Mitigating private key compromise

A thesis submitted in fulfillment of the
requirements for the award of the degree

Doctor of Philosophy

from

UNIVERSITY OF BIRMINGHAM

by

Jiangshan Yu

School of Computer Science
University of Birmingham

August 2016

Dedicated to
My Parents & Grandparents

Declaration

This is to certify that the work reported in this thesis was done by the author, unless specified otherwise, and that no part of it has been submitted in a thesis to any other university or similar institution.

Jiangshan Yu
August 12, 2016

Abstract

Cryptosystems rely on the assumption that the computer end-points can securely store and use cryptographic keys. Yet, this assumption is rather hard to justify in practice. New software vulnerabilities are discovered every day, and malware is pervasive on mobile devices and desktop PCs.

This thesis provides research on how to mitigate private key compromise in three different cases. The first case considers compromised signing keys of certificate authorities in public key infrastructure. To address this problem, we analyse and evaluate existing prominent certificate management systems, and propose a new system called Distributed and Transparent Key Infrastructure, which is secure even if all service providers collude together.

The second case considers the key compromise in secure communication. We develop a simple approach that either guarantees the confidentiality of messages sent to a device even if the device was previously compromised, or allows the user to detect that confidentiality failed. We propose a multi-device messaging protocol that exploits our concept to allow users to detect unauthorised usage of their device keys.

The third case considers the key compromise in secret distribution. We develop a self-healing system, which provides a proactive security guarantee: an attacker can learn a secret only if s/he can compromise all servers simultaneously in a short period.

Acknowledgements

I thankfully acknowledge the funding received towards my PhD from The University of Birmingham under EPSRC Project Grant *Analysing Security and Privacy Properties* (EP/H005501/1). I also would like to thank The School of Computer Science, University of Birmingham, for providing additional funding to cover my international student fee.

My experience as a PhD researcher in the University of Birmingham has been wonderful. I am grateful to my supervisor Mark Ryan for this opportunity. He has been an excellent supervisor — he offered me directions yet enough freedom to explore different research areas. His supervision was vital to me in achieving my goals.

I am fortunate to work with Cas Cremers, Liqun Chen, Vincent Cheval, Guilin Wang, and Yi Mu during my PhD. I would like to thank them for their contributions.

I am delighted to be here with a team of brilliant people. I must mention and thank Dan Ghica and Eike Ritter. They are my research monitoring group members and they have contributed comments and suggestions on my research.

I also would like to thank all my friends for their friendship.

Finally, I greatly appreciate my family for their love, support and understanding.

Publications and Drafts

The following papers have been published or submitted, and contain materials based on the content of this thesis.

1. Jiangshan Yu and Mark Ryan. "Device attacker models: fact and fiction", *Security Protocols XXIII*, 2015, Cambridge, UK.

2. Jiangshan Yu, Mark Ryan, and Cas Cremers. "DECIM: Detecting Endpoint Compromise In Messaging', *IACR Cryptology ePrint Archive*, 2015: 486.

3. Jiangshan Yu, Vincent Cheval, and Mark Ryan. "DTKI: a new formalized PKI with verifiable trusted parties ", *The Computer Journal*, 2016. Doi:10.1093 /comjnl/bxw039.

4. Jiangshan Yu and Mark Ryan. "Evaluating web PKIs".

5. Jiangshan Yu, Mark Ryan, and Liqun Chen. "Authenticating compromisable storage systems".

I am thankful to have opportunities to collaborate with others in other areas of computer and communications security. The contributions are listed below and they are beyond the scope of this thesis.

1. Jiangshan Yu, Guilin Wang, Yi Mu, and Wei Gao. "An Efficient and Improved Generic Framework for Three-Factor Authentication with Provably Secure Instantiation", *IEEE Transactions on Information Forensics and Security (TIFS)*, Vol.9, No.12, pp. 2302-2313.

2. Jiangshan Yu, Guilin Wang, and Yi Mu. "Efficient and Provably Secure Single Sign-on Schemes in Distributed Systems and Networks".

Contents

List of Tables

List of Figures

Part I

Introduction and background

CHAPTER 1

INTRODUCTION

Key management is the hardest part of cryptography and often the Achilles heel of an otherwise secure system.

Bruce Schneier [Sch96, Chapter 8].

Alice wants to communicate with Bob securely. Depending on what they want to do, e.g. to process on-line payment or to exchange private messages, they use different cryptographic protocols. Unfortunately, even if the protocol they use is secure, an attacker might still be able to learn the private communication between Alice and Bob by attacking the security assumption about secure key management. In fact, most attacks in practice are aimed at key management of cryptosystems rather than the cryptographic algorithms. So, successful key management is critical to the security of cryptosystems.

Key management mainly deals with the key generation, distribution, and storage. The attacks on key management are mainly trying to (A) compromise the authenticity of public keys; and (B) make unauthorised uses of private keys.

This thesis presents research on the solutions defending against the above attacks, with a focus on the following three cases.

Case 1 In public key cryptography, the authenticity of public keys is mainly assured by *certificate authorities* (CAs). If a CA is compromised, then the authenticity of public keys cannot be guaranteed. The first case considers how to provide authenticity of public keys when CAs are compromised by an attacker.

Case 2 In the presence of software bugs and malware, an attacker might be able to learn a victim's private keys by attacking the victim's devices. If all private keys are obtained by the attacker, then the security of associated systems is broken. The second case considers how to mitigate the damage caused by compromised private keys, with an application to secure messaging.

Case 3 In secret distribution schemes, a secret is distributed to a set of servers in the way that reconstructing the secret requires a sufficient number of servers to work together. If an attacker is able to gradually compromise a sufficient number of servers, then the attacker will be able to reconstruct all distributed secrets. The third case considers how to provide a better security guarantee against such an attacker.

1.1 Key compromise in web PKI

Public key cryptography is widely used in network protocols to secure communications. To ensure security, it is important to use the correct public keys of the communication parties. For example, suppose a user wishes to log in to her bank account through her web browser. The web session will be secured by the public key of the bank through the TLS protocol [DR08, TP11]. If the user's web browser accepts an inauthentic public key for the bank, then the traffic (including log-in credentials) can be intercepted and manipulated by an attacker.

The authenticity of keys is assured at present by *certificate authorities* (CAs). In the given example, the browser is presented with a public key certificate for the bank, which is intended to be unforgeable evidence that the given public key is the correct one for the bank. A public key certificate is a digital document declaring that the recorded subject owns the public key presented in the certificate. It contains a public key, the identity of the key owner, and a signature of an entity that has verified the certificate's contents are correct. In a typical web PKI scheme, the signer is a trusted party called certificate authority (CA), usually a company (e.g. VeriSign and Comodo) which charges customers to issue certificates for them. The user's browser is pre-configured to accept certificates from certain known CAs. A typical installation of Firefox has about 100 root certificates in its database. Each root CA can empower many intermediate CAs. The EFF SSL observatory has observed more than 1500 CAs [The].

Unfortunately, the CA model is broken. The main weakness of the CA model is that CAs must be assumed to be trustworthy. If a CA is dishonest or compromised, then the CA's private key may be misused to issue certificates asserting the authenticity of fake keys; those keys could be created by an attacker or by the CA itself. In practice, the assumption of honesty does not scale up very well. As already mentioned, a browser typically has hundreds of CAs registered in it, and the user cannot be expected to have evaluated the trustworthiness and security of all of them. This fact has been exploited by attackers [Eck11, Ley12, MS0, Rob11, Ste11, FMC11]. In 2011, two CAs were compromised: Comodo [App11] and DigiNotar [Bla12]. In both cases, certificates for high-profile sites were illegitimately obtained (e.g. Google, Yahoo, Skype, etc.). In the second case, these certificates reportedly used in a *man in the middle* (MITM) attack [Art11].

Another problem with the CA model is the *certificate revocation management*. When a mis-issued certificate is detected, or when a private key associated to a genuine certificate is lost or compromised, then this certificate should be revoked immediately. The CA model itself does not provide any effective way for managing certificate revocation. In common practice, Certificate Revocation Lists (CRL) [CSF$^+$08, Riv98, Lan12], On-line Certificate Status Protocol (OCSP), and certificate revocation trees [Koc98, NN98, LK12] are used to handle certificate revocation. In order to remove the need for on-the-fly revocation checking, they mostly involve periodically pushing revocation lists to browsers. However, such solutions create a window during which the browser's revocation lists are out of date until the next push.

Assuming an attacker is able to obtain a copy of CAs' private keys, *Case 1* considers how to securely manage public key certificate issuance and revocation.

1.2 Key compromise in secure communication

Encryption is the main mechanism used to protect the confidentiality of messages sent between computers. It relies on the assumption that the computer end-points can securely store and use cryptographic keys. If this assumption does not hold, then encryption does not guarantee confidentiality. Yet, this assumption is rather hard to justify in practice. New software vulnerabilities [CVE] are discovered every day, and malware is pervasive on mobile devices such as phones and tablets [FFC$^+$11] as well as on traditional platforms like desktop PCs. Although the security architecture of mobile devices running Android and iOS is an improvement over the PC security architecture, thanks to better security sandboxing of apps that limits attacks spreading between apps, it does not seem likely that completely secure platforms will be built soon.

Assuming an attacker is able to obtain a copy of private keys of a user or a server, *Case 2* considers how to reduce the damage caused from compromised keys by detecting unauthorised usage of private keys.

1.3 Key compromise in secret distribution

Secret distribution is a concept for distributing a secret amongst a group of servers. It enables a secret owner to distribute his secrets to many servers and reconstruct them when needed. One way to achieve it is using secret sharing schemes. A secret sharing scheme allows a user to split a secret into shares, so that each share is held by a server. Then, when the user wants to retrieve the secret, the servers can combine their shares to recover the data. Unfortunately, if the servers become compromised (say by malware) one by one over a long period, then an attacker would eventually be able to compromise sufficiently many servers, and use the accumulated shares to reconstruct the secrets.

Assuming an attacker is able to gradually obtain a copy of all secrets of servers over a long time period, *Case 3* considers how to protect the confidentiality of distributed secrets.

1.4 Aims and Contributions

Aims

The aim of this thesis is providing potential solutions to the above identified problems. The main research objectives are:

- to identify and define adversary models and security goals for each of the above problems;
- to design cryptosystems to achieve our defined security goals;
- to formally verify the security of proposed solutions.

Contributions

The contributions to each of the three cases are listed as follows:

Case 1

- We identify new properties for web certificate management, and provide a critical analysis on the existing web PKI alternatives. In particular, we classify 15 prominent proposals into four categories, and provide a qualitative analysis on selected proposals based on 16 identified criteria. (See Chapter 3.)
- We propose a new system for managing web certificates, called *Distributed Transparent Key Infrastructure* (DTKI). It prevents attacks that use mis-issued certificates, provides a transparent way for certificate management, verifies output from trusted parties, and is secure even if all service providers collude together. In addition, we provide an evaluation on its performance, a comparison between the proposed system and its predecessors, and discussions on variety of concerns related to DTKI. (See Chapter 4.)
- We provide formal machine-checked verification of the core security property of DTKI by using the TAMARIN prover [MSCB13]. Loosely speaking, the core security property guarantees that if the required (crowd-sourced) checkings have been successfully verified, and domain owners have successfully verified their initial certificate, then DTKI

can prevent attacks from an adversary with compromised keys from trusted parties. Otherwise, DTKI enables users to detect attacks afterwards. (See Chapter 4.)

Case 2

- Our first contribution to Case 2 develops the idea of *malware damage detection and containment*. This recognises that no architecture is immune from software vulnerabilities and consequent malware, and therefore it is useful to find new ways of limiting the impact that they have. We complement traditional software mitigation techniques (such as sandboxing and privilege limitation) by enabling a victim to detect that private keys have been compromised. To make this precise, we develop an attacker model in which platforms are *periodically compromised*. That means that they can be compromised by an attacker at any time, but we assume that the victim periodically takes steps to remove malware and eliminate vulnerabilities. Unfortunately, the compromise could have revealed long-term keys. We thus propose security goals that aim to detect the subsequent usage of such keys by the attacker. (See Chapter 5.)
- Second, we propose an approach for a messaging application to transparently manage ephemeral encryption/decryption keys. This approach is simple but effective. It detects subsequent usage of compromised long-term keys by the attacker, while avoiding the use of expensive and inconvenient manual process for re-authenticating and distributing keys through the underlying PKIs (e.g. applying for a new certificate from a CA), unless attacks are detected.

  We call this approach "key usage detection" (KUD), and we develop two protocols for it. The first is a basic protocol that makes strong assumptions about the participants being simultaneously on-line, and serves mostly to explain the concepts. The second protocol is a more fully developed messaging application, supporting multiple devices per user and allowing the receiver to be offline at the time the sender sends a message. (See Chapter 5.)
- Our third contribution to Case 2 is the security analysis, which shows

that the protocols satisfy precise properties expressing software damage containment. Informally, if an attacker controlled device has been recovered from a compromised state to a secure state, then our system can either guarantee the confidentiality of the communications during the secure state; or automatically detect the fact that the long term key is compromised and has been used by an attacker, and therefore the victim will be prompted to manually revoke the key and generate a new one. We use the TAMARIN prover to prove several key properties of our protocol. (See Chapter 5.)

Case 3

- We introduce a scheme based on bilinear pairings for distributed cloud storage, which we call "self-healing" distributed storage. It satisfies a list of requirements that is necessary for secure secret distribution. It is also optimal in round communication between a client and servers, i.e. it requires only one round communication per-server in both phases for data distribution and for data reconstruction, and does not require any client involvement for the periodic update. In addition, it requires only two exponentiation operations on the client side for data encryption or reconstruction, and provides a proactively secure channel. One notable feature of the system is that even though the service secrets change in each time period, the public key to be known by data owners remains constant. This feature could be used as a building block that allows us to tackle a more general server authentication issue, where the servers are compromisable cross time periods. (See Chapter 6.)
- We formalise a security model for this kind of "self-healing" system. Since there might be robust malware that cannot be removed from a server, our security model allows the adversary to permanently compromise servers. We provide a rigorous formal security proof of the proposed system under the defined security model. Our proof also shows that the proposed scheme provides IND-CCA2 security (See Chapter 6.)
- To the best of our knowledge, the proposed system is the first secure self-healing distributed storage, with formal security model and proof.

(See Chapter 6.)

1.5 Thesis structure

We proceed this thesis in the following way. After introducing the background in Chapter 2, we present our main work in three parts, namely Part II, Part III, and Part IV.

Part II presents our solutions to Case 1, by first evaluating the existing PKI systems in Chapter 3, and proposing our system with formal security proof in Chapter 4. Chapter 3 identifies desired features and security concerns of web certificate management systems, classifies the existing proposals for certificate management into four categories, and evaluates these proposals and concludes the observed characters of different categories. Chapter 4 proposes a new system for certificate management, called DTKI, with formal security analysis, performance evaluation, and a comparison with its predecessors. The full code required to understand and reproduce our security analysis is presented in Appendix A. In particular, the code also includes the complete DTKI models.

Part III presents our solutions to Case 2. It, in Chapter 5, formalises the associated security model, and details our basic key usage detection (KUD) approach with a more fully developed messaging application. A formal security proof and performance evaluations of the messaging application are also presented in the same chapter. Similar to the previous part, the code required to understand and reproduce our security analysis of the KUD messaging application is presented in Appendix B.

Part IV presents our solutions to Case 3 in Chapter 6. Before proposing our self-healing scheme for distributed storage, it first formalises a security model for such systems. After introducing our scheme in great detail, we formally prove the security of the proposed scheme under the defined security model.

Finally, Part V concludes the thesis.

CHAPTER 2

BACKGROUND

This chapter introduces related background knowledge. It first presents some crypto preliminaries, and then introduces TAMARIN prover — a tool for automatic security protocol verification.

2.1 Crypto preliminaries

The following problems are assumed hard in cryptography.

2.1.1 Discrete Logarithm Problem

The discrete logarithm problem is a significant element in cryptology, and is the root of many cryptographic security assumptions. Let G be a cyclic group of order p, and g be a generator of G.

Definition 2.1 (Discrete Logarithm (DL) Problem). *Given (g, X) such that $X = g^x$ for some random $x \in \mathbb{Z}_p^*$, the discrete logarithm problem is to compute x.*

2.1.2 Diffie-Hellman Problem

The Diffie-Hellman problem was proposed by Diffie and Hellman in 1976 [DH76].

Definition 2.2 (Diffie-Hellman Problem). *Given (g, g^a, g^b) for some random $a, b \in \mathbb{Z}_p^*$, the computational Diffie-Hellman problem is to compute g^{ab}.*

There are several versions of this problem. We normally refer the Diffie-Hellman problem as computational Diffie-Hellman (CDH) problem. The decisional version of the CDH problem [Bon98] is defined as follows.

Definition 2.3 (Decisional Diffie-Hellman (DDH) Problem). *Given (g, g^a, g^b, Q) for some random $a, b \in \mathbb{Z}_p^*$ and $Q \in G$, the decisional Diffie-Hellman problem is to distinguish between $Q = g^{ab}$ and Q random.*

Another related problem is called "q-Diffie-Hellman inversion (DHI) problem" [BB04a], defined as follows.

Definition 2.4 (q-Diffie-Hellman Inversion (DHI) Problem). *Given $(g, g^x, g^{x^2}, \ldots, g^{x^q}) \in (G)^{q+1}$, for x random the q-Diffie-Hellman inversion problem is to compute $g^{1/x}$.*

2.1.3 Bilinear paring

We first define a bilinear map, as follows.

Definition 2.5 (Bilinear Map). *Let G_1, $\hat{G}_1$ be two cyclic groups of a sufficiently large prime order p. A map $e : G_1 \times \hat{G}_1 \to G_2$ is said to be bilinear if $e(g^a, h^b) = e(g, h)^{ab}$ is efficiently computable for all $g \in G_1$, $h \in \hat{G}_1$ and $a, b \in \mathbb{Z}_p^*$; and e is non-degenerate, i.e. $e(g, h) \neq 1$.*

In the above definition, when $G_1 = \hat{G}_1$, the pairing is called symmetric. Let $e : G_1 \times G_1 \to G_2$ be a bilinear map, and g be a generator of G_1 whose order is p. We define the bilinear Diffie-Hellman (BDH) problem [BF03], the decisional bilinear Diffie-Hellman (DBDH) problem [Jou02], and the q-decisional bilinear Diffie-Hellman inversion (q-DBDHI) problem [BB04b], as follows.

Definition 2.6 (Bilinear Diffie-Hellman (BDH) Problem). *Given (g, g^a, g^b, g^c) for some random $a, b, c \in \mathbb{Z}_p^*$, the bilinear Diffie-Hellman problem is to compute $e(g, g)^{abc}$.*

Definition 2.7 (Decisional Bilinear Diffie-Hellman (DBDH) Problem). *Given (g, g^a, g^b, g^c, Q) for some random $a, b, c \in \mathbb{Z}_p^*$ and $Q \in G_2$, the decisional bilinear Diffie-Hellman problem is to distinguish between $Q = e(g, g)^{abc}$ and Q random.*

The q-Decisional Bilinear Diffie-Hellman Inversion (q-DBDHI) problem was first used by Boneh and Boyen [BB04b], and proved to be difficult in the generic group mode by Dodis and Yampolskiy [DY05].

Definition 2.8 (q-DBDHI Problem). *Given $(g, g^x, g^{x^2}, \ldots, g^{x^q}) \in (G_1)^{q+1}$ and $Q \in G_2$, the q-DBDHI problem is to distinguish between $Q = e(g, g)^{1/x}$ and Q random.*

2.1.4 Secret sharing

Secret sharing is a protocol for distributing a secret amongst a group of participants. Each of the participants will obtain a share of the secret, in the way that the secret can only be reconstructed when at least a sufficient number of shares are combined together. The smallest sufficient number is called a "threshold" of the system. If the size of the group is n and the threshold is t, then such a system is called (t, n)-threshold secret sharing scheme.

The Shamir secret sharing scheme [Sha79] is one of the classic secret sharing schemes. The main idea of Shamir secret sharing scheme is derived from the polynomial interpolation. For example, one needs at least two points to define a line. So, if the line is a secret, and points on the line are the shares, then it is a $(2, n)$-threshold secret sharing scheme. Similarly, t points are sufficient to define a polynomial of degree $t - 1$. Thus, it can be applied to obtain a (t, n)-threshold secret sharing scheme.

Let $\mathcal{F}_q$ be a finite field of order q, such that q is a prime power and $q > n$. The $(t - n)$-Shamir secret sharing is explained as follows.

Distributing a secret The secret s can be distributed as follows:

- choose a random polynomial $f \in \mathcal{F}_p$ of degree $t - 1 < n$, such that $f(0) = s \in \mathcal{F}_p$;
- generates shares for each of the participants, such that the ith-share for the ith-participant is $(i, f(i))$, where $i \in \{1, 2, \ldots, n\}$, and

  $$f(i) = s + \sum_{j=1}^{t-1} c_j \cdot i^j \mod p$$

  where all $c_j \in \mathcal{F}_p$ are random coefficients.

Reconstructing a secret Given any t out of n shares, let's say $(i, f(i))$ for $i \in \{1, 2, \ldots, t\}$. The secret s can be recovered as follows:

- reconstructs the polynomial f by applying Lagrange interpolation to the received shares, such that

$$f(x) = \sum_{i=1}^{t} f(i) \prod_{j=1, j\neq i}^{t} \frac{x-j}{i-j} \mod p$$

- recovers the secret by computing $f(0) \mod q$, such that

$$f(0) = \sum_{i=1}^{t} f(i) \prod_{j=1, j\neq i}^{t} \frac{-j}{i-j} \mod p$$

2.1.5 Knowledge Proof

The proof of knowledge is a protocol enabling a prover to convince a verifier that the prover knows some secrets without showing the secrets to the verifier. Zero-knowledge proof enables a prover to convince a verifier the truth of an assertion, without revealing anything but the validity of proof. If a proof of knowledge is also a zero knowledge proof, then it is called a zero-knowledge proof of knowledge (ZKPK). The Schnorr identification protocol [Sch89] is one of the simplest and frequently used honest verifier zero-knowledge proof of knowledge.

