachhawa, N. K. Singh, and D. S. Tomar, "A Novel Approach to Detect Web Page Tampering", *(IJCSIT) International Journal of Computer Science and Information Technologies*, 3rd ser., vol. 5, pp. 4604–4607, 2014.

[24] E. Medvet, C. Fillon, and A. Bartoli, "Detection of Web Defacements by means of Genetic Programming", in *Third International Symposium on Information Assurance and Security*, Aug. 2007, pp. 227–234.

[25] Ú. Erlingsson, B. Livshits, and Y. Xie, "End-to-end Web Application Security", in *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*, ser. HOTOS'07, USENIX Association, 2007, 18:1–18:6.

[26] J. Terrace, H. Laidlaw, H. E. Liu, S. Stern, and M. J. Freedman, "Bringing P2P to the Web: Security and Privacy in the Firecoral Network", in *Proceedings of the 8th International Conference on Peer-to-peer Systems*, ser. IPTPS'09, USENIX Association, 2009, pp. 7–7.

[27] A. Levy, H. Corrigan-Gibbs, and D. Boneh, "Stickler: Defending against Malicious Content Distribution Networks in an Unmodified Browser", *IEEE Security & Privacy*, vol. 14, no. 2, pp. 22–28, 2016.

[28] R. A. Popa, E. Stark, J. Helfer, S. Valdez, N. Zeldovich, M. F. Kaashoek, and H. Balakrishnan, "Building Web Applications on Top of Encrypted Data Using Mylar", in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, USENIX Association, 2014, pp. 157–172.

[29] D. Akhawe, F. Marier, F. Braun, and J. Weinberger, "Subresource integrity", *W3C working draft, W3C, July*, 2015.

[30] MDN Web Docs, *Subresource Integrity*, Nov. 2018. [Online]. Available: `https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity` (visited on 12/30/2018).

[31] N. Bielova, "Survey on JavaScript security policies and their enforcement mechanisms in a web browser", *The Journal of Logic and Algebraic Programming*, vol. 82, no. 8, pp. 243–262, 2013.

[32] Guide to Cryptography - OWASP, *Source Code Analysis Tools*, Dec. 2018. [Online]. Available: `https://www.owasp.org/index.php/Source_Code_Analysis_Tools` (visited on 12/30/2018).

[33] R. C. Merkle, "A Digital Signature Based on a Conventional Encryption Function", in *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, ser. CRYPTO '87, Springer-Verlag, 1988, pp. 369–378.

[34] D. Basin, C. Cremers, T. H.-J. Kim, A. Perrig, R. Sasse, and P. Szalachowski, "ARPKI: Attack Resilient Public-Key Infrastructure", in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14, ACM, 2014, pp. 382–393.

[35] J. R. Vacca, *Public Key Infrastructure: Building Trusted Applications and Web Services*, 1st. Auerbach Publications, 2004.

[36] E. Rescorla, "Http over tls", RFC 2818, 2000.

[37] S. Santesson, T. Polk, R. Housley, S. Farrell, and D. Cooper, *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*, RFC 5280, May 2008.

[38] S. S. Xenitellis, "The Open–source PKI Book", 2000.

[39] J. Yu, V. Cheval, and M. Ryan, "DTKI: A new formalized PKI with verifiable trusted parties", *The Computer Journal*, vol. 59, no. 11, pp. 1695–1713, 2016.

[40] T. Sterling, *Second firm warns of concern after Dutch hack*, Sep. 2011. [Online]. Available: `https://phys.org/news/2011-09-firm-dutch-hack.html` (visited on 12/28/2018).

[41] P. Roberts, *Phony SSL Certificates issued for Google, Yahoo, Skype, Others*, Mar. 2011. [Online]. Available: `https://threatpost.com/phony-ssl-certificates-issued-google-yahoo-skype-others-032311/75061/` (visited on 12/28/2018).

[42] P. Ducklin, *The TURKTRUST SSL certificate fiasco – what really happened, and what happens next?*, Feb. 2013. [Online]. Available: https://nakedsecurity.sophos.com/2013/01/08/the-turktrust-ssl-certificate-fiasco-what-happened-and-what-happens-next/ (visited on 12/27/2018).

[43] J. Appelbaum, *Detecting Certificate Authority compromises and web browser collusion*, Mar. 2011. [Online]. Available: https://blog.torproject.org/detecting-certificate-authority-compromises-and-web-browser-collusion (visited on 12/28/2018).

[44] J. Menn, *Key Internet operator VeriSign hit by hackers*, Feb. 2012. [Online]. Available: https://www.reuters.com/article/us-hacking-verisign/key-internet-operator-verisign-hit-by-hackers-idUSTRE8110Z820120202 (visited on 12/28/2018).

[45] C. Arthur, *Rogue web certificate could have been used to attack Iran dissidents*, Aug. 2011. [Online]. Available: https://www.theguardian.com/technology/2011/aug/30/faked-web-certificate-iran-dissidents (visited on 12/28/2018).

[46] J. Katz and Y. Lindell, *Introduction to Modern Cryptography*, 2nd. Chapman & Hall/CRC, 2014.

[47] E. Andreeva, B. Mennink, and B. Preneel, "Open Problems in Hash Function Security", *Designs, Codes and Cryptography*, vol. 77, no. 2-3, pp. 611–631, Dec. 2015.

[48] S. Indesteege, "Analysis and design of cryptographic hash functions", PhD thesis, Katholieke Universiteit Leuven, Leuven, Belgium, 2010.

[49] T. Duong and J. Rizzo, *Flickr's API Signature Forgery Vulnerability*, Sep. 2009.

[50] W. Diffie and M. Hellman, "New Directions in Cryptography", *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, Sep. 1976, ISSN: 0018-9448.

[51] R. Östersjö, "Sparse merkle trees: Definitions and space-time trade-offs with applications for balloon", PhD thesis, Karlstad University, Karlstad, Sweden, 2016.

[52] R. L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-key Cryptosystems", *Commun. ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978.

[53] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman, "CONIKS: Bringing Key Transparency to End Users", in *24th USENIX Security Symposium (USENIX Security 15)*, USENIX Association, 2015, pp. 383–398.

[54] S. Fahl, S. Dechand, H. Perl, F. Fischer, J. Smrcek, and M. Smith, "Hey, NSA: Stay away from my market! Future proofing app markets against powerful attackers", in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2014, pp. 1143–1155.

[55] M. Lepinski and S. Kent, "An infrastructure to support secure internet routing", RFC 6480, 2012.

[56] F. Li and P. Xiong, “Practical secure communication for integrating wireless sensor networks into the internet of things”, *IEEE Sensors Journal*, vol. 13, no. 10, pp. 3677–3684, 2013.

[57] K. Nikitin, E. Kokoris-Kogias, P. Jovanovic, N. Gailly, L. Gasser, I. Khoffi, J. Cappos, and B. Ford, “CHAINIAC: Proactive Software-Update Transparency via Collectively Signed Skipchains and Verified Builds”, in *26th USENIX Security Symposium (USENIX Security 17)*, USENIX Association, 2017, pp. 1271–1287.

[58] S. Galperin, S. Santesson, M. Myers, A. Malpani, and C. Adams, “X.509 Internet public key infrastructure online certificate status protocol – OCSP”, RFC 2560, 2013.

[59] D. Eastlake 3rd, “Transport layer security (tls) extensions: Extension definitions”, RFC 6066, 2011.

[60] P. Hallam-Baker, “X.509v3 transport layer security (tls) feature extension”, RFC 7633, 2015.

[61] H. Bock, *The Problem with OCSP Stapling and Must Staple and why Certificate Revocation is still broken*, May 2017. [Online]. Available: `https://blog.hboeck.de/archives/886-The-Problem-with-OCSP-Stapling-and-Must-Staple-and-why-Certificate-Revocation-is-still-broken.html` (visited on 12/27/2018).