Let G be a cyclic group of order p, and $g \in \mathbb{Z}_p^*$ be a generator of G. The prover's goal is to prove that $y = g^x$ for some x. The Schnorr identification protocol is described as follows:

- The prover generates a random number $r \in \mathbb{Z}_p^*$, commits to r by sending $a = g^r$ to the verifier;
- The verifier generates a challenge $c \in \mathbb{Z}_p^*$ and sends it to the prover;
- The prover computes the proof $t = r + cx \mod p$;
- The verifier verifies the proof by checking whether $g^t = ay^c$ or not. If $g^t = ay^c$, then the proof is valid. Otherwise, the proof is invalid.

The above interactive honest verifier ZKPK can be transferred to a non-interactive version through a Fiat-Shamir transformation [FS86]. To do so, rather than asking the verifier to generate a challenge c, the prover computes $c = \mathsf{h}(m, a)$, where h is a secure cryptographic hash function, and m is the message contains g, y, the prover's identity, and possibly some other information. So, a verifier can recompute c and verifies the proof in the same way. The Schnorr signature scheme is one of its application, where m is the to be signed message.

Another proof of knowledge related to this thesis is the Chaum-Pedersen protocol [CP92]. It is a protocol for proving the equality of two discrete logarithms. The application of which in our thesis is to prove that (g, g^x, g^y, g^{xy}) is a DDH tuple.

Let G be a cyclic group of order p, and g be a generator of G. The prover's aim is to prove that (g, y, w, u) is a DDH tuple, such that $(g, y, w, u) = (g, g^x, g^r, g^{rx})$. We describe the Chaum-Pedersen honest verifier ZK protocol as follows.

- The prover generates a random $s \in \mathbb{Z}_p^*$, and sends $(a = g^s, b = w^s)$ to the verifier;
- The verifier generates a random challenge $c \in \mathbb{Z}_p^*$, and sends it to the prover;
- The prover calculates the proof $t = s + cx \mod p$ and sends the proof to the verifier;
- The verifier verifies the proof. If $g^t = ay^c$ and $w^t = bu^c$, then the proof is valid. Otherwise, the proof is invalid.

To make the above proof non-interactive, rather than letting the verifier to generate a challenge c, the prover can generate $c = \mathsf{h}(w, u, a, b)$. The proof (c, t) is valid if $c = \mathsf{h}(w, u, \frac{g^t}{y^c}, \frac{w^t}{u^c})$. Otherwise, the proof is invalid.

2.2 TAMARIN **Prover**

TAMARIN prover is a symbolic security protocol verification tool. It supports an unbounded number of instances, and supports reasoning about protocols with mutable global state. The adversary in TAMARIN prover is in Dolev-Yao model, i.e. it carries the message exchanged in the protocol. With TAMARIN prover, cryptographic functions are assumed to be secure.

In TAMARIN prover, protocols and adversary models are specified using multiset rewriting rules on the multiset that models the state of the protocol. Security properties are expressed in a guarded fragment of first order logic that allows quantification over timepoints.

A multiset is a generalisation of set, allows multiple instances of elements in the multiset. An element contained in the multiset is called a "fact", which is a ground message, or a message derived from a ground message according to the rules. A fact is of the form $\mathbf{F}(t_1, \ldots, t_k)$, where $\mathbf{F}$ is the fact symbol, and $(t_1, \ldots, t_k)$ are k terms for some k. A fact is either linear or persistent. A linear fact can only be consumed once, whereas a persistent fact can be consumed arbitrarily. There are some reserved facts. A persistent fact $\mathbf{K}(m)$ denotes that the adversary has the knowledge of m. Linear facts $\mathbf{In}(m)$ and $\mathbf{Out}(m)$ denote that message m is sent or received by the protocol, respectively. A linear fact $Fr(n)$ denotes that a name n is freshly and randomly generated. A fresh generated name is unknown to the attacker.

A rule, denoted $[l] - [a] \rightarrow [r]$, is a sequences of facts, where the set l of facts is called premises, the set a of facts is called actions, and the set r of facts is called conclusions. Rules can only be applied if all the premises are in the multiset. A rule application can rewrite its premises, and can introduce new facts.

In TAMARIN prover, a theory specifies a security protocol together with its security properties; a lemma is a rule to specify only security properties; and an axiom is a rule to specify an assumption of the protocol. Since TAMARIN's property specification language is a fragment of first-order logic, it contains logical connectives (`|`, `&`, `==>`, `not`, ...) and quantifiers (`All`, `Ex`). The `#`-prefix is used to denote timepoints, and "`E @ #i`" expresses that the event E occurs at timepoint i.

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

Part II

Key compromise in web PKI

CHAPTER 3

EVALUATING WEB PKIS

3.1 Introduction

Certificate authorities serve as trusted parties to help secure web communications. They verify the identity of domains, and issue certificates declaring that the binding between a domain name and a public key is correct. Web browsers accept such a binding if the associated certificate is valid and issued by a certificate authority. Unfortunately, recent attacks using mis-issued certificates show this model is severely broken. Much research has been done to enhance web certificate management. However, none of it has been widely adopted yet, and it is hard to judge which one is the winner.

To analyse the existing issues and evaluate existing proposals, Clark and Van Oorschot [CvO13] have presented an analysis on TLS mechanism and issues, by concerning themselves with crypto weakness and implementation issues of HTTPS, and trust issues of certificate management. However, they left the log-based certificate management systems out of the analysis. The use of public logs is now the dominant trend in managing web certificates. The main idea of log-based certificate management systems is to make certificate management transparent by using public audit-able logs to record all issued certificates. Clients will only accept a certificate if it is recorded in the log. Site owners can compare their own local record with the log to check whether a mis-issued certificate has been recorded in the log. This gives the site owners the ability to verify issued certificates for their sites, and make the certificate management transparent.

Kim et al. [KHP^{+}13] have presented a comparison of web certificate management mainly based on the duration of compromise and duration of unavailability. The former shows, given the compromise of a domain's private key, how long the domain can be impersonated; and the later concerns the unavailability time period of a domain's certificate in a system.

The above two works are broad and they evaluate web certificate management systems from different perspectives. However, some important aspects are not considered in the existing work. For example, *offline verification* is one of the desired properties that have been left out from the above analyses. This property ensures that internet users can verify a received certificate without having to communicate with other parties. This is extremely useful when a user needs to connect from a captive portal in an airport or in a hotel, since the user's device cannot make other connections before they paid for the internet connection. In addition, this property also reduces the communication cost for certificate verification, as the verifier is not required to have extra connections for verifying a certificate.

Another important property not considered by the existing works is *trust agility* [Mar11] — it allows users to freely make decisions on which certificate management service provider they wish to trust, for establishing secure communications with domain servers. In particular, we discovered a new aspect of trust agility, namely independence of trust. It requires that one or more service providers cannot not influence another service provider's services. It is in particular useful in the scenario where there exists a set of service providers, and users need to put their trust in a subset of these service providers for certificate management. If a system does not offer this feature, then it means that even if the set of service providers chosen by a user is trustworthy, a malicious service provider that is not trusted by the user can still influence the certificate verification result, and put the user in the risk of accepting fake certificates. Since the independence of trust is more strict, it is possible that a system offers the generic trust agility, but it does not offer independence of trust. In this case, users are free to make their trust decisions, but servers that are not trusted by the user are still able to affect the certificate management services delivered to the user.

In addition, we observed the problem of *oligopoly*, which has not been considered before. The present-day certificate authority model requires that the set of global certificate authorities is fixed and known to every browser, which implies an oligopoly. Currently, the majority of CAs in browsers are organisations based in the USA, and it is hard to become a browser-accepted CA because of the strong trust assumption that it implies. This means that a Russian bank operating in Russia and serving Russian citizens living in Russia has to use an American CA for their public key. This cannot be considered satisfactory in the presence of mutual distrust between nations regarding cybersecurity and citizen surveillance, and also trade sanctions which may prevent the USA offering services (such as CA services) to certain other countries.

To help research in securing the web certificate management, and to have a better understanding on the problem, we classify prominent proposals into different categories, and provide a qualitative analysis on selected proposals based on 15 criteria.

3.2 Desired Features and security concerns

To evaluate different systems in a systemic way, we list the desired features and security concerns for web certificate management systems.

1. Trust

   - *Trust agility* [Mar11] allows users to freely decide which entities they want to trust for confirming public key information of domain servers, and to revise their decision at any time.

     In particular, we observed one aspect of the trust agility that has not been discovered in the literature, namely the independence of trust. It requires that the trust relations between service providers will not influence the trust relations between clients and the service providers they trust. In other words, one or more service providers cannot not influence another service provider's service to its clients.

   - *Free of trusted parties* is the property says that no party is required to be trusted for certificate issuance and revocation. This is the strongest one in all trust-related features.

- *Verifiable trusted parties* is the property says that the behaviour of trusted parties is transparent to and can be efficiently verified by users.
- *Anti-oligopoly* is a new property we identified. It prevents the monopoly or oligopoly of certificate management services.

  To achieve anti-oligopoly, the trust on any service provider (e.g. CAs) should be minimised, and the system should support self-issued certificates.

2. Availability

   - *Offline verification* is a feature such that in a system, clients can verify a given key or certificate without having to connect from other parties.

     This feature is desired when a user needs to connect from a captive portal— a login page or payment page — before using the internet. The use of captive portal is very common in public places, for example, airports or hotels. When a user is presented a captive portal, the user cannot establish a connection with any party to check the obtained public key as no internet is available. In addition, this feature also reduces the communication cost and network latency, as it does not require additional connections.
   - *Built-in key revocation* requires the system to have its own mechanism to effectively manage certificate revocation, rather than relaying on existing revocation protocols (*e.g.* certificate revocation list (CRL) or on-line certificate status protocol (OCSP)).

     The current certificate revocation management protocols (e.g. CRL and OCSP) have different limitations and cannot offer satisfactory services. So it is necessary for systems to have an integrated revocation mechanism to effectively manage certificate revocations.
   - *Scalability* is the property enabling a system to handle increasing real world workload. It is important that a system is capable to support enrolment from existing and potential future HTTPS servers.
   - *Multiple certificate support* says that the certificate verification system

allows a domain to have multiple certificates. The fact that many sites have multiple certificates emphasises the importance of this feature.

- *Timely key verification* says that the period from the time a domain owner establishes a key and the time a user can verify the key is short.

  This is a feature that has not been prominent in the literature. This feature is useful when a domain server updates its certificates. A system that does not offer this feature would cause the problem that the newly issued certificate cannot be verified and will not be accepted by web browsers within a short time period after the certificate issuance. This reduces the availability.

3. Security

- *First connection protection* is the feature that protects the first connection between two communication parties.

  This is useful to prevent attacks on 'trust on first use'-based systems. In addition, it is likely to be the first connection when a user connects from a captive portal. So the system should protect users' first connection to a domain server.

- *Denial of service (DoS) attack protection* is the security guarantee that prevents attacks on the key verification infrastructure in order to denial the verification services.

  This feature is useful to prevent attacks that attempt to block the verification servers to stop users verifying the received certificates.

- *Use of mis-issued certificate prevention* measures whether the system can prevent MITM attacks launched by an attacker with mis-issued certificates. In other words, even if an attacker has obtained a mis-issued certificate, web browsers should still not accept this certificate. This gives users extra security guarantee against compromised CAs.

- *Use of mis-issued certificate detection* measures whether the system provides features allowing one to detect MITM attacks launched by using mis-issued certificates.

This is a weaker security guarantee, as it can only detect attacks rather than prevent attacks. However, CAs are business, and they are willing to maintain their reputation to keep their customers. So they might not launch attacks if their attacks will be detected. So this feature still offers some sensible security guarantee.

- *Provably secure* measures if the security of a given system is formally verified.

  It is well-known that security protocols are notoriously difficult to get right, and the only way to avoid this is with systematic verification.

4. Usability

   - *No user involvement* is a feature related to usability, such that the key verification result and the decision of accepting or rejecting a certificate do not need the extra involvement of users.

     This is an important feature to have, as users are not qualified to make decisions on the browser warnings, and they will likely to click through security warnings [AF13].

5. Privacy

   - *Protecting browsing history* says that the system does not leak users browsing history to other parties. In a PKI, if a user needs to ask another party to verify a received certificate, then the user's browsing activity is leaked to the verification party, as the subject of the to be verified certificate would very likely to be the website that the user is going to visit.

3.3 Analysis of existing proposals

Several protocols are proposed to strengthen the current certificate management system. According to the principles of each design, we classify leading certificate management systems into three categories, namely *difference observation*, *scope restriction*, and *certificate management transparency* (shown in Table 3.1).

Table 3.1: Existing prominent proposals

Category	Existing Proposals
Classic	CA-based certificate management system;
Difference observation	Perspectives ('08) [WAP08]; DoubleCheck ('09) [AK09];
	Convergence ('11) [Mar11]; Certificate Patrol ('11) [Cer]; CertLock ('11) [SS11]; TACK ('12) [MP12].
Scope restriction	Public key pinning ('11) [Lan11]; DANE ('11) [Bar11b];CAge ('13) [KWH13].
Certificate management transparency	Sovereign Keys ('12) [Eck12]; Certificate Transparency ('12) [LLK13];
	AKI ('13) [KHP^{+}13]; CIRT('14) [Rya14]; ARPKI ('14) [BCK^{+}14]

3.3.1 Classic CA model

CA-based certificate management system is the current deployed PKI. It is highly usable and scalable. Unfortunately, it requires users to fully trust all certificate authorities, and the trust cannot be modified without sacrificing users' ability to securely communicate with some domains securely. As a result, it does not provide trust agility, implies an oligopoly (on CA), and cannot easily prevent nor detect MITM attacks using mis-issued certificates.

3.3.2 Difference observation

Difference observation is a concept aiming to detect untrustworthy CAs, by enabling a browser to verify if the received certificates are different from those that other people are being offered [WAP08, AK09, Cer, SS11, Mar11].

Perspectives In 2008, Wendlandt, Andersen and Perrig implemented a Firefox addon, called *Perspectives* [WAP08]. It is proposed to improve the security of trust-on-first-use authentication by asking different observers (a.k.a. notary servers) to detect inconsistent public keys of the same server. In Perspectives, observers are decentralised and independent. Each observer stores all observed keys or certificates with corresponding timestamps, and periodically checks updates and revocations. When a client wants to make a secure connection with a domain server, the client requests the server's public key from the server and from multiple observers, then compares the received keys. If the obtained public keys are consistent, the client considers the public key is trustworthy and uses this key to establish a secure connection. Otherwise, it might indicate that an attacker has launched man-in-the-middle (MITM) attack by offering a different public key to the client. So the client needs to make a decision on whether to use the obtained key or not.

- **Strength**
  Perspectives makes MITM attacks using mis-issued certificates difficult to launch without being detected, as an attacker would have to additionally intercept all connections between observers and the victim. In addition, it provides trust agility as users can choose which observer they want to use for certificate verification. Moreover, since it supports self-signed certificates, and does not require a fixed set of observers, it provides anti-oligopoly.

- **Weakness**
  With Perspectives, if a server has multiple public keys or certificates, then clients will likely get a warning of receiving inconsistent public keys. This is due to the fact that a client might receive two different genuine certificates of the same domain from the domain server and an observer. In addition, a new public key or a new server will suffer an unavailability period in the system. Since observers periodically check new public keys and revocations, the latest information about new public keys and revocations will not be immediately available from the observers. So, Perspectives does not offer timely key verification. Also, when a browser receives the latest genuine one from the server, and the revoked one from observers, then the browser will show a pop-up window warning the user that two different keys are observed, although what the server provided is a valid certificate. Such faulty warnings reduce usability of the system. Moreover, if two different certificates are detected, then the user needs to make a decision on whether to continue the connection. However, users are not qualified to make such a decision and they are likely to click through the warnings [AF13]. Furthermore, any observer can learn a user's browsing history when the user requests verification on a certificate. Last, it does not work when a user needs to connect from a captive portal, as no internet is available for connecting to an observer.

DoubleCheck In 2009, Alicherry and Keromytis [AK09] proposed *DoubleCheck* to solve the issue of leaking user browsing history, and the issue that new keys might suffer an unavailable period in Perspectives. The main idea is to query the certificate from a target server twice: once through a TLS connection, and once through *Tor* [DMS04].

- **Strength**
  Compared to Perspectives, it additionally protects user browsing history, and new keys does not suffer an unavailability period. Moreover, it can be deployed without requiring any new infrastructure.

- **Weakness**
  The use of *Tor* adds extra time cost (up to 15 seconds [SS11]) for each certificate verification. In addition, a use is likely to get a warning when a server has multiple certificates. Also, when a warning is given, a user will need to make a decision on which certificate to trust, and they are likely to click through the warning. Moreover, it will not work when a user needs to connect from a captive portal.

Convergence Marlinspike proposed *Convergence* [Mar11], a Firefox addon and an improvement on *Perspectives*, in Black Hat 2011. In Convergence, to protect users' browsing history, instead of directly communicating with notary servers (i.e. observers), users randomly choose one notary server to pass the client request to other notary servers, through an onion routing like mechanism. So the intermediate notary server does not know what a requester is requesting, and the end notary server does not know who is the requester. In addition, to reduce the number of connections a user has to make, users store verified certificates in their browser cache and only query notary servers when they received a different one. Moreover, rather than querying the certificate of a domain server from a notary server, users send the certificate received from the server to notary servers. The notary server will request a certificate from the domain server if the received certificate does not match the notary's cache.

- **Strength**

As an improvement of Perspectives, it additionally supports timely key verification, does not require user to make decisions on which certificate to trust, and it protects user privacy. Moreover, it offers offline verification if the site has been visited before.

- **Weakness**
  Similar to Perspectives, Convergence does not support multiple certificates, and does not protect users when they are connected to a captive portal.

Certificate Patrol Certificate Patrol [Cer] is another Firefox add-on for managing web certificates. It monitors and stores all SSL certificates a browser has obtained. Since the validity period of a certificate is fairly long, it is unlikely a certificate is changed in a short time. So, when a different certificate is observed, it is possible that one of them is a mis-issued certificate used by attackers. With Certificate Patrol, if the newly received certificate is different from the previously stored certificate of the same domain, the browser will display to the user the difference between the two certificates, and the user needs to make a decision on whether to trust the newly received one.

- **Strength**
  It is a lightweight tool to protect user browsing history, and to offer an extra layer of security – it helps users to detect any change of the previously received certificate.

- **Weakness**
  This addon will not work if a domain has multiple certificates, and it requires users to make decisions. In addition, it does not protect user's first connection to a website nor protect user connection from a captive portal.

CertLock CertLock [SS11] is a Firefox addon for monitoring CAs' location. In particular, it observes the country of the CA who issued the received certificate. On the detection that two CAs from different countries have issued certificates for the same site, the browser will display a warning to the user.

- **Strength**
  CertLock helps users to detect attacks in some specific scenario. For example,

a site authorised certificate authority CA_1 in country A to issue certificate for its domain. A malicious government agency in country B wants to intercept the communication between users and the site. The malicious government agency can compel a certificate authority CA_2 located in country B to issue fake certificates for the site, then uses this mis-issued certificate to launch MITM attacks. CertLock can help users to detect such attacks.

- **Weakness**
  CertLock won't be able to detect attacks using fake certificates that are issued by CAs in the same country. In addition, a false warning will be displayed if a site has switched from a CA in country A to a CA in country B. In addition, it still relies on the CA trust model, so it does not offer trust agility nor anti-oligopoly. Also, it cannot protect user's first connection and cannot protect a user who is connected to a captive portal.

TACK In 2012, Marlinspike and Perrin proposed *trust assertions for certificate keys* (TACK) [MP12] to remove the need of trusting CAs. In TACK, a domain server generates a TACK private/public key pair, and uses the TACK private key to certify its TLS public keys. After a client observes a consistent TACK public key of a domain multiple times, it pins the public key to the domain name, and trusts this "pin" for a period, and accepts the public key if it is certified by the private key corresponding to the observed TACK public key. If a certificate becomes compromised and the observed information has not been pinned, then the client must delete the observed TACK information and re-start the observation process. To be scalable, TACK will need an online pin store, where users can share their observed pins. However, the problem of how to design a secure pin store for users to share their observations, while prevent attackers to spoof or poison the store, remains unsolved.

- **Strength**
  TACK removes the need of CA, offers trust agility, does not require users to any trusted party[1], and provides anti-oligopoly. Once local observations are built, TACK allows offline verification, supports multiple certificates.

[1]Here, we only consider the TACK without having an online pin store

- **Weakness**
  Since TACK relies on visit patterns by clients to pin the domain's public key, the first several connections to a domain server will not be protected, and every new TACK key pair or new domain suffers an initial unavailability period. In addition, the revoked key will still be accepted by the client if the client still trusts its previous observation.

  To be scalable, TACK requires an on-line store to share TACK keys observed by different clients. The use of such on-line stores make TACK difficult to provide the independence of trust required by trust agility. Because a client Alice, might choose to trust some stores or clients for the TACK keys they observed. However, the store or clients trusted by Alice might put their trust on other stores and clients. This transitive trust relation could effect Alice's trust option and Alice's observation on the TACK keys. Currently, it is hard to judge whether TACK offers the independence of trust required by trust agility, as the online store is not designed yet.

3.3.3 Scope restriction

Scope restriction is the concept aiming to reduce the power of CAs by restricting the domain scope that a CA can vouch for.

Public key pinning (PKP) Public key pinning (a.k.a. certificate pinning) is a mechanism for domain servers to specify which CAs are authorised to certify public keys for a given domain. Langley et al. implemented it in Google Chrome [Lan11].

Scalability is a main challenge for key pinning, due to the need of pre-knowledge of the mapping between each domain server and CAs. Public key pinning extension for HTTP [EPS14] addresses the scalability challenge by allowing a domain server to declare the authorised CAs for its sites in an HTTP header.

- **Strength**
  As PKP is a way to restrict CAs' power by specifying which CAs are authorised for a given website, it protects user communications against attackers who have mis-issued certificates from CAs that are not authorised for the victim.

In addition, PKP allows a website to have multiple certificates, does support offline verification, and is scalable with the PKP extension for HTTP.

- **Weakness**
  The weakness of PKP is that it cannot completely protect all user connections. For example, it cannot protect when a user does not have a pin of a website, which is generally the case for the first connection. Also, it cannot protect the connection when the pin is expired in the user browser. Moreover, it cannot effectively detect attacks when a CA has mis-issued certificates for the domains that the CA is pinned for. Furthermore, it does not offer trust agility nor anti-oligopoly.

DANE *DNS-based authentication of named entities* (DANE) [Bar11a, HS12] binds the public key information to a domain name by using Domain Name System Security Extensions (DNSSEC). More specifically, DANE enables a domain server to certify its public keys by storing the public keys in its domain name system (DNS) records. This DNS record is valid only if it is correctly signed as specified in DNSSEC [WB13]. So, the parent domain servers are the authority of their child domains. In other words, only the parent domain can certify public keys of its child domains. In this way, DANE limits the damage of dishonest or compromised authorities.

- **Strength**
  Compared to PKP, DANE is highly scalable since it is based on DNSSEC. In addition, it can protect a user even when the user connects from a captive portal.

- **Weakness**
  The security of DANE strongly relies on the trustworthiness of parent domains according to the DNS hierarchy. As a result, ICANN, top-level domains (TLDs), and second-level domains (SLDs) become to be very powerful and fully trusted CAs. So, DANE does not provide trust agility and anti-oligopoly. In addition, domain servers cannot choose which CA they want to get service from, as they have to get their keys to be certified by their parent domain.

CAge In 2013, Kasten, Wustrow and Halderman proposed *CAge* [KWH13] to restrict the scope of domains that a CA can certify public keys for. According to the data observed in [HDWH12], they show that only a small number of CAs have signed certificates for TLDs. Based on this observation, *CAge* suggests to limit a CA's certification scope by only allowing a CA to issue certificates on a restricted set of TLDs. CAge limits the scale of MITM attacks, but cannot completely solve this problem.

- **Strength**
  As all systems in the category of scope restriction, CAge reduces the damage from a compromised CA by limiting the set of domains that a CA can vouch for.

- **Weakness**
  Since CAge is still based on the CA trust model although with restrictions on a CA's ability, it does not offer trust agility and anti-oligopoly. In addition, domain servers have less flexibility to choose which CA they want to use, because only a subset of CAs will be eligible for certifying keys for given domains.

3.3.4 Certificate management transparency

Certificate management transparency is the concept aiming to make CAs' behaviour transparent. The basic idea is to use a publicly visible log to record issued certificates. So interested parties can check the log to detect any mis-issued certificates.

Sovereign Keys Sovereign Keys (SK) [Eck12] aims to get rid of browser certificate warnings, by allowing domain owners to establish a long term ("sovereign") key and by providing a mechanism by which a browser can hard-fail if it doesn't succeed in establishing security via that key. A sovereign key is a long-term key used to cross-sign operational TLS keys, and it is stored in an append-only log on a "timeline server", which is abundantly mirrored.

When a browser connects to a website, it sends a query to a mirror of the "timeline server" to check if the site has a sovereign key. If the site does have a sovereign key, then the browser only accepts a certificate for this site if the certificate is issued by CAs and is cross-signed by the sovereign key. If the certificate is not cross-signed, then rather than emit certificate warnings, the browser will try to find a way to make a sovereign key connection to the site. There are several ways to establish a connection without having a cross-signed certificate. The strongest way is to compute a hash of the sovereign key, and use that as the .onion address of the Tor hidden service which allows the secure connection. Weaker ways include stapling to the sovereign key and trying to connect through other means such as proxy and VPN, until the browser gets a verified connection.

- **Strength**
  SK introduces the first log-based PKI. It eliminates browser certificate warnings, reduces the trust put on CAs, allow a site to have multiple certificates, and prevents attacks from an attacker who compromised CAs.

- **Weakness**
  Sovereign Keys doesn't have an efficient way for the timeline server and mirrors to prove their correct behaviour. The only way for verifying it is to download an verify the entire log. So internet users and domain owners have to trust mirrors of time-line servers. Additionally, it doesn't provide any mechanism for key revocation, either of TLS keys or sovereign keys. If a domain owner loses the sovereign private key, they lose the ability to switch to new TLS keys, and may even lose control of their domain, until the sovereign key expires. Another security concern is that if a site does not have a sovereign key yet, then a determined attacker could register his own sovereign key for the site and intercept secure connections made to the site.

Certificate transparency Certificate transparency (CT) [LLK13] is proposed by Google aiming to allow domain owners to efficiently detect mis-issued certificates, by making certificate issuance transparent.

The basic idea is to use public audit-able logs to record all issued certificates. In this way, interested parties can monitor the log to verify all of CAs' behaviour. To enforce CAs to publish all issued certificates into the log, web browsers only accept certificates if a verifiable evidence is provided to prove that the certificate is present in the log.

In more detail, domain owners request from the log maintainer signed confirmations saying that their certificates are included in the log, and then they can provide this confirmation together with the corresponding certificate to web browsers. Browsers only accept a certificate if both the certificate and the signed confirmation are valid. Browsers also need to periodically verify received signed confirmation against the public log to check if the certificate is indeed being inserted in the log.

To reduce the trust put on CAs and log maintainers, CT uses an append-only log which is organised as an append-only Merkle tree. In the tree, data items (i.e. certificates or references to certificates) are stored left-to-right in chronological order at the leaves, and added by extending the tree to the right. This structure enables the log maintainer to provide two types of verifiable cryptographic proofs: (a) a proof that the log contains a given certificate, and (b) a proof that a snapshot of the log is an extension of another snapshot (*i.e.*, only appends have taken place between the two snapshot). The time and size for proof generation and verification are logarithmic in the number of certificates recorded in the log. To ensure the log maintainer is behaving correctly, CT requires monitors to check the consistency of logs.

- **Strength**
  Since CAs' behaviour is transparent, CT does not require users to blindly trust CAs, i.e. the behaviour of CAs are verifiable. This makes CT to offer trust agility. In addition, CT enables domain owners to readily detect any mis-issued certificates.

- **Weakness**
  A main weakness of CT is that users still have to trust "monitors" for verifying the behaviour of logs. In addition, CT does not provide an efficient scheme for key revocation. Also, CT does not provide anti-oligopoly, because although the set of log servers are not fixed, it doesn't have any method to allocate different domains to different logs. In CT, when a domain owner wants to check whether mis-issued certificates are recorded in logs, he needs to contact all existing logs, and download all certificates in each of the logs, because there is no way to prove to the domain owner that no certificates for his domain is in the log, or to prove that the log maintainer has showed all certificates in

the log for his domain to him. Thus, to be able to detect fake certificates, CT has to keep a very small number of log maintainers. This prevents new log providers being flexibly created, creating an oligopoly. Another limitation is that CT can only detect mis-issued certificates, rather than prevent attacks that use mis-issued certificates.

Accountable key infrastructure Accountable key infrastructure (AKI) [KHP$^+$13] also uses public logs to make certificate management more transparent.

Similar to SK, AKI allows domain owners to define their own security policy by specifying several additional attributes of a certificate, such as which CA and log maintainer a domain owner wants to get services from, what is the minimum number of CA signatures to validate a certificate for her domain, etc. To obtain a certificate, a domain owner contacts at least a minimum number of CAs that she wishes to trust based on the policy, and to cross sign her public key with her security policy. Then she requests log maintainers to update her certificate, and expects a signed proof that the certificate is recorded in the log. Clients only accept a certificate if the certificate satisfies defined security policy, and is currently recorded in the log.

To be able to manage key revocations, AKI stores only the current valid certificates of domains in a public log. The log is organised as a hash tree, where certificates stored in leaves ordered lexicographically.

To detect mis-behaviours, AKI uses the “checks-and-balances” idea that allows parties to monitor each other’s behaviour. So AKI limits the requirement to trust any party. Moreover, AKI prevents attacks that use fake certificates rather than merely detecting such attacks (as in CT).

- **Strength**
  AKI extends the previous architectures in several ways. First, it allows multiple CAs to sign a single certificate. Additionally, the domain can specify in its certificate which CAs and logs are allowed to attest to the certificate’s authenticity. These features provide resilience against a certificate signed by a compromised or unauthorised CA. AKI can also handle key loss or compromise through cool-off periods. For example, if a domain loses its private key and registers a new certificate not signed by its old private key, the new certificate

will be subject to a cool-off period (e.g., three days) during which the certificate is publicly visible but not usable. This ensures that even if an adversary obtains and registers a fake certificate, the domain has the opportunity to contact the CAs and logs to resolve the issue.

- **Weakness**
  To ensure that any log server can provide a proof for a domain's certificate, AKI logs maintain a globally consistent view of the entries that they have for a given domain name. This applies for every certificate operation (registration, update, and revocation), meaning that even frequent certificate updates (such as in the case of short-lived certificates) are subject to successful log synchronisation. In addition, AKI requires that each domain name only has one active and valid certificate associate with it at any given time. Moreover, AKI needs to rely on third parties called validators to ensure that the log is maintained without improper modifications, and to assume that CAs, public log maintainers, and validators do not collude together.

Certificate issuance and revocation transparency Certificate issuance and revocation transparency (CIRT) [Rya14] improves certificate transparency by providing transparent key revocation, and reducing reliance on trusted parties.

To provide an effective way for certificate revocation, CIRT proposes a new log structure that consists of two tree structures presenting the same set of data. The first tree is called a ChronTree, which is an append-only Merkle tree (as in CT) ordered chronologically. The second tree is called LexTree, which is a Merkle tree ordered lexicographically by the subject of the certificate. The ChronTree stores in the leaves a pair (C, h), where C is a certificate appended in the ChronTree, and h is the hash root value of the LexTree in which the last inserted data is C. The LexTree stores $h(d_i)$ in every node for some i, where d is an ordered list of certificates that has the same subject. The last element in the list is the current valid certificate of the subject.

This log structure enables the log maintainer to provide efficient proofs that (A) some data is present in the log, (B) any data having a given attribute (e.g. an identity) is absent from the log, (C) some data is the latest valid one in the log, and (D) the current log is extended from a previous version.

Loosely speaking, by proving a proof that a certificate C is the last element in an ordered list d, and $h(d)$ is present in the LexTree of the log, a verifier is ensured that C is the currently valid certificate, i.e. not revoked. Due to the use of two different trees presenting the same set of data, it is crucial to ensure that the data presented by the two trees are consistent. To verify the consistency of the two trees, CIRT distributes the monitoring role among user browsers. To do so, each user browser verifies if a randomly selected certificate stored in the ChronTree is also in the LexTree. If the number of such random verification is big enough, then the consistency between the two trees is likely to be verified.

- **Strength**
  CIRT provides a solution for managing both certificate issuance and revocation by using a new log structure, and reduces reliance on trusted parties by using user side random verifications. It also allows a domain to have multiple certificates, and to update keys timely. In addition, similar to all other systems in certificate management transparency category, it does not need users to be involved.

- **Weakness**
  A weakness of CIRT is that it can only detect attacks that use fake certificates; it cannot prevent them. Also, since CIRT was proposed for email applications, it does not support the multiplicity of log maintainers that would be required for web certificates.

Attack Resilient Public-Key Infrastructure Attack Resilient Public-Key Infrastructure (ARPKI) [BCK+14] is an improvement on AKI. In ARPKI, a client can designate n service providers (e.g. CAs and log maintainers), and only needs to contact one CA to register her certificate. Each of the designated service providers will monitor the behaviour of other designated service providers. As a result, ARPKI prevents attacks even when $n-1$ service providers are colluding together, whereas in AKI, an adversary who successfully compromises two out of three designated service providers can successfully launch attacks [BCK+14].

- **Strength**

ARPKI is the first formally verified log-based PKI system. Its security properties are proved by using a protocol verification tool called TAMARIN prover [MSCB13]. The verification uses several abstractions during modelling. For example, they represent its underlying log structure (a Merkle tree) as a list.

- **Weakness**
  The weakness of ARPKI is that all n designated service providers have to be involved in all the processes (i.e. certificate registration, confirmation, and update), which would cause considerable extra latencies and the delay of client connections.

3.4 Observations

Based on the above analysis, we observed the advantage and weakness in each category. This section discusses the observations based on different perspectives, i.e. on the property perspective and on the system perspective. In addition, this section summarises our observation regarding to the leading proposals in Table 3.2.

3.4.1 Property Perspective

We summarise our observations on different system categories according to the perspective of identified properties.

Trust agility
The current CA model does not provide this feature, since any compromised CA can issue valid certificates for any domain server. Similarly, systems in the category of *Scope restriction* also do not provide this feature, because they merely restrict the set of domains that CAs can issue certificates for. Most systems in *difference observation* offer this feature, as any one can be a notary server, and users can select which notary servers they want to trust, and any notary server will not be influenced by other notary servers.

Anti-oligopoly
Systems in the category of *difference observation* normally provide this feature, as the number of observers are not fixed. In addition, the certificate verification

Table 3.2: Evaluation of proposals

Desired Features	Classic	Difference observation			Scope restriction		Cert Mngmt Trans	
	CA-based	Perspectives	Convergence	TACK	PKP	DANE	CT	ARPKI
Trust								
Trust agility	×	√	√	√[1]	×	×	√	√
Free of trusted parties	×	×	×	√[1]	×	×	×	×
Verifiable of trusted parties	×	×	×	×	×	×	√	√
Anti-oligopoly	×	√	√	√	×	×	×	×
Availability								
Offline verification	√	×	⊗[2]	√	√	√	√	√
Built-in key revocation	×	×	×	×	×	×	×	√
Scalability	√	√	√	×[1]	√	√	√	√
Multiple certificate support	√	×	×	√	√	√	√	×
Timely key verification	√	×	√	×	√	√	√	√
Security								
First connection protection	√	√	√	×	√	√	√	√
DOS attack protection	√	×	⊗[3]	√[1]	√	√	√	√
Use of mis-issued certificate prevention	×	√	√	√	⊗[4]	⊗[4]	×	√
Use of mis-issued certificate detection	×	√	√	√	⊗[4]	⊗[4]	√	√
Provably secure	×	×	×	×	×	×	×	√
Usability								
No user involvement	√	×	√	×	√	√	√	√
Privacy								
Protecting browsing history	√	×	√	√[1]	√	√	√	√

√ – The subject offers this feature.
⊗ – The subject offers this feature but with other concerns.
× – The subject does not offer this feature.
— – Not applicable.

[1] We consider the case without using an on-line crowed-sourced pin store. If an online pin store is used, then the result might be different depending on how the store is designed. (The pin store has not been proposed yet.)

[2] This feature is satisfied if and only if the received public key/certificate can be found in the local cache.

[3] This feature is satisfied if and only if the received public key certificate can be found in the local cache, and the subject of the certificate has not updated its certificate.

[4] This feature is satisfied if the malicious CA is not authorised for the victim domain.

result relies on the out-of-band checking through a different path. So, as far as the observed certificates are the same one, the client will accept it. Thus, the role of CA is minimised.

Offline verification
In the current CA model, clients only need to verify the validity of the received certificate. So, it satisfies offline verification. Systems in the category of *scope restriction* also provide this feature, as the way they work is similar to the current CA model, but with some restrictions.

Most systems in the category of *certificate management transparency* offer this feature as well, because in these systems the proofs to be verified about a certificate are provided together with the certificate. In contrast, most systems in the difference observation category don't offer this feature, because with these systems, clients have to make additional connections to verify the certificates they obtained.

Built-in key revocation
Most systems in the category of *difference observation* and *scope restriction* do not provide this feature. Most systems in the category of *certificate management transparency* do offer this feature. For example, CIRT proposed a way to manage certificate revocation by using an advanced log structure; and AKI and ARPKI manage certificate revocation by only recording the latest certificates of domains in their logs.

Multiple certificate support
The current CA model offers this feature. Systems in the category of *difference observation* generally don't provide this feature. Because when clients see different certificates of the same website from different paths or observers, a warning will be displayed to clients even if the received certificates are all genuine. Systems in the category of *scope restriction* and *certificate management transparency* provide this feature.

Timely key verification
Systems in *difference observation* are likely to not provide this feature, as the observers might not be always up to date with all domains.

First connection protection
Systems such as Certificate Patrol, Certlock, and TACK in *difference observation*

do not provide this feature, because they verify the certificate based on what has been observed in the previous connections.

Denial of service (DoS) attack protection
The CA model offers this feature. All systems in the category of *scope restriction* and most systems in the category of *certificate management transparency* provide this feature as well. However, some systems in *difference observation* require out-of-band observation, so they will not provide this feature, as the verification server can be blocked.

Use of mis-issued certificate prevention
All systems in *difference observation*, and some systems in the category of *certificate management transparency*, provide this feature. In contrast, systems in the category of *scope restriction* do not provide this feature if the mis-issued certificate is issued by a CA who is authorised for the victim domain. For example, DANE cannot prevent MITM attacks when the fake certificate used by an attacker is issued by the parent domain of the victim domain.

Use of mis-issued certificate detection
All systems in *difference observation* and in *certificate management transparency* provide this feature.

3.4.2 System Perspective

As shown in the table, systems in the category of *difference observation* provide better trust-related features. However, they can have difficulties to provide a better availability, because the observer might not have the latest update, the systems in general do not provide an effective key revocation management, and they require user involvement to make decisions. Moreover, they can suffer from DoS attacks on the observers.

Systems in the category of *scope restriction* provide better usability and availability. However, they have only restricted the power of each trusted parties, but internet users still need to trust them. This can limit the damage from attacks launched by malicious CAs, but cannot completely solve the problem.

Systems in the category of *certificate management transparency* provide better security and availability. However, anti-monopoly might be a problem for these systems. It is desired to provide a fully distributed system and still be able to remove the need of trusted parties.

Summary As discussed above, different systems in different categories have different pros and cons. Systems in the category of certificate management transparency is a new research trend, and they seem to provide more desired features than systems in other categories.

3.5 Conclusion

The current certificate authority trust model is broken. As a result, the communications between internet users and web servers might be vulnerable to man-in-the-middle attacks. Interested malicious parties could intercept user communication and steal user credential data.

Many security-enhanced alternatives have been proposed. Our evaluation shows that four main concepts are used to design web certificate management systems. They have different advantages and shortcomings, and as yet there is no clear winner. We hope our evaluation framework would help the ongoing research on web certificate management alternatives.

CHAPTER 4

DISTRIBUTED TRANSPARENT KEY INFRASTRUCTURE

4.1 Introduction

This chapter presents a new log-based architecture for managing web certificates, called *Distributed Transparent Key Infrastructure* (DTKI). DTKI minimises the presence of oligopoly, prevents attacks that use fake certificates, provides a way to manage certificate revocation, and is secure even if all service providers (e.g. CAs and log maintainers) collude together. We also provide formal machine-checked verification of core security properties of DTKI, by using the Tamarin prover.

The rest of this chapter is organised as follows. We first provide an overview of DTKI in §4.2, then detail the public log structure in §4.3, and our main protocol in §4.4. Our security verification and performance evaluation are presented in §4.5 and §4.6, respectively. The comparison with other systems in the category of certificate management transparency is given in §4.7, and more discussions are presented in §4.8.

4.2 Overview

Distributed Transparent Key Infrastructure (DTKI) is an infrastructure for managing keys and certificates on the web in a way that is *transparent*, minimises *oligopoly*, and allows verification of the behaviour of trusted parties. In DTKI, we mainly have the following agents:

Certificate log maintainer (CLM): A certificate log maintainer maintains a database of all valid and invalid (e.g. expired or revoked) certificates for a particular set of domains for which it is responsible. It commits to the digest of its log, and is able to provide efficiently verifiable proofs of presence or absence of a certificate in its log. Certificate log maintainers behave transparently and their actions can be verified. To minimise oligopoly, DTKI does not fix the set of certificate logs.

Mapping log maintainer (MLM): The mapping log maintainer maintains association between certificate logs and the domains they are responsible for. It also commits to digests of the log, and provides the proof of presence of a current association, and behaves transparently. Clients of the mapping log are not required to blindly trust the maintainer, because they can efficiently verify the obtained proofs with respect to the obtained associations.

Mirror: Mirrors are servers that maintain a full copy of the mapping log and certificate logs. In other words, mirrors maintain distributed copies of logs. Anyone (e.g. ISPs, CLMs, CAs, domain owners) can be a mirror. Unlike in SK, mirrors are not required to be trusted in DTKI, because all information provided by a mirror must accompany by proofs. The proofs are efficiently verifiable, and are associated to the digest committed by log maintainers.

Certificate authority (CA): They check the identity of domain owners, and create certificates for them. However, in contrast with today's CAs, the ability of CAs in DTKI is limited since the issuance of a certificate from a CA is not enough to convince web browsers to accept the certificate (proof of presence of the certificate in the relevant certificate log is also needed).