[62] E. Topalovic, B. Saeta, L.-S. Huang, C. Jackson, and D. Boneh, “Towards short-lived certificates”, *Proceedings of IEEE Oakland Web 2.0 Security and Privacy*, 2012.

[63] M. Abadi, A. Birrell, I. Mironov, T. Wobber, and Y. Xie, “Global Authentication in an Untrustworthy World”, in *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, ser. HotOS’13, USENIX Association, 2013, pp. 19–19.

[64] D. Wendlandt, D. G. Andersen, and A. Perrig, “Perspectives: Improving SSH-style Host Authentication with Multi-Path Probing”, in *USENIX Annual Technical Conference*, vol. 8, 2008, pp. 321–334.

[65] Convergence, *Convergence - The Realtime Collaboration App Platform*, 2011. [Online]. Available: `http://convergence.io/` (visited on 12/27/2018).

[66] E. F. Foundation, *The EFF SSL Observatory*, Apr. 2017. [Online]. Available: `https://www.eff.org/de/observatory` (visited on 12/27/2018).

[67] C. Evans, C. Palmer, and R. Sleevi, “Public key pinning extension for http”, RFC 7469, 2015.

[68] M. Marlinspike, *Tack - trust assertions for certificate keys*, T. Perrin, Ed., Jan. 2013. [Online]. Available: `http://tack.io/draft.html` (visited on 12/27/2018).

[69] P. Hallam-Baker and R. Stradling, “DNS certification authority authorization (CAA) resource record”, RFC 6844, 2013.

[70] P. Hoffman and J. Schlyter, “The DNS-based authentication of named entities (DANE) transport layer security (TLS) protocol: TLSA”, RFC 6698, 2012.

[71] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang, "An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends", in *2017 IEEE International Congress on Big Data (BigData Congress)*, Jun. 2017, pp. 557–564.

[72] G. Slepak, *DNSChain*, Apr. 2017. [Online]. Available: `https://github.com/okTurtles/dnschain` (visited on 12/28/2018).

[73] A. Loibl and J. Naab, "Namecoin", *namecoin.info*, 2014.

[74] S. Matsumoto and R. M. Reischuk, "IKP: Turning a PKI around with decentralized automated incentives", in *IEEE Symposium on Security and Privacy (SP)*, IEEE, 2017, pp. 410–426.

[75] K. Lewison and F. Corella, "Backing rich credentials with a blockchain PKI", *Pomcor.com*, 2016.

[76] X. Li, P. Jiang, T. Chen, X. Luo, and Q. Wen, "A Survey on the Security of Blockchain Systems", *Future Generation Computer Systems*, 2017.

[77] D. Kamboj and T. A. Yang, "An Exploratory Analysis of Blockchain: Applications, Security, and Related Issues", *Proceedings of the 16th Int'l Conf on Scientific Computing*, CSC '13, pp. 67–72,

[78] P. Eckersley, *Sovereign Key Cryptography for Internet Domains*, Jun. 2012. [Online]. Available: `https://github.com/EFForg/sovereign-keys/blob/master/sovereign-key-design.txt` (visited on 12/28/2018).

[79] B. Laurie, A. Langley, and E. Kasper, *Certificate Transparency*, RFC 6962, Jun. 2013.

[80] 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", in *Proceedings of the 22Nd International Conference on World Wide Web*, ser. WWW '13, ACM, 2013, pp. 679–690.

[81] S. Meier, B. Schmidt, C. Cremers, and D. Basin, "The TAMARIN prover for the symbolic analysis of security protocols", in *International Conference on Computer Aided Verification*, Springer, 2013, pp. 696–701.

[82] P. Szalachowski, S. Matsumoto, and A. Perrig, "PoliCert: Secure and flexible TLS certificate management", in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2014, pp. 406–417.

[83] M. D. Ryan, "Enhanced certificate transparency and end-to-end encrypted mail", in *Network and Distributed System Security Symposium (NDSS)*, Internet Society, 2014.

[84] S. Khan, Z. Zhang, L. Zhu, M. Li, K. Safi, Q. Gul, and X. Chen, "Accountable and Transparent TLS Certificate Management: An Alternate Public-Key Infrastructure with Verifiable Trusted Parties", *Security and Communication Networks*, vol. 2018, 2018.

[85] J. Yu and M. Ryan, "Evaluating Web PKIs", in *Software Architecture for Big Data and the Cloud*, Elsevier, 2017, pp. 105–126.

[86] T. Hagerup, “Informatik III”, Lecture Notes, Institut für Informatik, Universität Augsburg, Oct. 2013.

[87] S. A. Crosby and D. S. Wallach, “Authenticated Dictionaries: Real-World Costs and Trade-Offs”, *ACM Transactions on Information and System Security (TISSEC)*, vol. 14, no. 2, 17:1–17:30, Sep. 2011.

[88] B. Laurie and E. Kasper, “Revocation transparency”, *Google Research*, 2012.

[89] R. Dahlberg, T. Pulls, and R. Peeters, “Efficient sparse merkle trees”, in *Nordic Conference on Secure IT Systems*, Springer, 2016, pp. 199–215.

[90] A. Eijdenberg, B. Laurie, and A. Cutter, *Verifiable data structures*, 2015.

[91] T. Pulls and R. Peeters, “Balloon: A forward-secure append-only persistent authenticated data structure”, in *European Symposium on Research in Computer Security*, Springer, 2015, pp. 622–641.

[92] B. Kulynych, “Claimchain: Decentralized public key infrastructure”, PhD thesis, ETSI Informatica, Málaga, Spain, 2017.

[93] M. Bellare, O. Goldreich, and S. Goldwasser, “Incremental cryptography: The case of hashing and signing”, in *Annual International Cryptology Conference*, Springer, 1994, pp. 216–233.

[94] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin, “Dynamic Authenticated Index Structures for Outsourced Databases”, in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’06, ACM, 2006, pp. 121–132.

[95] R. Bayer and E. McCreight, “Organization and Maintenance of Large Ordered Indices”, in *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, ser. SIGFIDET ’70, ACM, 1970, pp. 107–141.

[96] M. Ogawa, E. Horita, and S. Ono, “Proving Properties of Incremental Merkle Trees”, in *Proceedings of the 20th International Conference on Automated Deduction*, ser. CADE’ 20, Springer-Verlag, 2005, pp. 424–440.

[97] V. Buterin, *Ethereum: A next-generation smart contract and decentralized application platform*, 2014. [Online]. Available: `https://github.com/ethereum/wiki/wiki/White-Paper` (visited on 12/29/2018).

[98] S. Nakamoto, *Bitcoin: A peer-to-peer electronic cash system*, 2009. [Online]. Available: `http://www.bitcoin.org/bitcoin.pdf` (visited on 12/29/2018).

[99] V. B. Ethereum Foundation, *Merkling in Ethereum*, Jan. 2015. [Online]. Available: `https://blog.ethereum.org/2015/11/15/merkling-in-ethereum/` (visited on 12/30/2018).

[100] A. Lakshman and P. Malik, “Cassandra: A Decentralized Structured Storage System”, *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, Apr. 2010.

[101] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store", in *ACM SIGOPS operating systems review*, vol. 41, ACM, 2007, pp. 205–220.

[102] J. Benet, "IPFS - Content Addressed, Versioned, P2P File System", *arXiv preprint arXiv:1407.3561*, 2014.

[103] J. Bonwick, M. Ahrens, V. Henson, M. Maybee, and M. Shellenbaum, "The Zettabyte File System", *Proc. of the 2nd Usenix Conference on File and Storage Technologies*, vol. 215, 2003.

[104] Git Project, *Git*. [Online]. Available: `http://git-scm.com` (visited on 12/30/2018).

[105] Mercurial Project, *Mercurial*. [Online]. Available: `https://www.mercurial-scm.org/` (visited on 12/30/2018).