In DTKI, each domain owner has two types of certificate, namely TLS certificate and master certificate. Domain owners can have multiple TLS certificates but can have only one master certificate. A TLS certificate contains the public key of a domain server for a TLS connection, whereas the master certificate contains a public key, called "master verification key". The corresponding secret key of the master certificate is called "master signing key". Similar to the "sovereign key" in SK [Eck12], the master signing key is only used to validate a TLS certificate (of the same subject) by issuing a signature on it. This limits the ability of certificate authorities since without having a valid signature (issued by using the master signing key), the TLS certificate will not be accepted. Hence, the TLS secret key is the one for daily use; and the master signing key is rarely used. (The master signing key will only be used for validating a new certificate, or revoking an existing certificate.)

After a domain owner obtains a master certificate or a TLS certificate from a CA, he needs to make a registration request to the corresponding CLM to publish the certificate into the log. To do so, the domain owner signs the certificate using the master signing key, and submits the signed certificate to a CLM determined (typically based on the top-level domain) by the mapping log maintainer. The certificate log maintainer checks the signature, and accepts the certificate by adding it to the certificate log if the signature is valid. The process of revoking a certificate is handled similarly to the process of registering a certificate in the log.

When establishing a secure connection with a domain server, the browser receives a certificate from the domain server and proofs from a mirror. The expected proofs include a proof that the certificate is not revoked, a proof that the certificate is recorded in the certificate log, and a proof that this certificate log is authorised to manage certificates for the domain. Users and their browsers should only accept a certificate if the certificate is issued by a CA, and validated by the domain owner, and all proofs are successfully verified.

Fake master certificates or TLS certificates can be easily detected by the domain owner, because the attacker will have had to insert such fake certificates into the log (in order to be accepted by browsers), and is thus visible to the domain owner.

In order to fool a victim, an attacker will have to insert fake certificates into the log. So domain owners can easily detect such attacks by checking the log.

Rather than relying solely on trusted monitors to verify the healthiness of logs and the relations between logs, DTKI uses a crowdsourcing-like way to ensure the integrity of the log, and to ensure the relations between the mapping log and certificate logs, and between certificate logs. In particular, the monitoring work in DTKI can be broken into independent little pieces, and thus can be done by distributing the pieces to users' browsers. In this way, users' browsers can perform randomly-chosen pieces of the monitoring role in the background (e.g. once a day). Thus, web users can collectively monitor the integrity of the logs. We envisage parameters in browsers allowing users to control how that works.

Table 4.1: The methods supported by the log.

Method	Input	Output
Size	T	The size of the Merkle tree T.
Root	T	The root value of the Merkle tree T.
Last	T	The data stored in the rightmost side leaf of the Merkle tree T. This proof can only work if T is a ChronTree.
PoP	(T, d)	*Proof of Presence*: The proof that d is in T.
PoC	(T, d)	*Proof of Currency*: The proof that d is the latest added data. This proof can only work if T is a ChronTree.
PoE	(T, dg')	*Proof of Extension*: The proof that the Merkle tree T is an extension of another Merkle tree whose digest is dg'. This proof can only work if both trees are ChronTrees.
PoA	(T, a)	*Proof of Absence*: The proof that any data d having attribute a is absent from the Merkle tree T. This proof can only work if T is a LexTree.

4.3 The public log

DTKI uses append-only logs to record all requests processed by the log maintainer, and allows log maintainers to generate proofs that can be efficiently verified. These proofs mainly include that some data (e.g. a certificate or a revocation request) has or has not been added to the log; and that a log is extended from a previous version. So, the log maintainer's behaviour is transparent to the public. In DTKI, to provide all needed proofs, the public log is constructed by using two types of data structures, namely chronological data structure and ordered data structure.

We first present the two types of data structures encapsulating the desired properties, together with concrete implementations using Merkle tree [Mer87]. Then, we explain how to use the data structure to construct our public logs.

4.3.1 Data structures

We use the notion of *digest* to uniquely characterise a set of data, such that the size of a digest is a constant. In our implementation (that will be presented later), the digest of a set of data is the root hash of a Merkle tree storing the set of data. Table 4.1 summarises the methods that our log supports, according to our detailed implementation.

Abstract data structures

A *chronological data structure* is an append-only data structure, i.e. the only allowed change operation is adding some data. Let S be a sequence of data, and N and dg be the size and digest of S, respectively. With a chronological data structure, we have that $d \in S$ for some data d, if and only if there exists a proof p of size $O(\log(N))$, called the proof of presence of d in S, such that p can be efficiently verified; and for all sequence S' with digest dg' and size $N' < N$, we are able to show that S' is a prefix of S, if and only if there exists a proof p' of size $O(\log(N))$, called the proof of extension of S from S', such that p' can be efficiently verified.

The chronological data structure enables one to prove that some data (e.g. a certificate) is in a set of data (e.g. the log), and a version of the set of data is an extension of a previous version. The size of these proofs are small, i.e. $O(log(N))$, and the proofs can be efficiently verified. This is useful for our public log since it enables users to verify the history of a log maintainer's behaviours.

Unfortunately, the chronological data structure does not provide all desired features needed for our log. For example, it is very inefficient to verify that some data (e.g. a certificate revocation request) is *not* in the chronological data structure (the cost is $O(N)$). To provide missing features, we need to use the *ordered data structure.*

An *ordered data structure* is a data structure allowing one to insert, delete, and modify stored data. In addition, with an ordered data structure, we have $d \in S$ (resp. $d \notin S$) for some data d, if and only if there exists a proof p of size $O(\log(N))$, called the proof of presence (resp. absence) of d in (resp. not in) S, such that p can be efficiently verified.

With an ordered data structure, however, the size of proof that the current version of the data is extended from a previous version is $O(N)$. As the chronological data structure and the ordered data structure have complementary properties, we will use the combination of them to organise our log. We first define the concrete data structures, then introduce how to construct the two data structures in detail, and finally show how to use them to construct our public logs.

Chronological data structure implementation

Possible implementations of chronological data structure include append-only Merkle tree [Mer87] and append-only skip list, as proposed in [LLK13] and [MB03], respectively. Our implementation of the chronological data structure is called *ChronTree*, which is based on the Merkle tree structure. We consider a secure hash function (e.g. SHA256), denoted h.

A *ChronTree* T is a binary tree whose nodes are labelled by bitstrings such that:

- every leaf node is labeled by a data item being stored in the ChronTree; and
- every non-leaf node in T has two children, and is labelled with $\mathsf{h}(t_\ell, t_r)$ where t_ℓ (resp. t_r) is the label of its left child (resp. right child); and
- the subtree rooted by the left child of a node is perfect, and its height is greater than or equal to the height of the subtree rooted by the right child.

Here, a subtree is "perfect" if every non-leaf node of the subtree has two children and all leaves of the subtree have the same depth.

Note that a ChronTree is a not necessarily a balanced tree. The two trees in Figure 4.1 are examples of ChronTrees where the data stored are the bitstrings denoted $d_1, \dots, d_6$.

$$\begin{tikzcd}
 & & \mathsf{h}(\mathsf{h}(d_1,d_2),d_3) \arrow[dl] \arrow[dr, "r"] & \\
 & \mathsf{h}(d_1,d_2) \arrow[dl] \arrow[dr] & & d_3 \\
d_1 & & d_2 &
\end{tikzcd}$$

(a) ChronTree T_a

$$\begin{tikzcd}
 & & & & & \mathsf{h}(\mathsf{h}(\mathsf{h}(d_1,d_2),\mathsf{h}(d_3,d_4)),\mathsf{h}(d_5,d_6)) \arrow[dlll, "\ell"'] \arrow[drr] & & & \\
 & & \mathsf{h}(\mathsf{h}(d_1,d_2),\mathsf{h}(d_3,d_4)) \arrow[dl] \arrow[drr, "r"] & & & & & \mathsf{h}(d_5,d_6) \arrow[dl] \arrow[dr] & \\
 & \mathsf{h}(d_1,d_2) \arrow[dl] \arrow[dr] & & & \mathsf{h}(d_3,d_4) \arrow[dl, "\ell"'] \arrow[dr] & & d_5 & & d_6 \\
d_1 & & d_2 & d_3 & & d_4 & & &
\end{tikzcd}$$

(b) ChronTree T_b

Figure 4.1: Example of two ChronTrees, T_a and T_b.

Given a ChronTree T with N leaves, we use $\mathcal{S}(T) = [d_1, \dots, d_N]$ to denote the sequence of bitstrings stored in T. Note that a ChronTree is completely defined by the sequence of data stored in the leaves. Moreover, we say that the size of a ChronTree is the number its leaves.

Intuitively, a proof of presence of d in T contains the minimum amount of information necessary to recompute the label of the root of T from the leaf containing d. The algorithm that outputs the unique proof is defined as follows.

Given a bitstring d and a ChronTree T, the *proof of presence of* d *in* T exists if there is a leaf n_1 in T labelled by d; and is defined as $(w, [b_1, \ldots, b_k])$ such that:

- w is the position in $\{\ell, r\}^*$ of n_1 (that is, the sequence of left or right choices which lead from the root to n_1), and $|w| = k$; and
- if $n_1, \ldots, n_{k+1}$ is the path from n_1 to the root, then for all $i \in \{1, \ldots, k\}$, b_i is the label of the sibling node of n_i.

Given d, (w, seq) and (h, N), to verify that (w, seq) proves the presence of d in the ChronTree T such that $\mathsf{Root}(T) = h$ and $\mathsf{Size}(T) = N$, a verifier should check $|w| = |seq|$, and verify if the final output of following algorithm $f(w, seq, d)$ is equal to h. For some value t, the algorithm is defined as follows:

- $f(\mathsf{null}, [], t) = t$
- $f(w \cdot \ell, [b_1, \ldots, b_k], t) = f(w, [b_2, \ldots, b_k], \mathsf{h}(t, b_1))$
- $f(w \cdot r, [b_1, \ldots, b_k], t) = f(w, [b_2, \ldots, b_k], \mathsf{h}(b_1, t))$

Example 4.1. *Consider the ChronTree T_b of Figure 4.1. The proof of presence of d_3 in T_b is the tuple (w, seq) where:*

- $w = \ell \cdot r \cdot \ell$
- $seq = [d_4, \mathsf{h}(d_1, d_2), \mathsf{h}(d_5, d_6)]$

To verify the above proof, one needs to check if the reconstructed hash value is equal to the root hash of T_b, i.e. whether $\mathsf{h}(\mathsf{h}(\mathsf{h}(d_1, d_2), \mathsf{h}(d_3, d_4)), \mathsf{h}(d_5, d_6)) = \mathsf{Root}(T_b)$ or not.

The *proof of currency* is the same as the proof of presence, but there is an extra constraint for the verifier to check, namely that the path w to the leaf (e.g., the path from the root to d_6 in T_b of Figure 4.1) is of the form $r \cdot r \ldots \cdot r$.

Let T and T' be ChronTrees of size N and N', digest h and h', respectively, such that $N' \leq N$, $\mathcal{S}(T) = [d_1, \ldots, d_{N'}, \ldots, d_N]$, and $\mathcal{S}(T') = [d_1, \ldots, d_{N'}]$ for some bitstrings d_1, …, $d_{N'}$, …, d_N. In the binary representation of a number, we consider that the rightmost bit is at position 0. For example in 01001100, the smallest position of the bit 1 is 2.

The *proof of extension* of T' into T is the list of nodes in the ChronTree required to verify that the first N' data items $\mathcal{S}(T')$ are equal in both trees. We define an algorithm that outputs the unique proof.

Let m be the smallest position of the bit "1" in the binary representation of N'; and let (d, w) be the $(m+1)$-th node in the path of the node labelled by $d_{N'}$ to the root in T, where d is a bitstring and $w \in \{\ell, r\}^*$ indicates the position of this node. Finally, let (w, seq') be the proof of presence of d in T. The proof of extension of T' into T is defined as the sequence seq of bitstrings such that

- if $N' = 2^k$ for some k, then $seq = seq'$; otherwise
- $seq = d||seq'$, where $||$ is the concatenation operation.

Example 4.2. *The proof of extension of T_a into T_b (Figure 4.1) is the sequence $seq = [d_3, d_4, \mathsf{h}(d_1, d_2), \mathsf{h}(d_5, d_6)]$. In more detail, since the size N' of T_a is 3 and the binary representation of which is 11, we have that $m = 0$. In addition, since the size does not satisfy $N' = 2^{k'}$ for some $k' \in \mathbb{N}$, we have that $seq = d_3||seq'$, where d_3 is the $(m+1)$-th node (i.e. the first node) in the path of the node labelled by $d_{N'}$ to the root in T_b, and $seq' = [d_4, \mathsf{h}(d_1, d_2), \mathsf{h}(d_5, d_6)]$ is the sequence in the proof (w, seq') of presence of d_3 in T_b. Thus, the proof of extension of T_a into T_b is $[d_3, d_4, \mathsf{h}(d_1, d_2), \mathsf{h}(d_5, d_6)]$.*

Let $w_1 \cdot w_2$ and $w'_1 \cdot w_2$ be the position of the node labelled by $d_{N'}$ in T and T', respectively. Given (N, h), (N', h'), and $seq = [d_1, \ldots, d_k]$, the verification of the above proof seq of extension is defined as follows:

- if $N' = 2^{k'}$ for some $k' \in \mathbb{N}$, then verify that (w_1, seq) is the proof of presence of a node labelled by h' in T;
- otherwise, we have $|w_1| = k - 1$, and if we denote $w_1 = a_k \cdot \ldots \cdot a_2$ and if $S = [i_1, \ldots, i_p]$ is the increasing sequence of integer such that $\forall j \in S$, $a_j = r$;

and $\forall j \in \{2, \dots, k\} \setminus S$, $a_j = \ell$ then we have that $(w_1, [b_2, \dots, b_k])$ is the proof of presence of b_1 in T, and $(w'_1, [b_{i_1}, \dots, b_{i_p}])$ is the proof of presence of b_1 in T'.

Example 4.3. *Figure 4.1 is the graphical representation of the verification of the proof of extension of T_a into T_b. Given size and root of T_a and T_b, one can derive the shape of the trees (so the position of the node labelled by $d_{N'}$ in T and T'), and verify the proof (generated in the previous example). In more detail, since N', which is the size of T_a, does not satisfy $N' = 2^{k'}$ for some $k' \in \mathbb{N}$, the verifier knows that $w_1 = k - 1$, where $k = |seq| = 4$. So, we have $w_1 = \ell \cdot r \cdot \ell = a_4 \cdot a_3 \cdot a_2$, $w'_1 = r$, $w_2 = \mathsf{null}$, and $S = [b_3]$, where b_3 is the third elements, i.e. $\mathsf{h}(d_1, d_2)$, in seq. Thus, the final check the verifier needs to perform is that $(\ell \cdot r \cdot \ell, [d_4, \mathsf{h}(d_1, d_2), \mathsf{h}(d_5, d_6)])$ proves the presence of d_3 in T_b, and $(r, [\mathsf{h}(d_1, d_2)])$ proves the presence of d_3 in T_a.*

Ordered data structure implementation

Possible implementations of the ordered data structure include lexicographically ordered Merkle tree ([Rya14]), and authenticated dictionaries [AGT01]. Our implementation of the ordered data structure is called *LexTree*, which is the combination of a binary search tree and a Merkle tree. The idea is that we can regroup all the information about a subject into a single node of the binary search tree, and while being able to efficiently generate and verify the proof of presence. We consider a total order on bitstrings denoted $\leq$. This order could be the lexicographic order in the ASCII representations but it could be any other total order on bitstrings.

A *LexTree* T is a binary search tree over pairs of bitstrings

- for all two pairs (d, h) and (d', h') of bitstrings in T, (d, h) occurs in a node left of the occurrence of (d', h') if and only if $d \leq d'$ lexicographically;
- for all nodes $n \in T$, n is labelled with the pair $(d, \mathsf{h}(d, h_\ell, h_r))$ where d is some bistring and (d_ℓ, h_ℓ) (resp. (d_r, h_r)) is the label of its left child (resp. right child) if it exists; or the constant null otherwise.

Note that contrary to a ChronTree, the same set of data can be represented by different LexTrees depending on how the tree is balanced. To avoid this situation, we assume that there is a pre-agreed way for balancing trees.

$$\begin{tikzcd}[column sep=tiny]
 & & & & & d_6,\ \mathsf{h}(d_6, \mathsf{h}(d_4, \mathsf{h}(d_2, \mathsf{h}_1, \mathsf{h}_3), \mathsf{h}_5), \mathsf{h}(d_{10}, \mathsf{h}(d_8, \mathsf{h}_7, \mathsf{h}_9), \mathsf{h}(d_{12}, \mathsf{h}_{11}, \mathsf{null}))) \arrow[dlll] \arrow[drrr] & & & & & \\
 & & d_4,\ \mathsf{h}(d_4, \mathsf{h}(d_2, \mathsf{h}_1, \mathsf{h}_3), \mathsf{h}_5) \arrow[dl] \arrow[dr] & & & & & & d_{10},\ \mathsf{h}(d_{10}, \mathsf{h}(d_8, \mathsf{h}_7, \mathsf{h}_9), \mathsf{h}(d_{12}, \mathsf{h}_{11}, \mathsf{null})) \arrow[dll] \arrow[drr] & & \\
 & d_2, \mathsf{h}(d_2, \mathsf{h}_1, \mathsf{h}_3) \arrow[dl] \arrow[dr] & & d_5, \mathsf{h}_5 & & & d_8,\ \mathsf{h}(d_8, \mathsf{h}_7, \mathsf{h}_9) \arrow[dl] \arrow[dr] & & & & d_{12},\ \mathsf{h}(d_{12}, \mathsf{h}_{11}, \mathsf{null}) \arrow[dl] \\
d_1, \mathsf{h}_1 & & d_3, \mathsf{h}_3 & & & d_7,\ \mathsf{h}_7 & & d_9,\ \mathsf{h}_9 & & d_{11}, \mathsf{h}_{11} &
\end{tikzcd}$$

Figure 4.2: An example of a LexTree T_c, where $\mathsf{h}_i = \mathsf{h}(d_i, \mathsf{null}, \mathsf{null})$ for all $i = \{1, 3, 5, 7, 9, 11\}$

Example 4.4. *The tree in Figure 4.2 is an example of LexTree where* $d_1 \leq d_2 \leq \ldots \leq d_{12}$.

Similar to ChronTree, the verification of the proof of presence of some data d in a LexTree T is to reconstruct the hash value of the root of T.

Example 4.5. *Consider the LexTree T of Figure 4.2. The proof of presence of d_8 in T is the tuple $(h_\ell, h_r, seq_d, seq_h)$ where:*

- $h_\ell = \mathsf{h}_7$ *and* $h_r = \mathsf{h}_9$*; and*
- $seq_d = [d_{10}, d_6]$
- $seq_h = [\mathsf{h}(d_{12}, \mathsf{h}_{11}, \mathsf{null}),\ \mathsf{h}(d_4, \mathsf{h}(d_2, \mathsf{h}_1, \mathsf{h}_3), \mathsf{h}_5)]$

The proof of absence of some data d in a LexTree can be done by showing that two data items d_i and d_j for some i and j are adjacent in the left-right traversal of LexTree, while lexicographically we have $d_j < d < d_j$; this is also $O(\log(N))$.

Example 4.6. *Consider the T_c of Figure 4.2, and some data d such that $d_7 \leq d \leq d_8$. The proof of absence of d in T_c is the tuple $(\mathsf{null}, \mathsf{null}, seq_d, seq_h)$ where:*

- $seq_d = [d_7, d_8, d_{10}, d_6]$
- $seq_h = [\mathsf{h}_9,\ \mathsf{h}(d_{12}, \mathsf{h}_{11}, \mathsf{null}),\ \mathsf{h}(d_4, \mathsf{h}(d_2, \mathsf{h}_1, \mathsf{h}_3), \mathsf{h}_5)]$

To verify this proof, one needs to verify that both d_7 and d_8 are present in T_c, and d_8 is the smallest one larger than d_7 lexicographically according to their positions in T_c, as shown in Figure 4.2.

4.3.2 The mapping log

To minimise oligopoly, DTKI uses multiple certificate logs, and does not fix the set of certificate logs and the mapping between domains and certificate logs. The mapping log records associations between domain names and certificate log maintainers, and its maintainer can provide efficiently verifiable proofs regarding the current association. Due to the large number of domains, it would be rather inefficient to explicitly associate each domain name to a certificate log. To address this problem, we use a class of simple regular expressions to present a group of domain names, and record the associations between regular expressions and certificate logs in the mapping log. For example, (*\.org, Clog$_1$) and ([a-h].*\.com, Clog$_1$) mean that certificate log maintainer Clog$_1$ is authorised for all domains end with *.org* and all domains start with letters from a to h end with *.com*.

Intuitively, the mapping log is organised by using a chronological data structure, and stores received requests together with the request time, and four digests of different ordered data structures representing the status of the log. Each record is of the form

$$\mathsf{h}(req, t, dg^s, dg^{bl}, dg^r, dg^i)$$

In this formula, as presented in Figure 4.3, req is the request received by the mapping log at time t; dg^{s}[1] stores information about certificate log maintainers (e.g. the certificate of the certificate log maintainer, and the current digest of the certificate log $clog$); dg^{bl} stores the identity of blacklisted certificate log maintainers; dg^r stores the mapping from a regular expression to the identity of certificate log maintainers, and dg^i stores the mapping from the identity of certificate log maintainers to a set of regular expressions.