[106] M. Blum, "Program Result Checking: A New Approach to Making Programs More Reliable", in *Proceedings of the 20th International Colloquium on Automata, Languages and Programming*, ser. ICALP '93, Springer-Verlag, 1993, pp. 1–14.

[107] H. J. Hof, "User-Centric IT Security - How to Design Usable Security Mechanisms", *arXiv preprint arXiv:1506.07167*, 2015.

[108] A. Barth and C. Reis, "The Security Architecture of the Chromium Browser", Tech. Rep., 2008.

[109] MozillaWiki, *Security/sandbox*, Nov. 2018. [Online]. Available: `https://wiki.mozilla.org/Security/Sandbox` (visited on 12/28/2018).

[110] Google, *Trillian: General transparency*, Jan. 2019. [Online]. Available: `https://github.com/google/trillian` (visited on 12/30/2019).

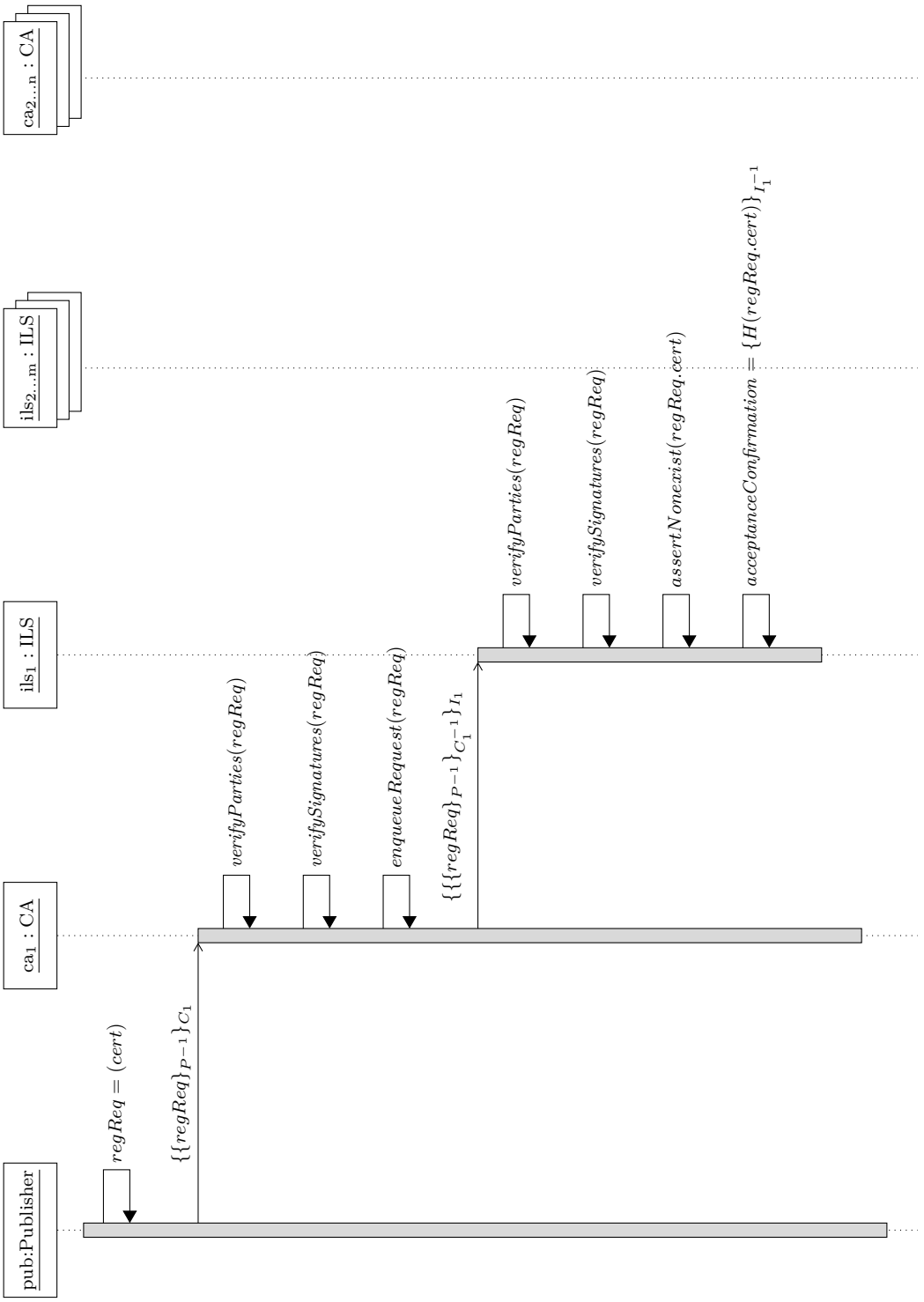
[111] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. Van Steen, "Gossip-based peer sampling", *ACM Transactions on Computer Systems (TOCS)*, vol. 25, no. 3, p. 8, 2007.

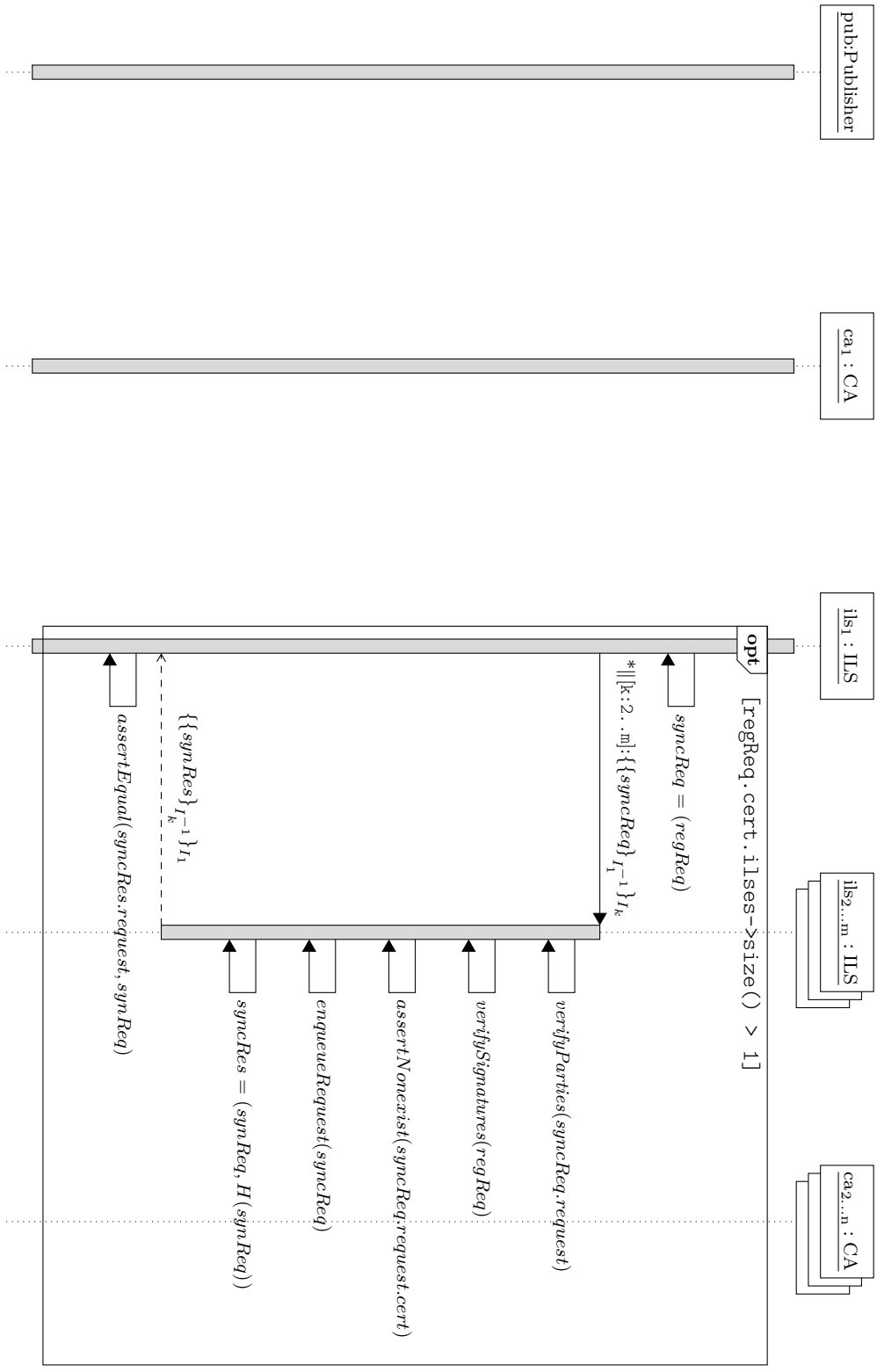
[112] L. Chuat, P. Szalachowski, A. Perrig, B. Laurie, and E. Messeri, "Efficient gossip protocols for verifying the consistency of certificate logs", *arXiv preprint arXiv:1511.01514*, 2015.

[113] R. Kohavi and R. Longbotham, "Online Controlled Experiments and A/B Testing", in *Encyclopedia of Machine Learning and Data Mining*. Springer US, 2017, pp. 922–929.

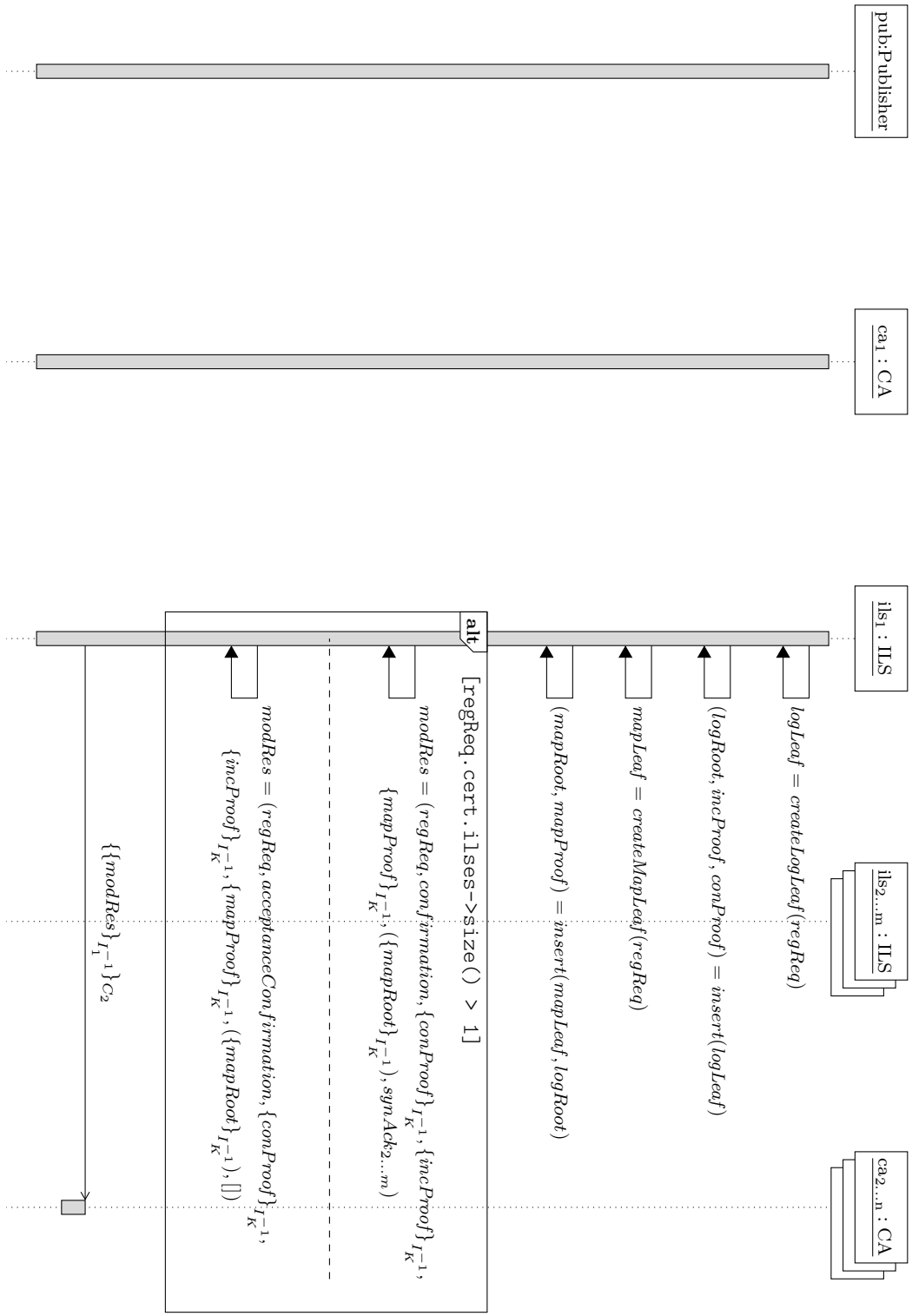
[114] G. Lowe, "An attack on the Needham-Schroeder public-key authentication protocol", *Information Processing Letters*, vol. 56, pp. 131–133, 1995.