In more detail, each record of the mapping log contains digests after processing the request req (received by the mapping log maintainer at time t) on the digest stored in the previous record. Each of the notations is explained as follows:

- req includes $\mathsf{add}(rgx, id)$, $\mathsf{del}(rgx, id)$, $\mathsf{new}(cert)$, $\mathsf{mod}(cert, \mathsf{sign}\{cert'\}_{sk}, \mathsf{sign}\{n, dg, t\}_{sk'})$, $\mathsf{bl}(id)$, and end, respectively corresponding to a request to

[1]We simplified the description here: we should say the ordered data structure represented by dg^s stores the information, rather than the digest dg^s stores it. We will use this simplification through the thesis.

add a mapping (rgx, id) of regular expression rgx and identity id of a $clog$, to delete a mapping (rgx, id), to add a certificate $cert$ of a new $clog$, to change the certificate of a $clog$ from $cert$ to $cert'$, to blacklist id of an existing $clog$, and to close the update request; where sk and sk' are signing keys associated to the certificate $cert$ and $cert'$, respectively; $cert$ and $cert'$ share the same subject, and n and dg are the size and the digest of the corresponding $clog$ at time t, respectively;

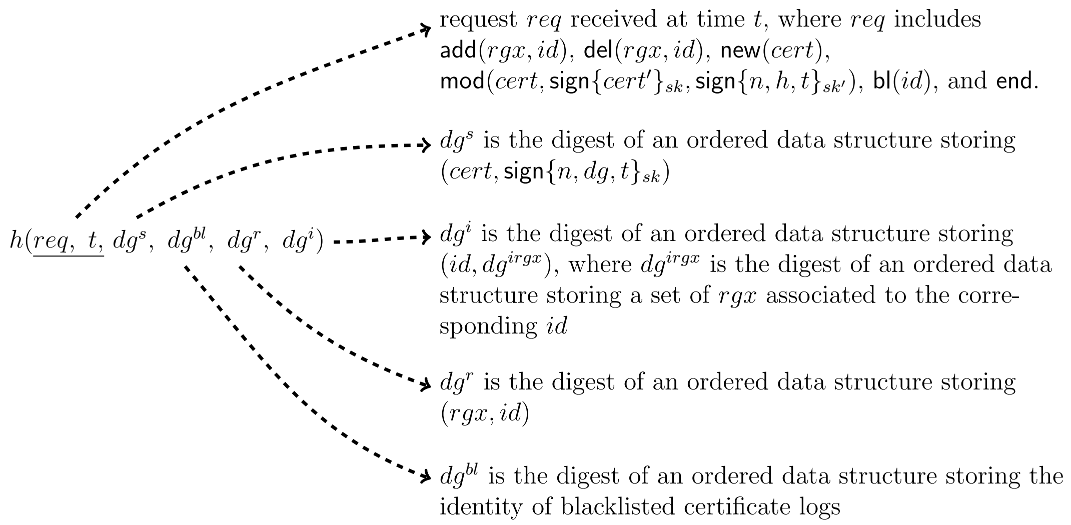
- dg^s is the digest of an ordered data structure storing the identity information of the form $(cert, \mathsf{sign}\{n, dg, t\}_{sk}$ for the currently active certificate logs, where $cert$ is the certificate for the signing key sk of the certificate log, and n and dg are respectively the size and digest of the certificate log at time t. Data are ordered by the domain name in $cert$.

- dg^{bl} is the digest of an ordered data structure storing the domain names of blacklisted certificate logs. Data are ordered by the domain names.

- dg^r is the digest of an ordered data structure storing elements of the form (rgx, id), which represents the mapping from regular expression rgx to the identity id of a $clog$, data are ordered by rgx;

- dg^i is the digest of an ordered data structure storing elements of the form (id, dg^{irgx}), which represents the mapping from identity id of a $clog$ to a digest dg^{irgx} of ordered data structure storing a set of regular expressions, data are ordered by id.

The requests are used for modifying mappings or the existing set of certificate log maintainers. For example, when a request $\mathsf{del}(rgx, id)$ (or $\mathsf{add}(rgx, id)$) has been processed, the mapping between the certificate log with identity id and regular expression rgx is revoked (resp. created). After appending all needed update requests in an update, the end will be appended in the mapping log to indicate that the current update process is done.



request req received at time t, where req includes $\mathsf{add}(rgx, id)$, $\mathsf{del}(rgx, id)$, $\mathsf{new}(cert)$, $\mathsf{mod}(cert, \mathsf{sign}\{cert'\}_{sk}, \mathsf{sign}\{n, h, t\}_{sk'})$, $\mathsf{bl}(id)$, and end.

dg^s is the digest of an ordered data structure storing $(cert, \mathsf{sign}\{n, dg, t\}_{sk})$

dg^i is the digest of an ordered data structure storing (id, dg^{irgx}), where dg^{irgx} is the digest of an ordered data structure storing a set of rgx associated to the corresponding id

dg^r is the digest of an ordered data structure storing (rgx, id)

dg^{bl} is the digest of an ordered data structure storing the identity of blacklisted certificate logs

Figure 4.3: A figure representation of the format of each record in the mapping log.

4.3.3 Certificate logs

A certificate log mainly stores certificates for domains according to the mappings presented in the mapping log. In particular, a certificate log is organised by using a chronological data structure, and each record of the log is of the form

$$\mathsf{h}(req, N, dg^{rgx})$$

where req is the received request and is processed at the time such that the mapping log is of size N; dg^{rgx} represents an ordered data structure storing a set of mappings from regular expressions to the information associated to the corresponding domains, such that the domain name is an instance of the regular expression. The stored information of a domain includes the identity and the master certificate of the domain, and two digests dg^a and dg^{rv} each presents an ordered data structure storing a set of active TLS certificates and a set of expired or revoked TLS certificates, respectively.

Elements in a record (as shown in 4.4) of a certificate log are detailed as follows.

- req includes $\mathsf{reg}(\mathsf{sign}\{cert, t, \text{‘reg’}\}_{sk})$, $\mathsf{rev}(\mathsf{sign}\{cert, t, \text{‘rev’}\}_{sk}$, $\mathsf{upadd}(\mathsf{h}(id), h)$, and $\mathsf{updel}(\mathsf{h}(id), h)$. In which, the request $\mathsf{reg}(\mathsf{sign}\{cert, t, \text{‘reg’}\}_{sk})$ (or $\mathsf{rev}(\mathsf{sign}\{cert, t, \text{‘rev’}\}_{sk})$ is a request to register (resp. revoke) a certificate $cert$ signed by using the master key sk, at an agreed time t, where ‘reg’ and 'rev' are constant. $\mathsf{upadd}(\mathsf{h}(id), h)$ (or $\mathsf{updel}(\mathsf{h}(id), h)$) is the request to update the certificate log by adding (resp. deleting) certificates of identity id according to the changes of $mlog$, where h is the digest presenting the status of id at time t. The status of id includes its master certificate, and the set of its active certificates and the set of its revoked certificates.

- N is the size of $mlog$ at the time req is processed;
- dg^{rgx} is the digest of an ordered data structure storing a set of elements of the form (rgx, dg^{id}), represents the status of the certificate log after processing the request req, and stores all the regular expressions rgx that the certificate log is associated to. dg^{id} is the digest of an ordered data structure storing a set of elements of the form $(\mathsf{h}(id), \mathsf{h}(cert, dg^{a}, dg^{rv}))$. It represents all domains associated to rgx. id is an instance of rgx and is the subject of master certificate $cert$. dg^{a} and dg^{rv} are digests of two ordered data structures each of which respectively stores a set of active and revoked TLS certificates. In addition, data in the structure represented by dg^{rgx} and dg^{id} are ordered by rgx and $\mathsf{h}(id)$, respectively; data in the structure represented by dg^{a} and dg^{rv} are ordered by the subject of TLS certificates.

Note that requests $\mathsf{upadd}(\mathsf{h}(id), h)$ and $\mathsf{updel}(\mathsf{h}(id), h)$ are made according to the mapping log. Even though these modifications are not requested by domain owners, it is important to record them in the certificate log to ensure the transparency of the log maintainer's behaviour. Request $\mathsf{upadd}(\mathsf{h}(id), h)$ states that the certificate log maintainer is authorised to manage certificates for the domain name id from now on, and the current status of certificates for id is represented by h, where $h = \mathsf{h}(cert, dg^{a}, dg^{rv})$ for some master certificate $cert$ and some digest dg^{a} and dg^{rv} representing the active and revoked certificates of id. h is the value obtained from the certificate log that is previously authorised to manage certificates for domain id. Similarly, request $\mathsf{updel}(\mathsf{h}(id), h)$ indicates that the certificate log cannot manage certificates for domain id any more according to the request in the mapping log.

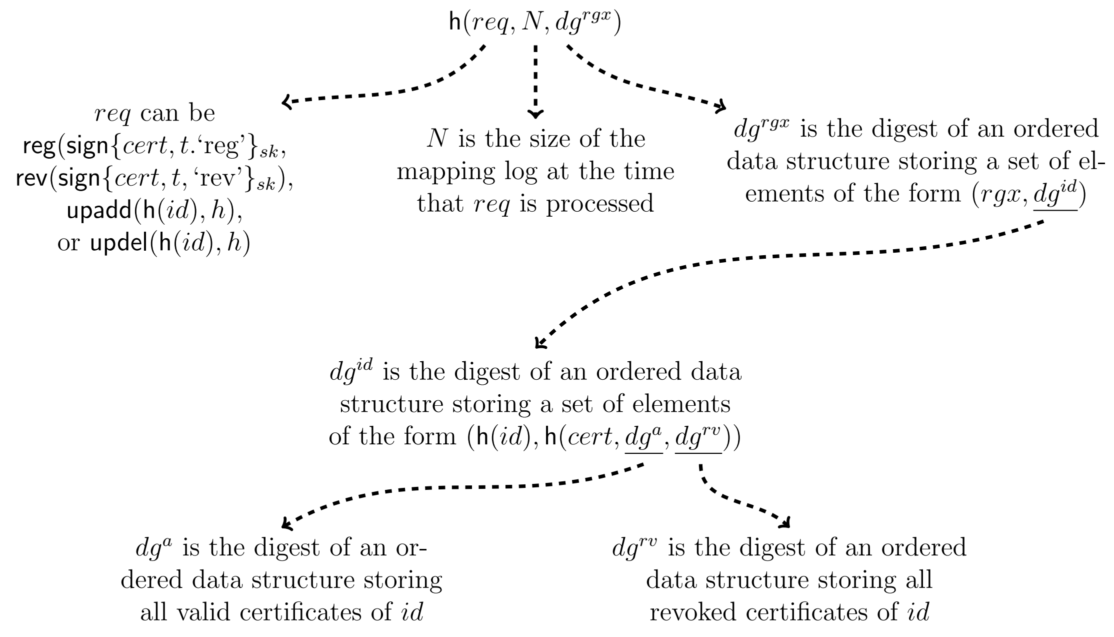

$\mathsf{h}(req, N, dg^{rgx})$
req can be $\mathsf{reg}(\mathsf{sign}\{cert, t.\text{‘reg’}\}_{sk}$, $\mathsf{rev}(\mathsf{sign}\{cert, t, \text{‘rev’}\}_{sk})$, $\mathsf{upadd}(\mathsf{h}(id), h)$, or $\mathsf{updel}(\mathsf{h}(id), h)$
N is the size of the mapping log at the time that req is processed
dg^{rgx} is the digest of an ordered data structure storing a set of elements of the form $(rgx, \underline{dg^{id}})$
dg^{id} is the digest of an ordered data structure storing a set of elements of the form $(\mathsf{h}(id), \mathsf{h}(cert, \underline{dg^{a}}, \underline{dg^{rv}}))$
dg^{a} is the digest of an ordered data structure storing all valid certificates of id
dg^{rv} is the digest of an ordered data structure storing all revoked certificates of id

Figure 4.4: A figure representation of the format of each record in the certificate log.

4.3.4 Synchronising the mapping log and certificate logs

The mapping log periodically (e.g. every day) publishes a signature $\mathsf{sign}\{t, dg, N\}_{sk}$, called *signed Mlog time-stamp*, on a time t indicating the publishing time, and the digest dg and size N of the mapping log. Similarly, certificate log maintainers also publishes their *signed Clog time-stamp* periodically. Mirrors need to download these signed data, and update their copy of logs when the logs are updated. A signed time-stamp is only valid for a short period (e.g. one day). Note that mirrors can provide the same set of proofs as the log maintainers, since they have a complete copy of the logs. Mirrors are not required to be trusted, because they do not need to sign anything, and a mirror which altered a log cannot convince browsers to accept it since the mirror cannot forge a signed time-stamp.

When a mapping log maintainer needs to update the mapping log, he requests all certificate log maintainers to perform the required update, and expects to receive the digest and size of all certificate logs once they are updated. After the mapping log maintainer receives these confirmations from all certificate log maintainers, he publishes the series of update requests in the mapping log, and appends an extra constant request end after them in the log to indicate that the update is done.

Log maintainers only answer requests according to their newly updated log if the mapping log maintainer has published the update requests in the mapping log. If requests received during the log update period, log maintainers will answer to the request according to the version of their log before the update started.

We say that the mapping log and certificate logs are *synchronised*, if certificate logs have completed the log update according to the request in the mapping log. Note that a mis-behaving certificate log maintainer (e.g. one recorded fake certificates in his log, or did not correctly update his log according to the request of the mapping log) can be terminated by the mapping log maintainer by putting the certificate log maintainer's identity into the blacklist, which is organised as an ordered data structure represented by dg^{bl} (as presented in 4.3.2).

4.4 Detailed implementation

Distributed transparent key infrastructure (DTKI) contains three main protocols, namely certificate publication, certificate verification, and log verification. In the certificate publication protocol, domain owners can upload new certificates and revoke existing certificates in the certificate log they are assigned to; in the certificate verification protocol, one can verify the validity of a certificate; and in the log verification protocol, one can verify whether a log behaves correctly.

Let Alice be an internet user who wants to securely communicate with a domain owner Bob who maintains the domain *example.com*.

4.4.1 Certificate publication

To insert or revoke certificates in the certificate log, the domain owner Bob needs to know which certificate log is authorised to record certificates for his domain. This can be done by communicating with the maintainer of (a mirror of) the mapping log. We detail the protocol for requesting the mapping for Bob's domain.

Request mappings Upon receiving the request, the mirror locates the certificate of the authorised certificate log maintainer, and generates proofs that

a) the provided information is the latest information; and

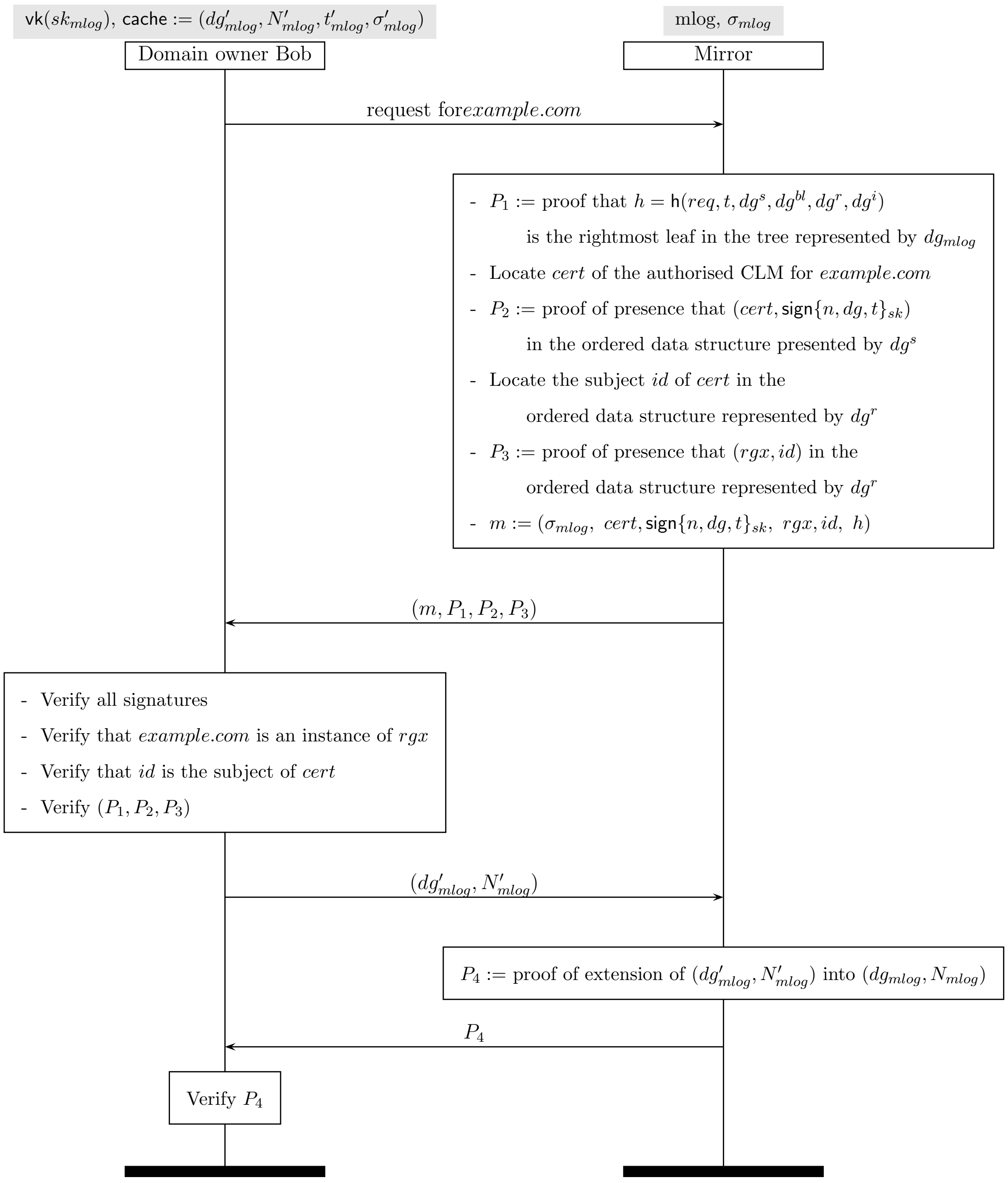

$\mathsf{vk}(sk_{mlog})$, $\mathsf{cache} := (dg'_{mlog}, N'_{mlog}, t'_{mlog}, \sigma'_{mlog})$
Domain owner Bob
mlog, σ_{mlog}
Mirror
request for*example.com*
- $P_1 :=$ proof that $h = \mathsf{h}(req, t, dg^s, dg^{bl}, dg^r, dg^i)$ is the rightmost leaf in the tree represented by dg_{mlog}
- Locate *cert* of the authorised CLM for *example.com*
- $P_2 :=$ proof of presence that $(cert, \mathsf{sign}\{n, dg, t\}_{sk})$ in the ordered data structure presented by dg^s
- Locate the subject *id* of *cert* in the ordered data structure represented by dg^r
- $P_3 :=$ proof of presence that (rgx, id) in the ordered data structure represented by dg^r
- $m := (\sigma_{mlog},\ cert, \mathsf{sign}\{n, dg, t\}_{sk},\ rgx, id,\ h)$
(m, P_1, P_2, P_3)
- Verify all signatures
- Verify that *example.com* is an instance of *rgx*
- Verify that *id* is the subject of *cert*
- Verify (P_1, P_2, P_3)
(dg'_{mlog}, N'_{mlog})
$P_4 :=$ proof of extension of (dg'_{mlog}, N'_{mlog}) into (dg_{mlog}, N_{mlog})
P_4
Verify P_4

Figure 4.5: The protocol presenting how domain owner Bob requests a mapping for his domain *example.com* with a mirror of mapping log. In which, $\sigma'_{mlog} = \mathsf{sign}\{t'_{mlog}, dg'_{mlog}, N'_{mlog}\}_{sk_{mlog}}$, and $\sigma_{mlog} = \mathsf{sign}\{t_{mlog}, dg_{mlog}, N_{mlog}\}_{sk_{mlog}}$

b) the certificate for the certificate log maintainer is recorded in the log;

c) the certificate log maintainer is authorised for the domain.

Loosely speaking, proof a) is the proof that both dg^s and dg^r are present in the latest record of the mapping log; proof b) is the proof that the certificate with subject id is present in dg^s; and proof c) is the proof that the mapping from regular expression rgx to identity id is present in the digest dg^r (as presented in the mapping log structure), such that *example.com* is an instance of rgx, and id is the identity of the certificate log maintainer. All proofs should be linked to the latest digest signed by the mapping log maintainer. If Bob has previously observed a version of the mlog, then a proof that the current mlog is an extension of the version that Bob observed will also be provided.

Bob accepts the response if all proofs are valid. He then stores the verified data in his cache for future connection until the signed digest is expired.

In more detail, after a mirror receives a request from Bob, the mirror obtains the data of the latest element of its copy of the mapping log, denoted $h = \mathsf{h}(req, t, dg^s, dg^{bl}, dg^r, dg^i)$, and generates the proof of its presence in the digest (denoted dg_{mlog}) of its log of size N. Then, it generates the proof of presence of the element $(cert, \mathsf{sign}\{n, dg, t\}_{sk})$ in the digest dg^s for some $\mathsf{sign}\{n, dg, t\}_{sk}$. This proves that the certificate log associated to $cert$ is still active. Moreover, it generates the proof of presence of some element (rgx, id) in the digest dg^r, where id is the subject of $cert$ and *example.com* is an instance of the regular expression rgx. This proves that id is authorised to store the certificates of *example.com*. The mirror then sends to Bob the hash h, the signature $\mathsf{sign}\{n, dg, t\}_{sk}$, the regular expression rgx, the three generated proofs of presence, and the latest *signed Mlog time-stamp* containing the time t_{mlog}, and digest dg_{mlog} and size N_{mlog} of the mapping log.