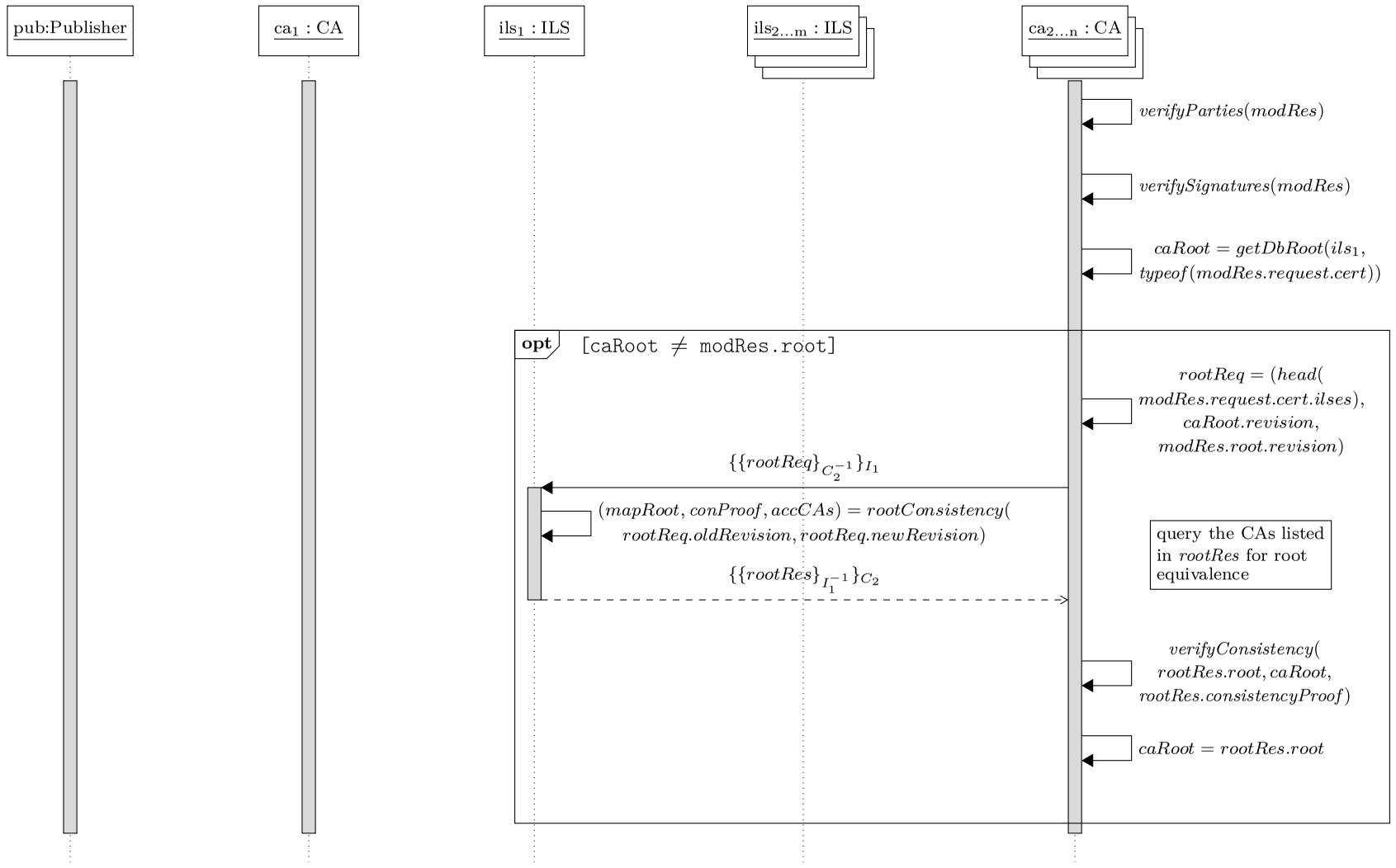
A Log Modification Diagram

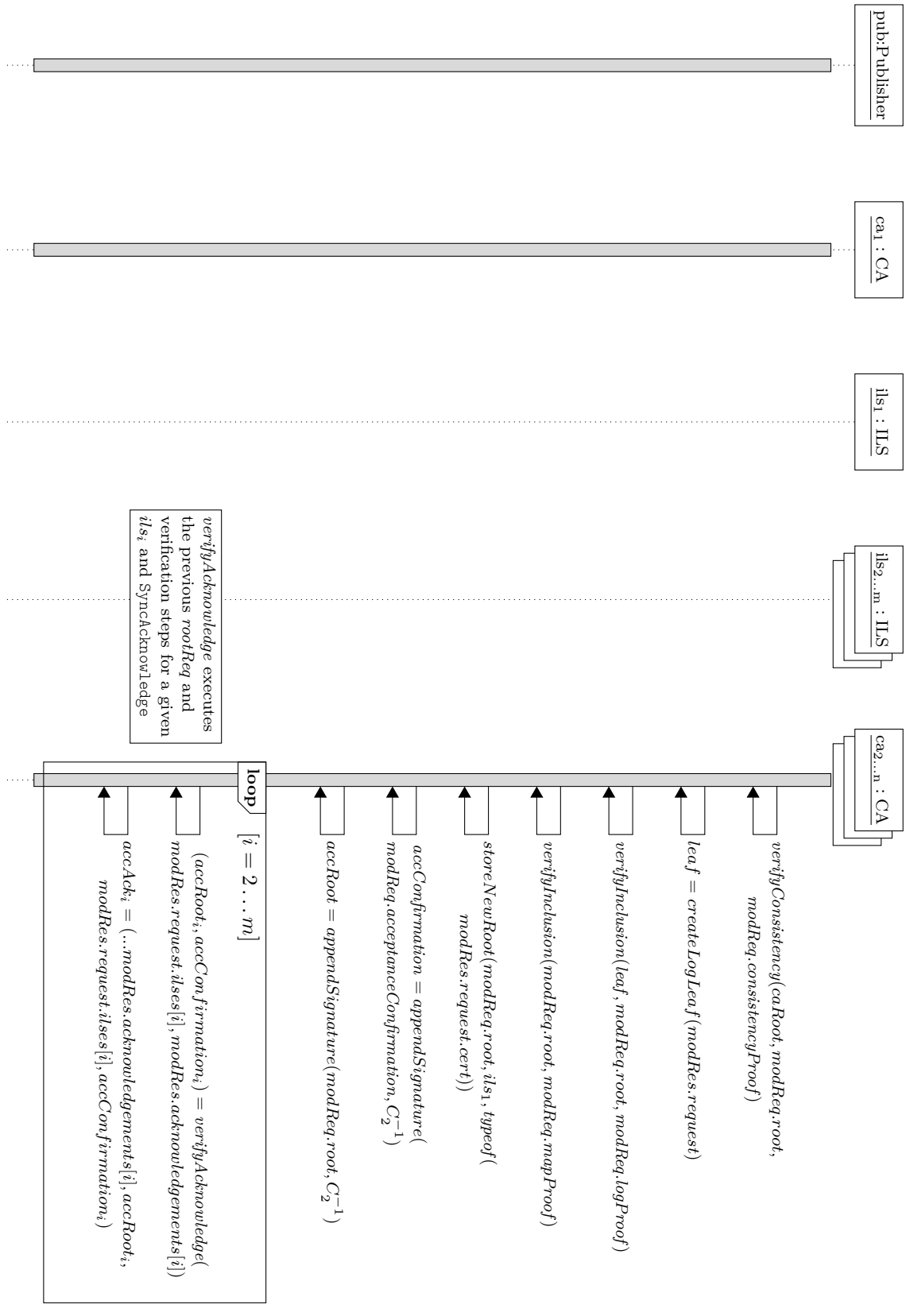


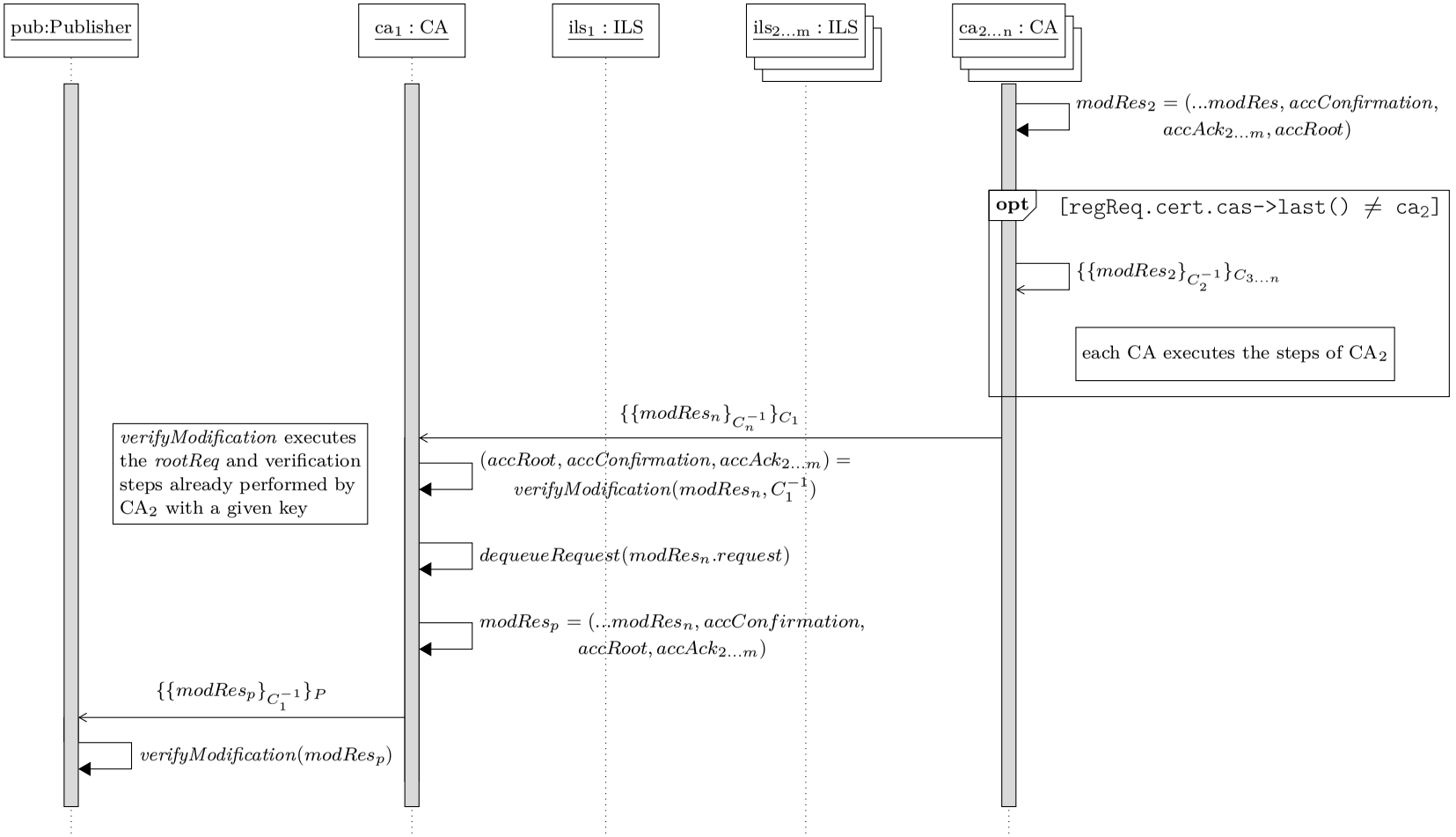

pub:Publisher
ca1 : CA
ils1 : ILS
ils2...m : ILS
ca2...n : CA
opt
[regReq.cert.ilses->size() > 1]
syncReq = (regReq)
*||[k:2..m]:{{syncReq}I1-1}Ik
verifyParties(syncReq.request)
verifySignatures(regReq)
assertNonexist(syncReq:request.cert)
enqueueRequest(syncReq)
syncRes = (synReq, H(synReq))
{{synRes}Ik-1}I1
assertEqual(syncRes.request, synReq)

pub:Publisher
ca1 : CA
ils1 : ILS
ils2...m : ILS
ca2...n : CA
opt
[regReq.cert.ilses->size() > 1]
syncCmt = (H(synReq), acceptanceConfirmation)
*||[k:2..m]:{{syncCmt}}
req = dequeueRequest(syncCmt.hash)
logLeaf = createLogLeaf(req)
(logRoot, incProof, conProof) = insert(logLeaf)
mapLeaf = createMapLeaf(req)
(mapRoot, mapProof) = insert(mapLeaf, logRoot)
confirmation = appendSignature(
synCmt.acceptanceConfirmation,
synAck
{incProof}
{{syncAck

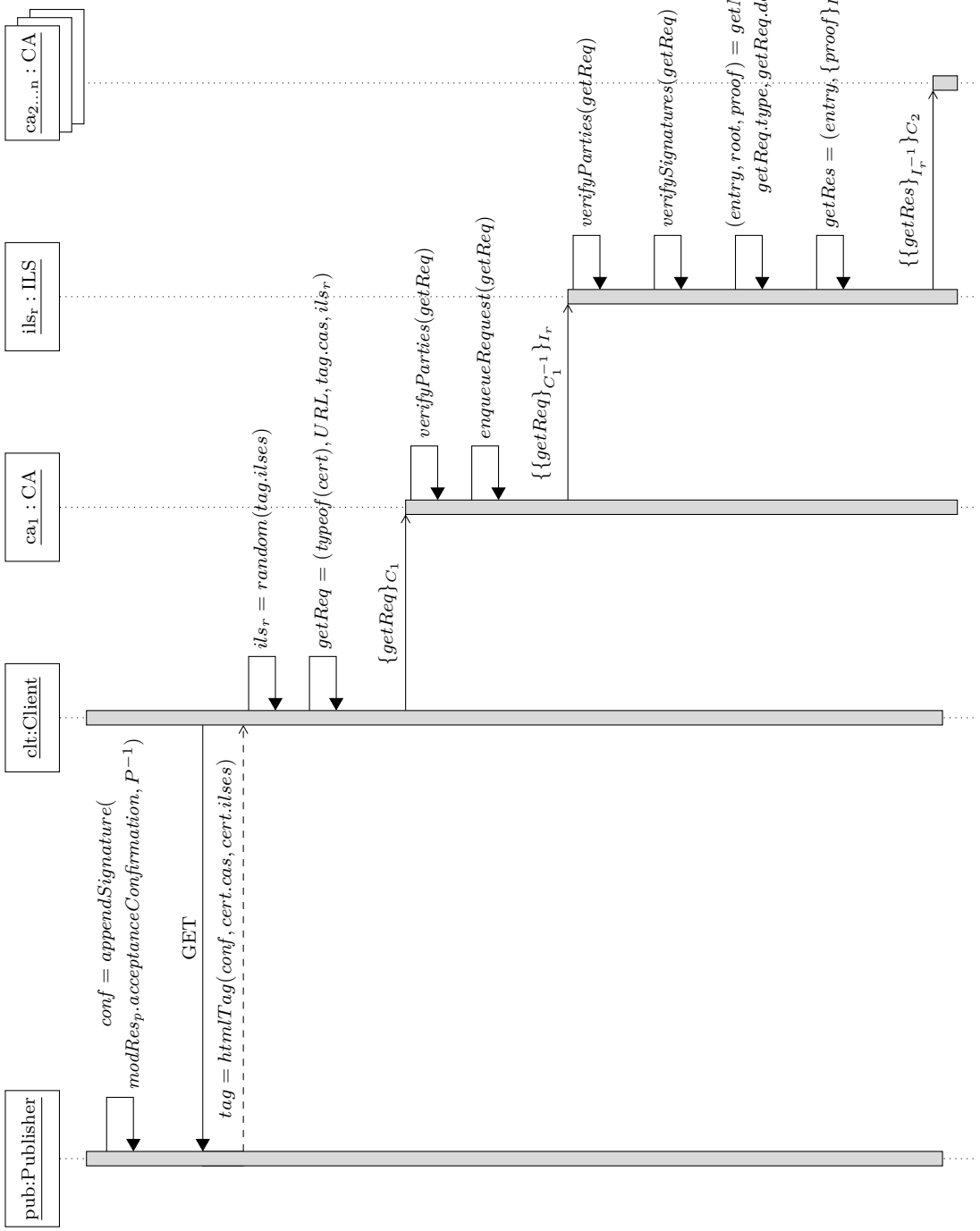

pub:Publisher
ca1 : CA
ils1 : ILS
ils2...m : ILS
ca2...n : CA
logLeaf = createLogLeaf(regReq)
(logRoot, incProof, conProof) = insert(logLeaf)
mapLeaf = createMapLeaf(regReq)
(mapRoot, mapProof) = insert(mapLeaf, logRoot)
alt
[regReq.cert.ilses->size() > 1]
modRes = (regReq, confirmation, {conProof}I_K^-1, {incProof}I_K^-1, {mapProof}I_K^-1, ({mapRoot}I_K^-1), synAck2...m)
modRes = (regReq, acceptanceConfirmation, {conProof}I_K^-1, {incProof}I_K^-1, {mapProof}I_K^-1, ({mapRoot}I_K^-1), [])
{{modRes}I_1^-1}C2