Bob first verifies the received *signed Mlog time-stamp* with the public key of the mapping log maintainer embedded in the browser, and verifies whether t_{Mlog} is valid or not. Then Bob checks that *example.com* is an instance of rgx, and verifies the three different proofs of presence. If all checks hold, then Bob sends the *signed Mlog time-stamp* containing $(t'_{Mlog}, dg'_{mlog}, N'_{mlog})$ that he stored during a previous connection, and expects to receive a proof of extension of (dg'_{mlog}, N'_{mlog}) into (dg_{mlog}, N_{mlog}). If the received proof of extension is valid, then Bob stores the current *signed Mlog time-stamp*, and believes that the certificate log with identity id, certificate $cert$, and size that should be no smaller than n, is currently authorised for managing certificates for his domain.

Insert and revoke certificates At the first time when Bob wants to publish a certificate for his domain, he needs to generate a pair of master signing key, denoted sk_m, and verification key. The latter is sent to a certificate authority, which verifies Bob's identity and issues a master certificate $cert_m$ for Bob. After Bob receives his master certificate, he checks the correctness of the information in the certificate. The TLS certificate can be obtained in the same way.

To publish the master certificate, Bob signs the certificate together with the current time t by using the master signing key sk_m, and sends it together with the request *AddReq* to the authorised certificate log maintainer whose signing key is denoted sk_{clog}. The certificate log maintainer checks whether there exists a valid master certificate for *example.com*; if there is one, then the log maintainer aborts the conversation. Otherwise, the log maintainer verifies the validity of time t and the signature.

If they are all valid, the log maintainer updates the log, generates the proof of presence that the master certificate for Bob is included in the log, and sends the signed proof and the updated digest of the log back to Bob. If the signature and the proof are valid, and the size of the log is no smaller than what the mirror says, then Bob accepts and stores the response as an evidence of successful certificate publication. If Bob has previously observed a version of the clog, then a proof that the current clog is an extension of the version that Bob observed is also required.

Figure 4.6 presents the detailed process to publish the master certificate $cert_m$. After a log maintainer receives and verifies the request from Bob, the log maintainer updates the log, generates the proof of presence of $(\mathsf{h}(id), \mathsf{h}(cert_m, dg^a, dg^{rv}))$ in dg^{id}, (rgx, dg^{id}) in dg^{rgx}, and $\mathsf{h}(\mathsf{reg}(\mathsf{sign}\{cert_m, t, \text{`reg'}\}_{sk_m}), N_{mlog}, dg^{rgx})$ is the last element in the data structure represented by $dg_{clog}$, where $id$ is the subject of $cert_m$ and an instance of $rgx$; $\mathsf{reg}(\mathsf{sign}\{cert_m, t, \text{`reg'}\}_{sk_m})$ is the register request to adding $cert_m$ into the certificate log with digest dg_{clog} at time t. The log maintainer then issues a signature on (dg_{clog}, N, h), where N is the size of the certificate log, and $h = \mathsf{h}((rgx, dg^{id}), dg^{rgx}, P)$, where P is the sequence of the generated proofs, and sends the signature σ_2 together with $(dg_{clog}, N, rgx, dg^{id}, dg^{rgx}, dg^{a}, dg^{rv}, P)$ to Bob. If the signature and the proof are valid, and N is no smaller than the size n contained in the *signed Mlog time-stamp* that Bob received from the mirror, then Bob stores the signed (dg_{clog}, N, h), sends the previous stored (dg'_{clog}, N') to the certificate log maintainer, and expects to receive a proof of extension of (dg'_{clog}, N') into (dg_{clog}, N). If the received proof of extension is valid, then Bob believes that he has successfully published the new certificate.

Note that it is important to send (dg'_{clog}, N') after receiving (dg_{clog}, N), because otherwise the log maintainer could learn the digest that Bob has, then give a pair (dg''_{clog}, N'') of digest and size of the log such that $N' < N'' < N$. This may open a window to attackers who wants to convince Bob to use a certificate which was valid in dg''_{clog} but revoked in dg_{clog}.

In addition, if Bob has run the request mapping protocol more than once, and has obtained a digest that is different from his local copy of the corresponding certificate log, then he should ask the certificate log maintainer to prove that one of the digests is an extension of the other.

The process of adding a TLS certificate is similar to the process of adding a master certificate, but the log maintainer needs to verify that the TLS certificate is signed by the valid master signing key corresponding to the master certificate in the log.

To revoke a (master or TLS) certificate, the domain owner can perform a process similar to the process of adding a new certificate. For a revocation request with $\mathsf{sign}\{cert, t\}_{sk_m}$, the log maintainer needs to check that $\mathsf{sign}\{cert, t'\}_{sk_m}$ is already in the log and $t > t'$. This ensures that the same master key is used for the revocation.

4.4.2 Certificate verification

When Alice wants to securely communicate with *example.com*, she sends the connection request to Bob, and expects to receive a master certificate $cert_m$ and a signed TLS certificate $\mathsf{sign}\{cert, t\}_{sk_m}$ from him. To verify the received certificates, Alice checks whether the certificates are expired. If both of them are still in the validity time period, Alice requests (as described in 4.4.1) the corresponding mapping from

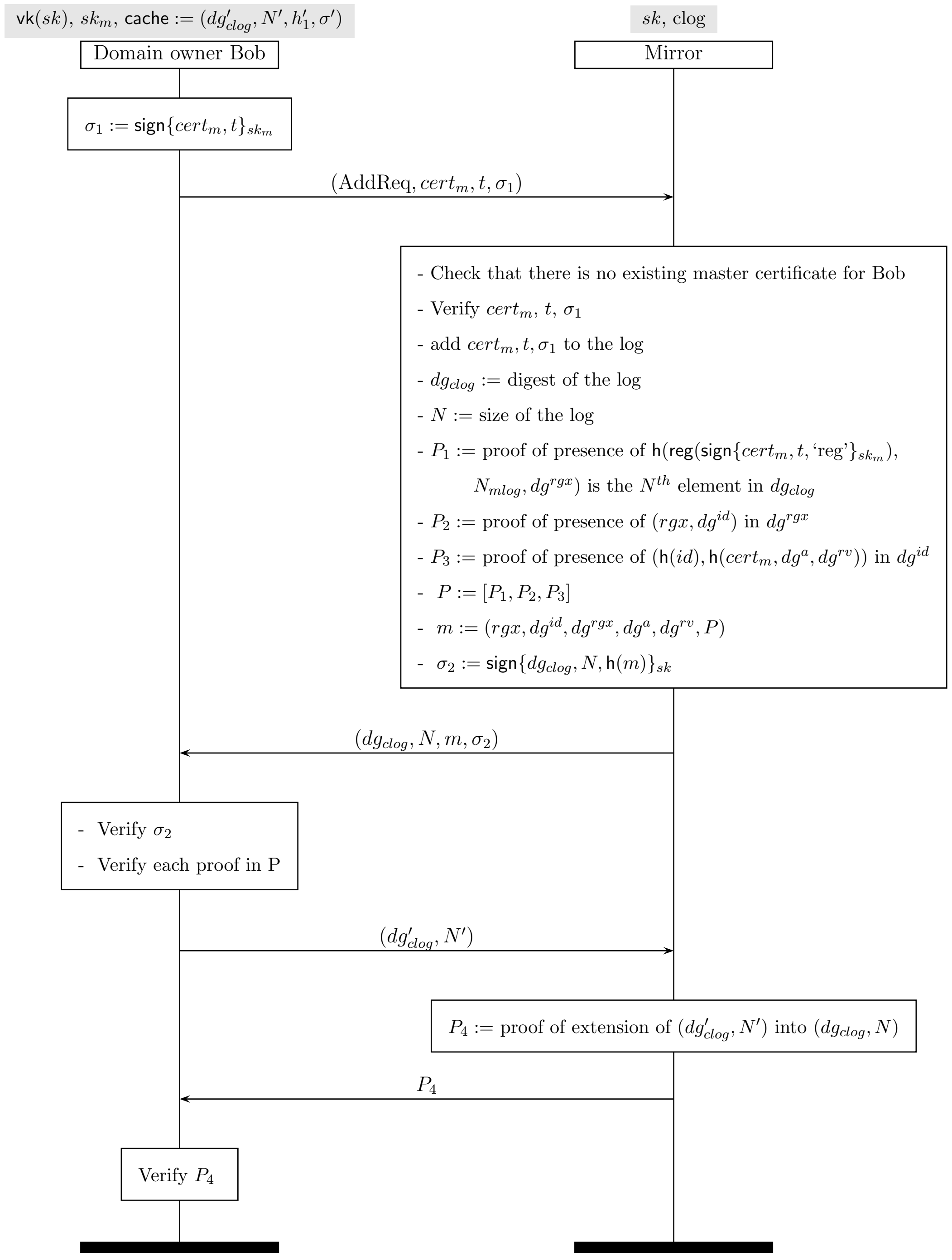

$\mathsf{vk}(sk)$, sk_m, $\mathsf{cache} := (dg'_{clog}, N', h'_1, \sigma')$
Domain owner Bob
sk, clog
Mirror
$\sigma_1 := \mathsf{sign}\{cert_m, t\}_{sk_m}$
$(\text{AddReq}, cert_m, t, \sigma_1)$
- Check that there is no existing master certificate for Bob
- Verify $cert_m$, t, σ_1
- add $cert_m, t, \sigma_1$ to the log
- dg_{clog} := digest of the log
- N := size of the log
- P_1 := proof of presence of $\mathsf{h}(\mathsf{reg}(\mathsf{sign}\{cert_m, t, \text{'reg'}\}_{sk_m}), N_{mlog}, dg^{rgx})$ is the N^{th} element in dg_{clog}
- P_2 := proof of presence of (rgx, dg^{id}) in dg^{rgx}
- P_3 := proof of presence of $(\mathsf{h}(id), \mathsf{h}(cert_m, dg^a, dg^{rv}))$ in dg^{id}
- $P := [P_1, P_2, P_3]$
- $m := (rgx, dg^{id}, dg^{rgx}, dg^a, dg^{rv}, P)$
- $\sigma_2 := \mathsf{sign}\{dg_{clog}, N, \mathsf{h}(m)\}_{sk}$
$(dg_{clog}, N, m, \sigma_2)$
- Verify σ_2
- Verify each proof in P
(dg'_{clog}, N')
P_4 := proof of extension of (dg'_{clog}, N') into (dg_{clog}, N)
P_4
Verify P_4

Figure 4.6: The protocol presenting how domain owner Bob communicates with a mirror of the certificate log to publish a master certificate $cert_m$.

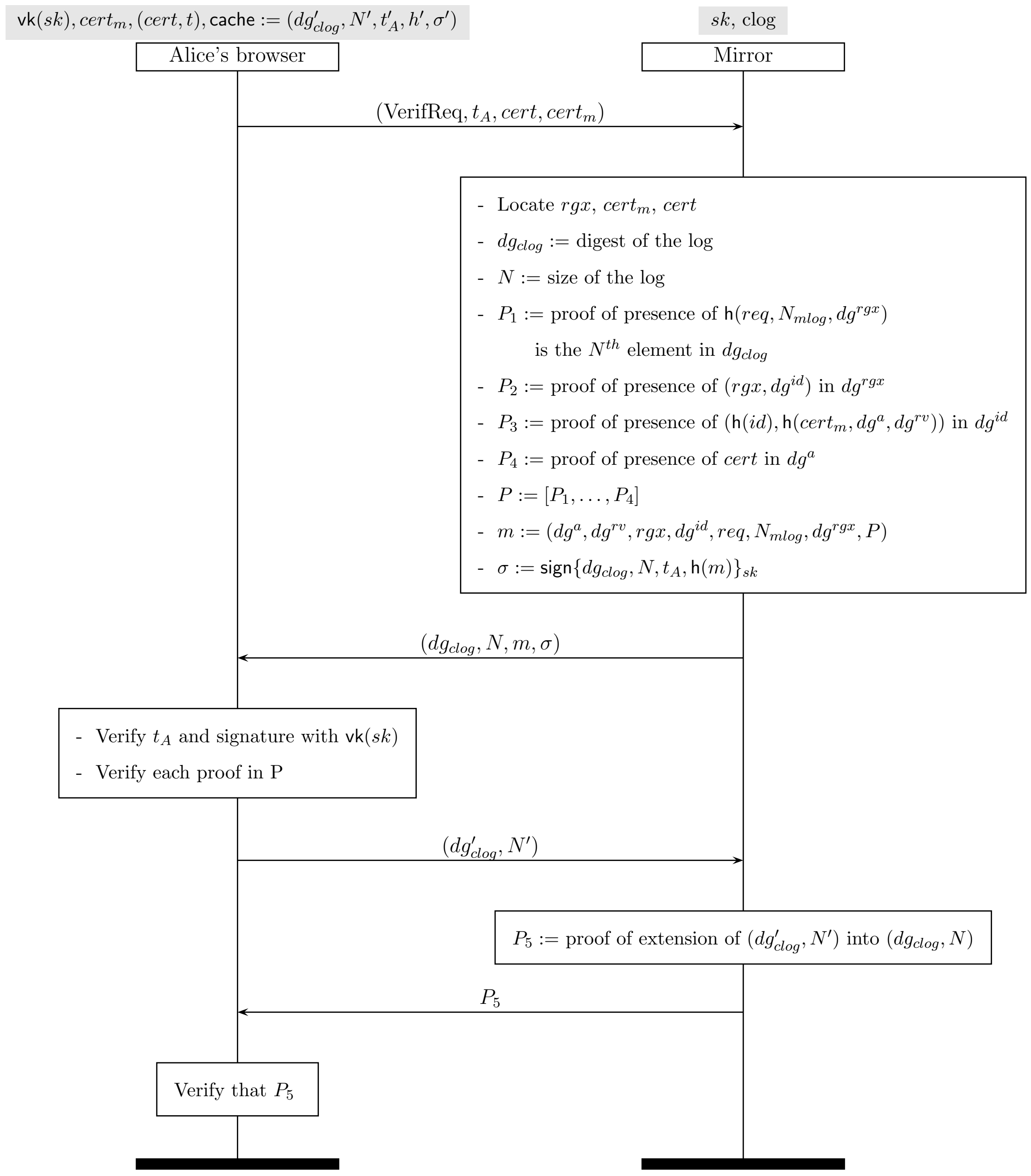

$\mathsf{vk}(sk), cert_m, (cert, t), \mathsf{cache} := (dg'_{clog}, N', t'_A, h', \sigma')$
sk, clog
Alice's browser
Mirror
$(\text{VerifReq}, t_A, cert, cert_m)$
- Locate rgx, $cert_m$, $cert$
- dg_{clog} := digest of the log
- N := size of the log
- P_1 := proof of presence of $\mathsf{h}(req, N_{mlog}, dg^{rgx})$ is the N^{th} element in dg_{clog}
- P_2 := proof of presence of (rgx, dg^{id}) in dg^{rgx}
- P_3 := proof of presence of $(\mathsf{h}(id), \mathsf{h}(cert_m, dg^a, dg^{rv}))$ in dg^{id}
- P_4 := proof of presence of $cert$ in dg^a
- $P := [P_1, \ldots, P_4]$
- $m := (dg^a, dg^{rv}, rgx, dg^{id}, req, N_{mlog}, dg^{rgx}, P)$
- $\sigma := \mathsf{sign}\{dg_{clog}, N, t_A, \mathsf{h}(m)\}_{sk}$
$(dg_{clog}, N, m, \sigma)$
- Verify t_A and signature with $\mathsf{vk}(sk)$
- Verify each proof in P
(dg'_{clog}, N')
P_5 := proof of extension of (dg'_{clog}, N') into (dg_{clog}, N)
P_5
Verify that P_5

Figure 4.7: The protocol for verifying a certificate with a mirror of the corresponding certificate log.

a mirror to find out the authorised certificate log for *example.com*, and communicates with the (mirror of) authorised certificate log maintainer to verify the received certificate.

Note that this verification requests extra communication round trips, but it gives a higher security guarantee. An alternative way is that Bob provides both certificates and proofs, and Alice verifies the received proofs directly.

The Figure 4.7 presents the detailed process of verifying a certificate. After Alice learns the identity of the authorised certificate log, she sends the verification request $VerifReq$ with her local time t_A and the received certificate to the certificate log maintainer. The time t_A is used to prevent replay attacks, and will later be used for accountability. The certificate log maintainer checks whether t_A is in an acceptable time range (e.g. t_A is in the same day as his local time). If it is, then he locates the corresponding (rgx, dg^{id}) in dg^{rgx} in the latest record of his log such that *example.com* is an instance of regular expression rgx, locates $(\mathsf{h}(id), \mathsf{h}(cert_m, dg^a, dg^{rv}))$ in dg^{id} and $cert$ in dg^a, then generates the proof of presence of $cert$ in dg^a, $(\mathsf{h}(id), \mathsf{h}(cert_m, dg^a, dg^{rv}))$ in dg^{id}, (rgx, dg^{id}) in dg^{rgx}, and $\mathsf{h}(req, N_{mlog}, dg^{rgx})$ is the latest record in the digest dg_{clog} of the log with size N. Then, the certificate log maintainer signs (dg_{clog}, N, t_A, h), where $h = \mathsf{h}(m)$ such that $m = (dg^a, dg^{rv}, rgx, dg^{id}, req, N_{mlog}, dg^{rgx}, P)$, and P is the set of proofs, and sends (dg_{clog}, N, σ) to Alice.

Alice should verify that N_{mlog} is the same as her local copy of the size of mapping log. If the received N_{mlog} is greater than the copy, then it means that the mapping log is changed (it rarely happens) and Alice should run the request mapping protocol again. If N_{mlog} is smaller, then it means the certificate log maintainer has misbehaved. Alice then verifies the signature and proofs, and sends the previously stored dg'_{clog} with the size N' to the log maintainer, and expects to receive the proof of extension of (dg'_{clog}, N') into (dg_{clog}, N). If they are all valid, then Alice replaces the corresponding cache by the signed (dg_{clog}, N, t_A, h) and believes that the certificate is an authentic one.

In order to preserve privacy of Alice's browsing history, instead of asking Alice to query all proofs from the log maintainer, Alice can send the request to Bob who will redirect the request to the log maintainer, and redirect the received proofs from the log maintainer to Alice.

With DTKI, Alice is able to verify whether Bob's domain has a certificate by querying the proof of absence of certificates for *example.com* in the corresponding certificate log. This is useful to prevent TLS stripping attacks, where an attacker can maliciously convert an HTTPS connection into an HTTP connection.

4.4.3 Log verification

Users of the system need to verify that the mapping log maintainer and certificate log maintainers did update their log correctly according to the requests they have received, and certificate log maintainers did follow the latest mappings specified in the mapping log.

These checks can be easily done by a trusted monitor. However, to reduce the need of trusted parties, DTKI uses a crowdsourcing-like method, based on random checking, to monitor the correctness of the public log. The basic idea of random checking is that each user randomly selects a record in the log, and verifies whether the request and data in this record have been correctly managed. If all records are verified, the entire log is verified. Users only need to run the random checking periodically (e.g. once a day). We provide some examples of the random checking. Example 4.7 presents the random checking process to verify the correct behaviour of the mapping log.

Example 4.7. *Suppose verifier has randomly selected the k^{th} record of the mapping log, and the record has the form $\mathsf{h}(\mathsf{add}(rgx, id), t_k, dg^s_k, dg^{bl}_k, dg^r_k, dg^i_k)$. The verifier should check that all digests in this record are updated from the $(k-1)^{th}$ record by adding a new mapping (rgx, id) in the mapping log at time t_k.*

Let the label of the $(k-1)^{th}$ record be $\mathsf{h}(req_{k-1}, t_{k-1}, dg^s_{k-1}, dg^{bl}_{k-1}, dg^r_{k-1}, dg^i_{k-1})$, then to verify the correctness of this record, the verifier should run the following process:

- $t_k > t_{k-1}$;
- *verify that $dg^s_k = dg^s_{k-1}$ and $dg^{bl}_k = dg^{bl}_{k-1}$; and*
- *verify that dg^r_k is the result of adding (rgx, id) into dg^r_{k-1}; and*
- *verify that dg^i_k is the result of replacing (id, dg^{irgx}_{k-1}) in dg^i_{k-1} by (id, dg^{irgx}_k); and*

- *verify that dg_k^{irgx} is the result of adding rgx into dg_{k-1}^{irgx}.*

Note that all proofs required in the above are given by the log maintainer. If the above tests succeed, then the mapping log maintainer has behaved correctly for this record.

Every time a certificate log maintainer is blacklisted by the mapping log maintainer, Bob checks the authenticity of the master certificate for his domain stored in the corresponding certificate log.

In addition, we need to ensure that the mapping log maintainer and certificate log maintainers behaved honestly. In particular, we need to ensure that the mapping log maintainer and certificate log maintainers did update their log correctly according to the request, and certificate log maintainers did follow the latest mappings specified in the mapping log.

The verification on the certificate log is similar to the mapping log. However, there is one more thing needed to be verified – the synchronisation between the mapping log and certificate logs. This verification includes that the certificate log only manage the certificates for domains they are authorised to (according to the mapping log); and if there are modifications on the mapping, then the corresponding certificate log maintainer should add or remove all certificates according to the modified mapping. We present an example to show what a verifier should do to verify that the certificate log was authorised to add or remove a certificate.