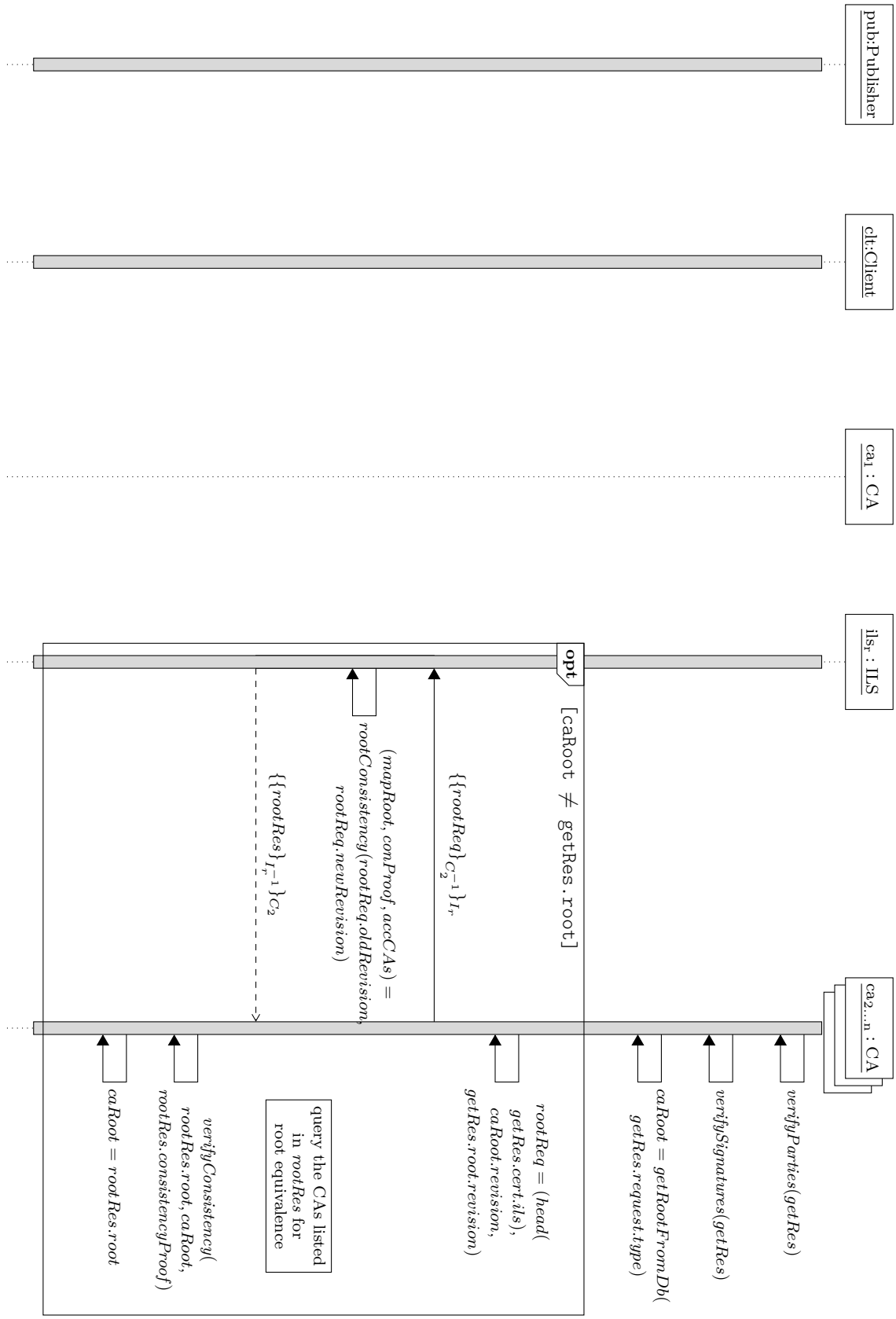

pub:Publisher
ca$_1$: CA
ils$_1$: ILS
ils$_{2...m}$: ILS
ca$_{2...n}$: CA
verifyParties(modRes)
verifySignatures(modRes)
caRoot = getDbRoot(ils$_1$, typeof(modRes.request.cert))
opt
[caRoot ≠ modRes.root]
rootReq = (head(modRes.request.cert.ilses), caRoot.revision, modRes.root.revision)
{{rootReq}$_{C_2^{-1}}$}$_{I_1}$
(mapRoot, conProof, accCAs) = rootConsistency(rootReq.oldRevision, rootReq.newRevision)
query the CAs listed in rootRes for root equivalence
{{rootRes}$_{I_1^{-1}}$}$_{C_2}$
verifyConsistency(rootRes.root, caRoot, rootRes.consistencyProof)
caRoot = rootRes.root

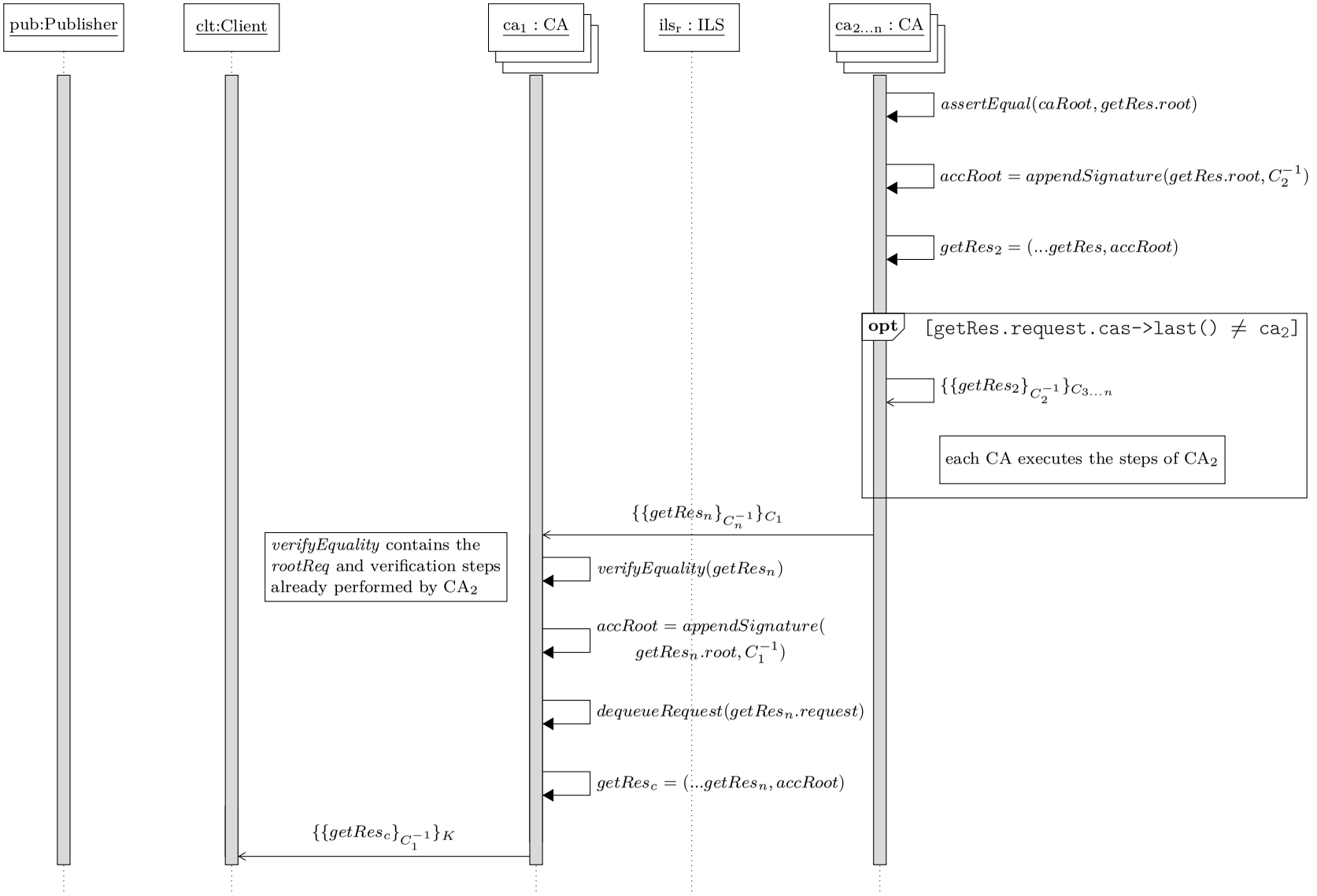

pub:Publisher
ca1 : CA
ils1 : ILS
ils2...m : ILS
ca2...n : CA
verifyConsistency(caRoot, modReq.root, modReq.consistencyProof)
leaf = createLogLeaf(modRes.request)
verifyInclusion(leaf, modReq.root, modReq.logProof)
verifyInclusion(modReq.root, modReq.mapProof)
storeNewRoot(modReq.root, ils1, typeof(modRes.request.cert))
accConfirmation = appendSignature(modReq.acceptanceConfirmation, C2−1)
accRoot = appendSignature(modReq.root, C2−1)
loop
[i = 2 . . . m]
(accRoot_i, accConfirmation_i) = verifyAcknowledge(modRes.request.ilses[i], modRes.acknowledgements[i])
accAck_i = (...modRes.acknowledgements[i], accRoot_i, modRes.request.ilses[i], accConfirmation_i)
verifyAcknowledge executes the previous rootReq and verification steps for a given ils_i and SyncAcknowledge