Example 4.8. *Suppose the verifier has randomly selected the k^{th} record of a certificate log, and the record has the form $\mathsf{h}(\mathsf{reg}(\mathsf{sign}\{cert_{TLS}, t, \text{'}reg\text{'}\}_{sk}), N_k, dg_k^{rgx})$, where dg_k^{rgx} is the digest of ordered sequence of format (rgx, dg_k^{id}), dg_k^{id} is the digest of ordered sequence of format $(\mathsf{h}(id), \mathsf{h}(cert_m, dg_k^a, dg_k^{rv}))$, $cert_m$ is a master certificate, and $cert_{TLS}$ is a TLS certificate. Let dg_{k-1}^{rgx} be the digest in the $k-1^{th}$ record, and similarly for dg_{k-1}^{id}, dg_{k-1}^{a} ,dg_{k-1}^{rv}. Let the subject of $cert_{TLS}$ be id'. The verifier should verify the following tests:*

- *Verify that $\mathsf{sign}\{cert_{TLS}, t\}_{sk}$ is correctly signed according to $cert_m$; and*

- *Verify that $cert_m$ is not expired, and shares the same subject id' with $cert_{TLS}$, and $id' = id$; and*

- *Verify that dg_k^a is the result of adding $cert_{TLS}$ into dg_{k-1}^a; and*

- *Verify that* dg_k^{id} *is the result of replacing* $(\mathsf{h}(id), \mathsf{h}(cert_m, dg_{k-1}^a, dg_{k-1}^{rv}))$ *by* $(\mathsf{h}(id), \mathsf{h}(cert_m, dg_k^a, dg_k^{rv}))$ *in* dg_{k-1}^{id}*; and*
- *Verify that* $dg_k^{rv} = dg_{k-1}^{rv}$*; and*
- *Verify that* dg_k^{rgx} *is the result of replacing* (rgx, dg_{k-1}^{id}) *by* (rgx, dg_k^{id}) *in* dg_{k-1}^{rgx}*; and*
- *Verify that* (rgx', id'') *is in the* $dg_{N_k}^r$ *of the* N_k^{th} *element of the mapping log, such that* $rgx' = rgx$*, and* id' *is the identity of the certificate log.*

If the above tests succeed, then the certificate log maintainer behaves correctly on this record.

4.5 Security Analysis

We consider an adversary who can compromise the private key of all infrastructure servers in DTKI. In other words, the adversary can collude with all log servers and certificate authorities to launch attacks.

Main result Our security analysis shows that

- if the distributed random checking has verified all required tests, and domain owners have successfully verified their initial master certificates, then DTKI can prevent attacks from the adversary; and
- if the distributed random checking has not completed all required tests, or domain owners have not successfully verified their initial master certificates, then an adversary can launch attacks, but the attacks will be detected afterwards.

We analyse the main security properties of the DTKI protocol using the TAMARIN prover [MSCB13]. Since TAMARIN prover supports an unbounded number of instances and reasoning about protocols with mutable global state, it is suitable for our log-based protocol. We provide all source codes and files required to understand and reproduce our security analysis at Appendix A. In particular, these include the complete DTKI models and the verified proofs.

Modeling aspects We used several abstractions during modeling. We model our log as lists, similar to the abstraction used in [BCK$^+$14]. We also assume that the random checking is verified, and participants can see the same log.

We model the protocol roles D (domain server), M (mapping log maintainer), C (certificate log maintainer), and CA (certificate authority) by a set of rewrite rules. Each rewrite rule typically models receiving a message, taking an appropriate action, and sending a response message. Our modeling approach is similar to the one used in most TAMARIN models. Our modeling of the roles directly corresponds to the protocol descriptions in the previous sections. TAMARIN provides built-in support for a Dolev-Yao style network attacker, i.e., one who is in full control of the network. We additionally specify rules that enable the attacker to compromise service providers, namely the mapping log maintainer, certificate log maintainers and CAs, learn their secrets, and modify public logs.

Our final DTKI model (presented in Appendix A) consists of 959 lines for the base model and five main property specifications, examples of which we will give below.

Proof goals We state several proof goals for our model, exactly as specified in TAMARIN's syntax. Since TAMARIN's property specification language is a fragment of first-order logic, it contains logical connectives (`|`, `&`, `==>`, `not`, ...) and quantifiers (`All`, `Ex`). In TAMARIN, proof goals are marked as `lemma`. The `#`-prefix is used to denote timepoints, and "`E @ #i`" expresses that the event E occurs at timepoint i.

The first goal is a check for executability that ensures that our model allows for the successful transmission of a message. It is encoded in the following way.

```
lemma protocol_correctness:
 exists-trace
 " /* It is possible that */
   Ex D Did m rgx ltpkD stpkD #i1.

   /* The user has sent an encrypted message aenc{m}stpkD to domain
   server D whose identity is Did and TLS key is stpk, and the user
   received from D a confirmation h(m) of receipt. */

       Com_Done(D, Did, m, rgx, ltpkD, stpkD) @ #i1

     /* without the adversary compromising any party. */
   &   not (Ex #i2 CA ltkCA.
                Compromise_CA(CA,ltkCA) @ #i2)

   &   not (Ex #i3 C ltkC.
                Compromise_CLM(C,ltkC) @ #i3)

   &   not (Ex #i4 M ltkM.
                Compromise_MLM(M,ltkM) @ #i4)

"
```

The property holds if the Tamarin model exhibits a behaviour in which a domain server received a message without the attacker compromising any service providers. This property mainly serves as a sanity check on the model. If it did not hold, it would mean our model does not model the normal (honest) message flow, which could indicate a flaw in the model. Tamarin automatically proves this property in 49 steps and generates the expected trace in the form of a graphical representation of the rule instantiations and the message flow.

We additionally proved several other sanity-checking properties to minimize the risk of modeling errors.

The second example goal is a secrecy property with respect to a classical attacker, and expresses that when no service provider is compromised, the attacker cannot learn the message exchanged between a user and a domain server. Note that `K(m)` is a special event that denotes that the attacker knows m at this time.

```
lemma message_secrecy_no_compromised_party:
 "
 All D Did m rgx ltpkD stpkD #i1.

   /* The user has sent an encrypted message aenc{m}stpkD to domain
   server D whose identity is Did and TLS key is stpk, and the user
   received from D a confirmation h(m) of receipt. */

       (Com_Done(D, Did, m, rgx, ltpkD, stpkD) @ #i1

/* and no party has been compromised */
   &   not (Ex #i2 CA ltkCA.
                Compromise_CA(CA,ltkCA) @ #i2)

   &   not (Ex #i3 C ltkC.
                Compromise_CLM(C,ltkC) @ #i3)
```

```
  &   not (Ex #i4 M ltkM.
              Compromise_MLM(M,ltkM) @ #i4)
     )
     ==>
     ( /* then the adversary cannot know m */
      not (Ex #i5. K(m) @ #i5)
     )
 "
```

Tamarin proves this property automatically (in 575 steps).

The above result implies that if a domain server D, whose domain name is Did such that Did is an instance of regular expression rgx, receives a message that was sent by a user, and the attacker did not compromise server providers, then the attacker will not learn the message.

The next two properties encode the unique security guarantees provided by our protocol, in the case that even all service providers are compromised.

The first main property we prove is that when all service providers (i.e. CAs, the MLM, and CLMs) are compromised, and the domain owner has successfully verified his master certificate in the log, then the attacker cannot learn the message exchanged between a user and a domain owner. It is proven automatically by Tamarin in 5369 steps.

```
lemma message_secrecy_compromise_all_domain_verified_master_cert:
 "
 All D Did m rgx ltpkD stpkD #i1.

   /* The user has sent an encrypted message aenc{m}stpkD to domain
   server D whose identity is Did and TLS key is stpk, and the user
   received from D a confirmation h(m) of receipt. */

   (Com_Done(D, Did, m, rgx, ltpkD, stpkD) @ #i1

   /* and at an earlier time, the domain server has verified his
      master certificate */
     & Ex #i2.
     VerifiedMasterCert(D, Did, rgx, ltpkD) @ #i2
     & #i2 < #i1

      )
      ==>
      ( /* then the adversary cannot know m */
       not (Ex #i3. K(m) @ #i3)
      )
 "
```

The property states that if a domain server D receives a message that was sent by a user, and at an earlier time, the domain server has verified his master certificate, then even if the attacker can compromise all server providers, the attacker cannot learn the message.

The final property states that when all service providers can be compromised, and a domain owner has not verified his/her master certificate, and the attacker learns the message exchanged between a user and the domain owner, then afterwards the domain owner can detect this attack by checking the log. It is also verified by Tamarin within a few minutes.

```
lemma detect_bad_records_in_the_log_when_master_cert_not_verified:

 "
 All D Did m rgx ltpkD flag stpkD #i1 #i2 #i3.

   /* The user has sent an encrypted message aenc{m}stpkD to domain
   server D whose identity is Did and TLS key is stpk, and the user
   received from D a confirmation h(m) of receipt. */

   (Com_Done(D, Did, m, rgx, ltpkD, stpkD) @ #i1

   /* and the adversary knows m */

     &  K(m) @ #i2

   /* and we afterwards check the log */
     &  CheckedLog(D, Did, rgx, ltpkD, flag, stpkD) @ #i3
     &   #i1 < #i3)
      ==>
     ( /* then we can detect a fake record in the log */
        (flag = 'bad')
     )
 "
```

4.6 Performance Evaluation

Assumptions We assume that the size of a certificate log is 10^8 (the total number of registered domain names currently is 2.71×10^8 [Dom14], though only a fraction of them have certificates). In addition, we assume that the number of stored regular expressions, the number of certificate logs, and the size of the mapping log are 1000 each. (In fact, if we assume a different number or size (e.g. 100 or 10000) for them, it makes almost no difference to the conclusion). Moreover, in the certificate log, we assume that the size of the set of data represented by dg^{rgx} is 10, by dg^{id} is 10^5, by dg^{a} is 10, and by dg^{rv} is 100. These assumptions are based on the fact that dg^{rgx} represents the set of regular expressions maintained by a certificate log; the dg^{id} represents the set of domains which is an instance of a regular expression; and dg^{a} and dg^{rv} represent the set of currently valid certificates and the revoked certificates, respectively. Furthermore, we assume that the size of a certificate is 1.5 KB, the size of a signature is 256 bytes, the length of a regular expression and an identity is 20 bytes each, and the size of a digest is 32 bytes.

Space Based on these assumptions, the approximate size of the transmitted data in the protocol for publishing a certificate is 4 KB, for requesting a mapping is 3 KB, and for verifying a certificate is 5 KB. Since the protocols for publishing a certificate and requesting a mapping are run occasionally, we mainly focus on the cost of the protocol for verifying a certificate, which is required to be run between a log server and a web browser in each secure connection.

By using Wireshark, we[2] measure that the size of data for establishing an HTTPS protocol to log-in to the internet bank of HSBC, Bank of America, and Citibank are 647.1 KB, 419.9 KB, and 697.5 KB, respectively. If we consider the average size ($\approx$588 KB) of data for these three HTTPS connections, and the average size ($\approx$6 KB) of data for their corresponding TLS establishment connections, we have that in each connection, DTKI incurs 83% overhead on the cost of the TLS protocol. However, since the total overhead of an HTTPS connection is around 588 KB, so the cost of DTKI only adds 0.9% overhead to each HTTPS connection, which we consider acceptable.

Time Our implementation uses a SHA-256 hash value as the digest of a log and a 2048 bit RSA signature scheme. The time to compute a hash[3] is ≈ 0.01 millisecond (ms) per 1KB of input, and the time to verify a 2048 bit RSA signature is 0.48 ms. The approximate verification time on the user side needed in the protocol for verifying certificates is 0.5 ms.

[2]We use a MacBook Air 1.8 GHz Intel Core i5, 8 GB 1600 MHz DDR3.

[3]SHA-256 on 64 byte size block.

Hence, on the user side, the computational cost on the protocol for verifying certificates incurs 83% on the size of data for establishing a TLS protocol, and 0.9% on the size of data for establishing an HTTPS protocol; the verification time on the protocol for verifying certificates is 1.25 % of the time for establishing a TLS session (which is approximately 40 ms measured with Wireshark on the TLS connection to HSBC bank).

4.7 Comparison

Following our evaluation of existing proposals (as shown in Table 3.2), the only feature that DTKI does not offer is offline verification. This is the sacrifice that DTKI made in exchange for a strong security guarantee.

As mentioned previously, DTKI builds upon a wealth of ideas from SK [Eck12], CT [LLK13], CIRT [Rya14], and AKI [KHP^{+}13]. It is more interesting to see the comparison among these systems in the same category, i.e. certificate management transparency. Figure 4.2 shows the dimensions along which DTKI aims to improve on those systems.

Compared with CT, DTKI supports revocation by enabling log providers to offer proofs of *absence* and *currency* of certificates. In CT, there is no mechanism for revocation. CT has proposed additional data structures to hold revoked certificates, and those data structures support proofs of their contents [LK12]. However, there is no mechanism to ensure that the data structures are maintained correctly in time.

Compared to CIRT, DTKI extends the log structure of CIRT to make it suitable for multiple log maintainers, and provides a stronger security guarantee as it prevents attacks rather than merely detecting them. In addition, the presence of the mapping log maintainer and multiple certificate log maintainers create some extra monitoring work. DTKI solves it by using a detailed crowd-sourcing verification system to distribute the monitoring work to all users' browsers.

Compared to AKI and ARPKI, in DTKI the log providers can give proof that the log is maintained append-only from one step to the next. The data structure in A(RP)KI does not allow this, and therefore they cannot give a verifiable guarantee to the clients that no data is removed from the log.

DTKI improves the support that CT and A(RP)KI have for multiple log providers.

In CT and AKI, domain owners wishing to check if there exists a log provider that has registered a certificate for him has to check all the log providers, and therefore the full set of log providers has to be fixed and well-known. This prevents new log providers being flexibly created, creating an oligopoly. In contrast, DTKI requires the browsers only to have the MLM public key built-in, minimising the oligopoly element.

In DTKI, trusted monitors are optional, as it uses crowd-sourced verification. More precisely, a trusted monitor's verification work can be done probabilistically in small pieces by users' browsers.

Unlike the mentioned previous work, DTKI allows the possibility that all service providers (i.e. the MLM, CLMs, and mirrors) to collude together, and can still prevent attacks. In contrast, SK and CT can only detect attacks, and to prevent attacks, A(RP)KI requires that not all service providers collude together. Similar to A(RP)KI, DTKI also assumes that the domain is initially registered by an honest party to prevent attacks, otherwise A(RP)KI and DTKI can only detect attacks. Similar to CT and AKI, DTKI also protects user privacy – instead of asking users to query all proofs from the log maintainer, the domain servers gather the proof of presence and proof of currency from the log maintainer, and send the certificate along with these proofs to the users. The user still needs to query the proof of consistency, but this does not expose the browsing history.

4.8 Discussion

Responding to incorrect proofs How should the browser (and the user) respond if a received proof (e.g., a proof of presence in the log) is incorrect? Such situations should be handled in the background by the software in the browser that verifies proofs, and be sent to domain owners for further investigations. The browser can also present errors to the user in the same way as the current state of the art. So, the user interface will remain the same. For example, a user might be shown two options, i.e. either to continue anyway, or not to trust the certificate and abort this connection. Another possible way is to hard fail if the verification has not been succeeded, as suggested by Google certificate transparency. However, this might be an obstacle for deploying DTKI in early stages.

	SK [Eck12]	CT [LLK13]	AKI [KHP+13]	ARPKI [BCK+14]	DTKI
Terminology					
Log provider	Time-line server	Log	Integrity log server (ILS)	ILS	Log maintainer
Log extension	-	Log consistency	-	-	Log extension
Trusted party	Mirror	Auditor & monitor	Validator	Validator†	-
Whether answers to queries rely on trusted parties or are accompanied by a proof					
Certificate-in-log query:	Rely	Proof	Proof	Proof	Proof
Certificate-current-in-log query:	Rely	Rely	Proof	Proof	Proof
Subject-absent-from-log query:	Rely	Rely	Proof	Proof	Proof
Log extension query:	Rely	Proof	Rely	Rely	Proof
Non-necessity of trusted monitors					
The role of trusted monitors can be distributed to browsers	No	No	No+	No+	Yes
Trust assumptions					
Not all service providers collude together	Yes	Yes	Yes	Yes	No
Domain is initially registered by an honest party	No	No	Yes*	Yes*	Yes*
Security guarantee					
Attacks detection or prevention	Detection	Detection	Prevention	Prevention	Prevention
Oligopoly issues					
Log providers required to be built into browser (oligopoly)	Yes	Yes	Yes	Yes	Only the MLM
Monitors required to be built into browser (oligopoly and trust non-agility)	Yes	No	Yes	Yes†	No

+ The system limits the trust in each server by letting them to monitor each other's behaviour.

* Without the assumption, the security guarantee is detection rather than prevention.

† The trusted party is optional.

Table 4.2: Comparison of log-based approaches to certificate management. **Terminology** helps compare the terminology used in the papers. **How queries rely on trusted parties** shows whether responses to browser queries come with proof of correctness or rely on the honesty of trusted parties. **Necessity of trusted parties** shows whether the TP role can be performed by browsers. **Trust assumptions** shows the assumption for the claimed security guarantee. **Oligopoly issues** shows the entities that browsers need to know about.

Coverage of random checking As mentioned previously, several aspects of the logs are verified by user's browsers performing randomly-chosen checks. The number of things to be checked depends on the size of the mapping log and certificate logs. The size of the mapping log mainly depends on the number of certificate logs and the mapping from regular expressions to certificate logs; and the size of certificate logs mainly depends on the number of domain servers that have a TLS certificate. Currently, there are 2.71×10^8 domains [Dom14] (though not every domain has a certificate), and 3×10^9 internet users [Int14]. Thus, if every user makes one random check per day, then everything will on average, be checked 10 times per day.

Another way to see this is that the probability of a given domain not being checked on a given day (or. week) is $(1 - \frac{1}{2.71\times10^8})^{3\times10^9} \approx 1.56 \times 10^{-5}$ (resp. $((1 - \frac{1}{2.71\times10^8})^{3\times10^9})^7 \approx 2.25 \times 10^{-34}$). Thus, the expected number of unchecked domains per day (resp. per week) is 4.23×10^3 (resp. 6.10×10^{-26}).

Gossip protocol A potential problem in CT or CIRT arises if an attacker shows different versions of the log to different clients. This is sometimes called the "bubble" problem; two clients in different bubbles could see different keys for the same subject. A gossip protocol is a mechanism that allows clients of a log to directly exchange digests of the log, in order to ensure that they have the same view of the log. If Alice holding digest dg_A receives a digest dg_B from Bob, she can challenge the log maintainer to prove that dg_A and dg_B are related by extension. Gossip protocols for log transparency are currently being specified [Nor14a, Nor14b, CSP+15]. In DTKI, we also assume gossip protocols [JVG+07] are used to disseminate digests of the log.

Accountability of mis-behaving parties The main goal of new certificate management schemes such as CT, CIRT, AKI, ARPKI and DTKI is to address the problem of mis-issued certificates, and to make the mis-behaving (trusted) parties accountable.

In DTKI, a domain owner can readily check for rogue certificates for his domain. First, he queries a mirror of the mapping log maintainer to find which certificate log maintainers (CLMs) are allowed to log certificates for the domain (section 4.4). Then he examines the certificates for his domain that have been recorded by those CLMs. The responses he obtains from the mirror and the CLMs are accompanied by proofs.

If he detects a mis-issued certificate, he requests revocation in the CLM. If that is refused, he can complain to the top-level domain, who in turn can request the MLM to change the CLM for his domain (after that, the offending CLM will no longer be consulted by browsers). This request should not be refused because the MLM is governed by an international panel. The intervening step, of complaining to the top-level domain, reflects the way domain names are actually managed in practice. Different top-level domains have different terms and conditions, and domain owners take them into account when purchasing domain names. In DTKI, log maintainers are held accountable because they sign and time-stamp their outputs. If a certificate log maintainer issues an inconsistent digest, this fact will be detected and the log maintainer can be blamed and blacklisted. If the mapping log misbehaved, then its governing panel must meet and resolve the situation.

In certificate transparency, this process is not as smooth. Firstly, the domain owner doesn't get proof that the list of issued certificates is complete; he needs to rely on monitors and auditors. Next, the process for raising complaints with log maintainers who refuse revocation requests is less clear (indeed, the RFC [LLK13] says that the question of what domain owners should do if they see an incorrect log record is beyond scope of their document). In CT, a domain owner has no ability to dissociate himself from a log maintainer and use a different one.

AKI addresses this problem by saying that log maintainer that refuses to unregister an record will eventually lose credibility through a process managed by validators, and will be subsequently ignored. The details of this credibility management are not very clear, but it does not seem to offer an easy way for domain owners to control which log maintainers are relied on for their domain.

Master certificate concerns One concern is that a CA might publish fake master certificates for domains that the CA doesn't own and are not yet registered. However, this problem is not likely to occur: CAs are businesses, they cannot afford the bad press from negative public opinion and they cannot afford the loss of reputation. Hence, they will only want to launch attacks that would not be caught. (Such an adversary model has been described by Franklin and Yung [FY92], Canetti and Ostrovsky [CO99], Hazay and Lindell [HL08], and Ryan [Rya14]). In DTKI, if a CA attempts to publish a fake master certificate for some domain, it will have to leave evidence of its misbehaviour in the log, and the misbehaviour will eventually be detected by the genuine domain owner.

Another concern is the assumption that the domain owners can securely handle their master keys. In practice, the domain owners might have problems looking after their master keys due to lack of awareness of good practices. This problem arises in any web PKI: it is assumed that domain owners can securely handle their TLS keys. Our system adds one more key (the master key) to that requirement. A possible practical solution for domain owners is to use a trustworthy service to handle TLS keys (and the master key); the details are beyond the scope of the thesis.

Avoidance of oligopoly As we mentioned in the introduction, the predecessors (SK, CT, CIRT, AKI, ARPKI) of DTKI do not solve a foundational issue, namely *oligopoly*. These proposals require that all browser vendors agree on a fixed list of log maintainers and/or validators, and build it into their browsers. This means there will be a large barrier to create a new log maintainer.

CT has some support for multiple logs, but it doesn't have any method to allocate different domains to different logs. In CT, when a domain owner wants to check whether mis-issued certificates are recorded in logs, he needs to contact all existing logs, and download all certificates in each of the logs, because there is no way to prove to the domain owner that no certificates for his domain is in the log, or to prove that the log maintainer has showed all certificates in the log for his domain to him. Thus, to be able to detect fake certificates, CT has to keep a very small number of log maintainers. This prevents new log providers being flexibly created, creating an oligopoly.