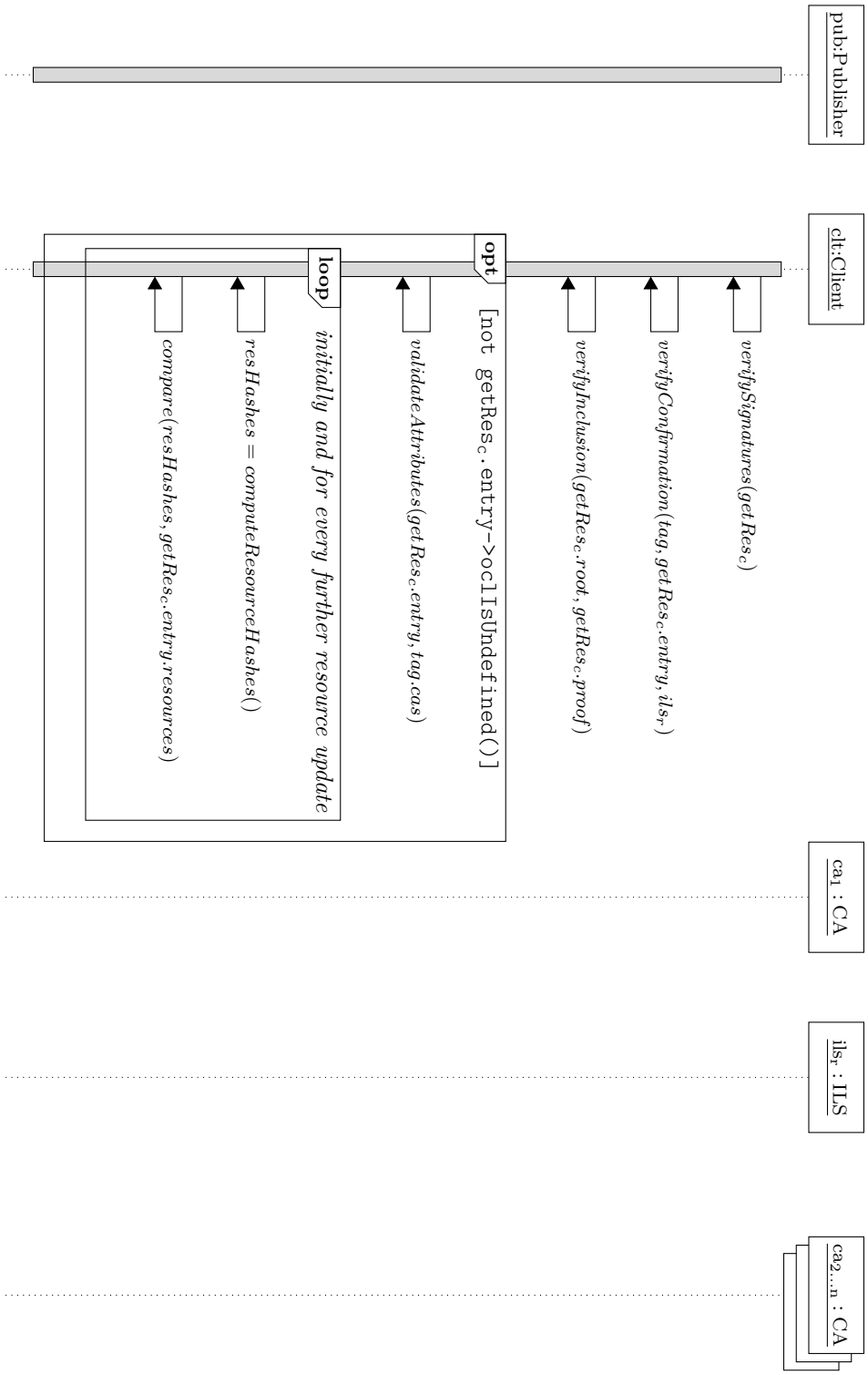

pub:Publisher
ca$_1$: CA
ils$_1$: ILS
ils$_{2...m}$: ILS
ca$_{2...n}$: CA
$modRes_2 = (...modRes, accConfirmation, accAck_{2...m}, accRoot)$
opt
[regReq.cert.cas->last() ≠ ca$_2$]
$\{\{modRes_2\}_{C_2^{-1}}\}_{C_{3...n}}$
each CA executes the steps of CA$_2$
$\{\{modRes_n\}_{C_n^{-1}}\}_{C_1}$
verifyModification executes the *rootReq* and verification steps already performed by CA$_2$ with a given key
$(accRoot, accConfirmation, accAck_{2...m}) = verifyModification(modRes_n, C_1^{-1})$
$dequeueRequest(modRes_n.request)$
$modRes_p = (...modRes_n, accConfirmation, accRoot, accAck_{2...m})$
$\{\{modRes_p\}_{C_1^{-1}}\}_P$
$verifyModification(modRes_p)$

B Resource Verification Diagram




pub:Publisher
clt:Client
ca1 : CA
ilsr : ILS
ca2...n : CA
opt
[caRoot ≠ getRes.root]
verifyParties(getRes)
verifySignatures(getRes)
caRoot = getRootFromDb(
getRes.request.type)
rootReq = (head(
getRes.cert.ils),
caRoot.revision,
getRes.root.revision)
{{rootReq}C2-1}Ir
(mapRoot, conProof, accCAs) =
rootConsistency(rootReq.oldRevision,
rootReq.newRevision)
{{rootRes}Ir-1}C2
query the CAs listed
in rootRes for
root equivalence
verifyConsistency(
rootRes.root, caRoot,
rootRes.consistencyProof)
caRoot = rootRes.root


pub:Publisher
clt:Client
ca1 : CA
ilsr : ILS
ca2...n : CA
assertEqual(caRoot, getRes.root)
accRoot = appendSignature(getRes.root, C2^-1)
getRes2 = (...getRes, accRoot)
opt
[getRes.request.cas->last() ≠ ca2]
{{getRes2}C2^-1}C3...n
each CA executes the steps of CA2
{{getResn}Cn^-1}C1
verifyEquality contains the rootReq and verification steps already performed by CA2
verifyEquality(getResn)
accRoot = appendSignature(
getResn.root, C1^-1)
dequeueRequest(getResn.request)
getResc = (...getResn, accRoot)
{{getResc}C1^-1}K


pub:Publisher
clt:Client
ca1 : CA
ilsr : ILS
ca2..n : CA
verifySignatures(getResc)
verifyConfirmation(tag, getResc.entry, ilsr)
verifyInclusion(getResc.root, getResc.proof)
opt
[not getResc.entry->oclIsUndefined()]
validateAttributes(getResc.entry,tag.cas)
loop
initially and for every further resource update
resHashes = computeResourceHashes()
compare(resHashes, getResc.entry.resources)