In contrast to its predecessors, DTKI does not have a fixed set of certificate log maintainers (CLMs) to manage certificates for domain owners, and it allows operations of adding or removing a certificate log maintainer by updating the mapping log. In DTKI, the public log of the MLM is the only thing that browsers need to know.

The MLM may be thought to represent a monopoly; to the extent that it does, it is likely to be a much weaker monopoly than the oligopoly of CAs or log maintainers. CAs and log maintainers offer commercial services and compete with each other, by offering different levels of service at different price points in different markets. The

MLM should not offer any commercial services; it should perform a purely administrative role, and is not required to be trusted because it behaves fully transparently and does not manage any certificates for web domains. In addition, the MLM is expected to be operated by an international panel with a lot of members. In practice, we expect ICANN to be the MLM, as it is responsible for coordinating name-spaces of the Internet, and is governed by a Governmental Advisory Committee containing representatives from 111 states. However, there might be concerns here, including the concern that ICANN might not be interested to be the MLM, due to the fact that the service won't generate any revenue. Our solution does not address political issues around making decisions of whether to add or remove some CLMs or not.

Additional latency DTKI introduces additional round-trips in the TLS connection to verify certificates and prevent potential attacks. This will add some extra latency to the TLS connection. This may be considered justified by the fact that DTKI offers a strong security guarantee.

In fact, the additional latency can be eliminated by delaying the added verification process from the user side. In this case, users obtain a slightly weaker security guarantee: they are still able to verify the authenticity of received certificates afterwards and therefore can detect mis-issued certificates.

Synchronisation concerns The synchronisation among a large number (e.g. thousands) of participants is normally a difficult task. However, in DTKI, the synchronisation among the MLM and CLMs is not expected to be a problem. First, the mapping log is rarely changed – it will be changed only if a new CLM has been added or terminated. In the steady state, this is likely to be no more than a few times per year. Second, the MLM can send the corresponding update request to CLMs in advance, and the synchronisation process is allowed to take an acceptable time period. During this time period, users will use the current logs until all logs are synchronised. Third, the MLM can terminate a CLM that has failed to update on time (e.g. have not finished the update process in a certain time period). So, in a long run, all parties will be able to do their work properly.

4.9 Conclusion

In the category of certificate management transparency, sovereign keys (SK), certificate transparency (CT), accountable key infrastructure (AKI), certificate issuance and revocation transparency (CIRT), and attack resilient PKI (ARPKI) are recent proposals to make public key certificate authorities more transparent and verifiable, by using public logs. CT is currently being implemented in servers and browsers. Google is building a certificate transparency log containing all the current known certificates, and is integrating verification of proofs from the log into the Chrome web browser.

Unfortunately, as it currently stands, CT risks creating an oligopoly of log maintainers (as discussed in section 4.8), of which Google itself will be a principal one. Therefore, adoption of CT risks investing more power about the way the internet is run in a company that arguably already has too much power.

We proposed DTKI – a transparent public key validation system using an improved construction of public logs. DTKI can prevent attacks based on mis-issued certificates, and minimises undesirable oligopoly situations by using the mapping log. In addition, since devising new security protocols is notoriously error-prone, we provide a formalisation of DTKI, and formally proved its security properties by using TAMARIN prover.

Part III

Key compromise in secure communication

CHAPTER 5

KEY USAGE DETECTION

5.1 Introduction

The previous chapter introduces how to provide the authenticity of public keys when an attacker is able to compromise certificate authorities. However, even if the authenticity of keys is guaranteed, the confidentiality of encrypted data still relies on another assumption that the associated decryption key has not been exposed to attackers. In particular, it requires that the device performing the crypto operations is free of malware. In practice, this assumption is hard to justify.

This chapter explores ways in which some security guarantees can be obtained, even if the end-point devices have software vulnerabilities that allow an attacker to obtain keys and/or control the device. Although we consider situation in which the device is controlled by an adversary, we assume that devices are *periodically trustworthy*. That is, a device may become vulnerable or infected at any time, but at some later time it will be again made secure. In other words, we assume that users periodically successfully perform malware scans, operating system upgrades, and software updates, bringing their devices back into a trustworthy state.

If a device is compromised by exploiting software vulnerabilities, and is then made secure again, the attacker remains in possession of secrets (such as keys) he obtained during the compromise. Since victims do not know when compromises take place, they are not motivated to revoke their keys. In reality, it is impractical to ask users to manually revoke their keys and distribute new ones after every security update.

We develop messaging protocols that allow users to detect if their long-term keys have been compromised and are being used by an attacker. It is clear that if a recipient's device becomes temporarily compromised and leaks all of its secrets, it is impossible to ensure the secrecy of messages sent during the compromised period. Informally speaking, we achieve the following unique security guarantee: if an attacker abuses previously compromised secrets to learn the contents of messages sent during trustworthy periods, for example by using the recipient's long-term key to impersonate him, then the recipient will detect this has happened. This detection is done by presenting information about which devices (keys) have been used, which the user can verify against her own experience and recollection (see Figure 5.3). We achieve this by exploiting *temporary asymmetric keys* to limit attack windows, whose public keys are sent to a *log server* to enable attack detection. We call this approach "key usage detection" (KUD).

We minimise the burden placed on users: in particular, reflecting the fact that it is perceived as a burden to do so, we do not require that users routinely change passwords and regenerate long-term keys. This means that an attacker that has compromised a device and obtained its secrets continues to possess the secrets even after the device has been restored to a trustworthy state.

Our proposal not only detects situations in which the adversary has copied a key and uses it, but also situations in which he has access to a key but is not able to copy it (for example, if a key is protected in a Trusted Platform Module (TPM)).

Use case We expect our protocol will be most useful for high-value targets that may be the victim of a determined attack. This can include politicians, senior company executives and civil servants that typically conduct business using commodity devices, as well as professional users, such as doctors, lawyers, engineers, patent attorneys, that have to deal in confidential documents and need this extra layer of security. These are cases in which attackers may be highly motivated to breach confidentiality.

Those users will take all the precautions available to ensure the confidentiality of messages they send and receive. Our protocol offers them an extra layer of security: it enables a user to detect if a malfeasor has obtained her private key, and is using it surreptitiously. Our protocol doesn't defend against all attacks, but it raises the bar for the attacker. It allows detection of key usage if the legitimate device in possession of the key is either restored to a secure state, or is switched off.

Key usage detection could be applied to solve other problems. One example is to apply it to identity-based signatures (IBS) to mitigate the key escrow problem, as it allows a signer to detect that an unauthorised signature (e.g. one made by the identity provider) has been issued.

We proceed in the following way. We detail our attacker model and security goals in Section 5.2 and present our key usage detection protocols in Section 5.3. The detailed implementation of our messaging protocol is presented in Section 5.4, and its security is formally proved in Section 5.5. We present performance evaluations in Section 5.6, and conclude this chapter in Section 5.7.

5.2 Threat model and design goals

Assumptions We assume a role called sender, that sends messages, and another one called receiver, that receives messages. Users can perform one or both of those roles. Each user has one or more devices, and can pick any of his/her devices to send a message, and can receive messages on any of them. We use **S**ally and **R**obert to refer to an arbitrary sender and receiver, respectively.

Threat model The attacker has control over the network, but not completely. This means he can eavesdrop, modify, insert and suppress any messages, and as many of them he wants, but he cannot suppress or modify *all* the messages. In other words, we assume that Sally and Robert can eventually exchange an unmodified message[1]. In addition, the attacker may compromise any user's devices at any time. After compromising a device, the attacker fully controls it, and can retrieve and store all the data (including secret keys) that are stored on it.

Periodically and routinely, users detect and remove malware on their devices, upgrade the operating system, and install software patches that remove known vulnerabilities. This can put the device back into a trustworthy state. The users do not regenerate long-term keys or change passwords.

Thus, we assume that devices are *periodically trustworthy.* An attacker compromises the device by exploiting a vulnerability, and sometime later the device owner

[1]In practice, this can be achieved in many ways, such as by using diverse channels. For example, although two Hotmail users can be intercepted completely when the adversary controls the Hotmail servers, they can still get an unmodified message through by using Gmail.

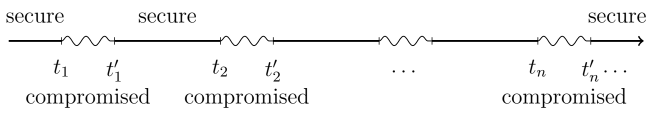


Figure 5.1: A device is compromised at time t_1, and then restored into a secure state at time t'_1. This cycle is repeated. Thus, the device is in a compromised state during the intervals $\{(t_j, t'_j) \mid j \in \{1, 2, 3, \ldots\}\}$.

restores it into a secure state. This cycle repeats, as illustrated in Figure 5.1.

The problem Once a device is compromised, then the victim's secrets stored in the device are exposed to the attacker. Performing security updates and removing malware is insufficient to prevent the attacker masquerading as the victim.

Security goals To solve this problem, our system detects key usages by the attacker. We state our security goal here, and explain how to achieve the goal in the following sections. In the security statements below, we assume a parameter ζ, which is a time period set by the user. A shorter ζ brings greater security. However, devices are automatically unregistered from the system if they are not used for periods longer than ζ, and have to be re-registered. Thus, a very short ζ reduces usability. Typically, ζ would be about two days. We discuss ζ and other system parameters later.

In the next section, we develop two protocols: the basic KUD protocol and full KUD messaging application. These offer slightly different guarantees.

- **Basic KUD protocol.**

  Suppose receiver Robert's device is compromised during the periods $\{(t_j, t'_j) \mid j \in N\}$. Suppose a message is sent by sender Sally at time t from a device in a trustworthy state, and the plaintext is obtained by the attacker. Robert can detect this attack provided his device

  - was well within a trustworthy state when the message was sent; that is, $t'_j + \zeta \leq t \leq t_{j+1} - \zeta$ for some j.

- **Messaging application (many users each with many devices).**

  Suppose Robert's devices are periodically compromised, as before: D_i is compromised during the intervals

$\{(t_{i,j}, t'_{i,j}) \mid j \in N\}$. Suppose a message is sent by Sally at time t from a device in a trustworthy state, and the plaintext is obtained by the attacker. Robert can detect this attack provided, for each of his devices D_i,

- D_i was well within a trustworthy state when the message was sent; that is, $t'_{i,j} + \zeta \leq t \leq t_{i,j+1} - \zeta$ for some j, or
- D_i was in a compromised state, but had not been used by Robert since $t - \zeta$.

The last condition reflects the fact that one can tell that a device has been compromised if the device was not being used at the time its key was used. Later, in Section 5.3.2, we show the user interface that allows a user to check this.

As part of our solution, we introduce an auxiliary role called the log maintainer. In practice, there can be one or more agents acting as log maintainers. We do not require that any of these log maintainers are trusted and assume that the attacker controls them.

5.3 Overview

We present an overview of two protocols for Key Usage Detection (KUD). In the first, the participants are a single sender and a single receiver, assisted by a log maintainer. This situation is too simple to be useful, but serves to illustrate the core concepts. The second protocol is more involved; there are multiple senders and receivers, and each of them has multiple devices. This reflects a more realistic situation, and the multiple devices assist in the detection of attacks.

5.3.1 The basic protocol

Our solution involves three roles: senders, receivers, and a log maintainer. We assume all of these can be compromised. We assume a log maintainer is capable of receiving data and storing it in an append-only log.

During the bootstrapping phase, the receiver Robert obtains or generates a long-term signing and verification key pair (sk_R, vk_R), and the sender Sally obtains an authentic copy of vk_R. The log maintainer has a signing key sk_L, and Robert and Sally have an authentic copy of the corresponding verification key vk_L. How these keys are securely distributed is not the subject of this chapter; we assume it can be done through PKIs such as the DTKI presented in the previous chapter.

The log maintainer signs and publishes *digests* of the log, and provides supported proofs as presented in the previous chapter.

Sally and Robert track the digests issued by the log, all the time checking the proofs issued by the log that later digests represent extensions of earlier ones. Sally and Robert also periodically directly exchange the digests they know about, and request and check proofs of extension of those digests with respect to those they already have. Our assumption that the attacker cannot suppress all messages ensures that they are being presented with the same version of the log.

The transmission part of the basic KUD protocol then runs as follows (see Figure 5.2).

- To prepare for receiving a message, Robert's device creates an ephemeral encryption and decryption key pair (ek, dk), and certifies it with his long-term signing key sk_R. He publishes the certificate $\mathsf{Cert}_{sk_R}(R, ek)$ in the log. Publishing the certificate in the log assures Sally that it is a valid encryption key belonging to Robert.

- To send a message, Sally's device retrieves $\mathsf{Cert}_{sk_R}(R, ek)$ from the log along with a proof of its currency in the log. She encrypts the message with ek and sends it to Robert. Sally will not use a key whose certificate is not in the log.

- Robert's device receives the encrypted message and decrypts it.

Additionally, Robert's device periodically checks (where the period is determined by the parameter ζ) that all the keys ek' for which a certificate $\mathsf{Cert}_{sk_R}(R, ek')$ exists in the log were put there by him. If he finds entries in the log not corresponding to his actions, then he knows that his long term credentials have been disclosed and abused by an attacker.

The basic protocol assumes that Robert is online at the time that Sally wants to send a message. In the messaging application protocol below, we generalise this to work when Robert is offline.

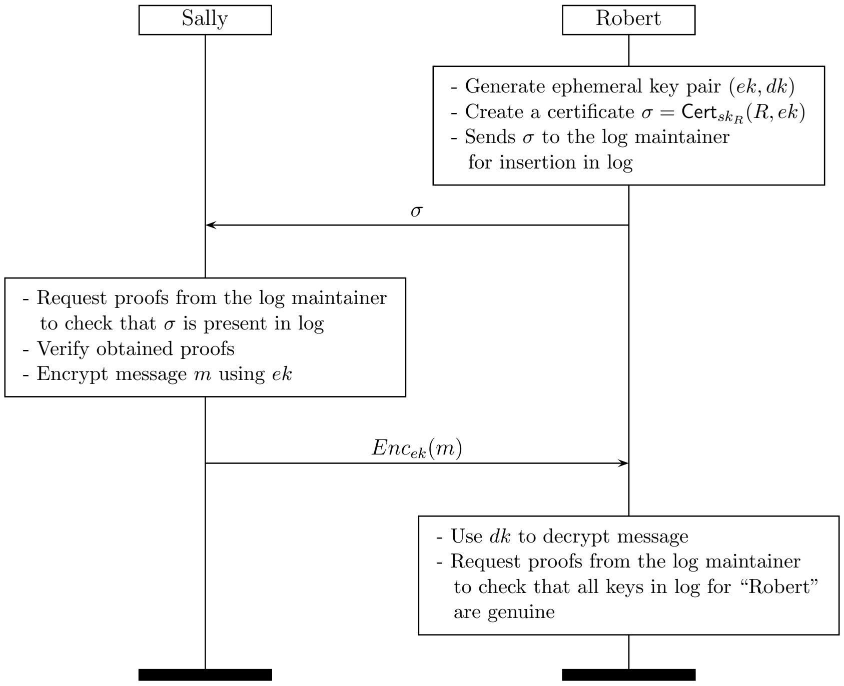

Sally
Robert
- Generate ephemeral key pair (ek, dk)
- Create a certificate $\sigma = \mathsf{Cert}_{sk_R}(R, ek)$
- Sends σ to the log maintainer for insertion in log
σ
- Request proofs from the log maintainer to check that σ is present in log
- Verify obtained proofs
- Encrypt message m using ek
$Enc_{ek}(m)$
- Use dk to decrypt message
- Request proofs from the log maintainer to check that all keys in log for "Robert" are genuine

Figure 5.2: The basic KUD protocol. Robert has a pair (sk_R, vk_R) of long term keys for signature signing and verification. He generates an ephemeral key pair (ek, dk) for encryption, creates the certificate $\sigma = \mathsf{Cert}_{sk_R}(R, ek)$ on ek, and sends the certificate to the log maintainer for insertion into the public log. Meanwhile, Robert also sends the certificate to Sally. After receiving σ, Sally requests from the log maintainer proofs that the certificate is present in the log. If the proof is valid, Sally sends a message m to Robert encrypted with ek. Robert requests proofs from the log maintainer to enable him to verify whether the log contains signatures that he did not generate.

Intuitively, our protocol design detects compromise attacks because an attacker in possession of Robert's long term key would have to leave evidence of its usage of the key in the log. We give examples of how this detection works in Section 5.3.3. We perform a formal analysis of our designs in Section 5.5.

5.3.2 Messaging application

The messaging application generalises the basic protocol, allowing the users to have multiple devices. Sally can choose any of her devices to send a message, and Robert is able to receive the message on all of his devices. Although this makes the protocol a bit more complicated, it also allows us to obtain a stronger security guarantee, because even if one of Robert's devices is in an untrustworthy state we are able to leverage security from the other ones.

As before, we assume a log, with the same capabilities mentioned above. We also assume that Robert and the log maintainer have long-term signing and verification key pairs (sk_R, vk_R) (sk_L, vk_L) respectively, and all parties have authentic copies of the verification keys they need.

The parameters δ, ϵ and ζ The protocol is parameterised by three values:

- δ is the period between the times at which device registration requests are processed. It is set by the log maintainer. We expect it to be typically one hour.

- ϵ is the period between the times at which key update requests are processed. We refer to such periods as "epochs". It is also set by the log maintainer, and is typically one day.

- ζ is the maximum lifetime of a key. It is set by the user. Different users can choose different values of ζ, subject to the constraint $\epsilon \leq \zeta$. We expect it to be about two or three days.

The messaging protocol has three main sub-protocols: enrolling, message transmission, and key updates. We describe these in turn.

Enrolling a device To enroll a device D_ℓ, Robert needs to install sk_R onto it. We assume that sk_R is derived from a passphrase that Robert types into D_ℓ. Next, D_ℓ needs to create a device key and publish its certificate in the log. More precisely:

- D_ℓ generates a new ephemeral encryption key pair (ek_ℓ, dk_ℓ) and sends the certificate $\mathsf{Cert}_{sk_R}(D_\ell, ek_\ell, t_\ell)$ to the log maintainer. Here, t_ℓ is the key creation time. The key will be used from the current time until the next epoch beginning, for the purpose of encrypting messages for Robert's device.

- After time δ, the log maintainer has inserted the certificate into the log and sends to D_ℓ the list of device certificates $\mathsf{Cert}_{sk_R}(D_i, ek_i, t_i)$ for Robert present in the log, together with a proof that the list is complete, and current in the log.

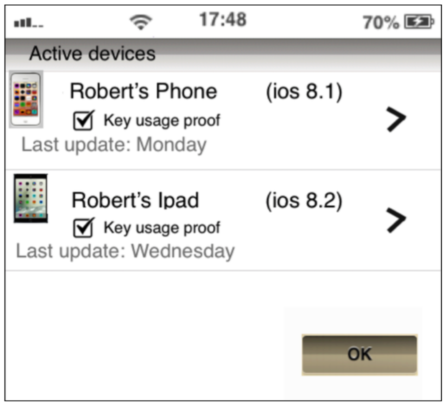


Figure 5.3: An example of envisaged GUI that presents the table (D_i, t_i) for $i = \{1, 2\}$ to Robert. Section 5.3.3 describes how Robert uses this information. The figure gives an impression of the kind of user interface we envisage. Usability is important and difficult to get right, and we need to work with HCI experts to design the interface fully.

- D_ℓ verifies the proof of currency for $\mathsf{Cert}_{sk_R}(D_\ell, ek_\ell, t_\ell)$. It displays the table (D_i, t_i) (for each i) to Robert, so he can check that the devices mentioned are indeed recently used. If Robert sees a device mentioned that he has not recently used, it is evidence of an attack (§ 5.3.4). Figure 5.3 presents an example of the envisaged GUI to show how the information is likely to be presented to Robert.

The device is now ready to be used. When Sally encrypts a message, her device will obtain all the current device keys for Robert from the log, and encrypt the messages with each of them.

Remark 5.1. *The method of displaying on a user's device the user's activities on other devices is well-known (for example, in Gmail, a user can click "last account activity" to see a table of the sessions open by other devices). A crucial difference in our protocol is that the displaying device can fully verify the veracity of the account activity provided by the untrusted log maintainer.*

Sending and receiving a message

- To send a message, Sally retrieves $\mathsf{Cert}_{sk_R}(D_i, ek_i, t_i)$ (for each available i) from the log along with proofs of currency. Her device encrypts a copy of the message with a fresh symmetric key k, and encrypts k with each received ek_i. It sends the encrypted message and together with the encrypted k to each of Robert's devices.

- Robert picks up any of his devices, receives the encrypted message, and decrypts it.

Updating the keys Whenever Robert invokes the messaging app on a device D_ℓ, the device checks to see if it is the first time it has run the app during that ϵ-long epoch. If so, it generates a new device key which will become the key for the following epoch. More precisely, on the first invocation during an epoch:

- D_ℓ requests and verifies proof of currency for all of the current epoch's device certificates $\mathsf{Cert}_{sk_R}(D_i, ek_i, t_i)$ for each available i. It verifies that ek_ℓ is indeed the one it created and sent the previous epoch; if this verification fails, it is evidence of an attack (§ 5.3.4). D_ℓ displays the table (D_i, t_i) (each i) to Robert, so he can check that the devices mentioned are indeed recently used. If Robert sees a device mentioned that he has not recently used, it is again evidence of an attack.

- D_ℓ next creates a new ephemeral encryption key pair (ek'_ℓ, dk'_ℓ) and sends the certificate $\mathsf{Cert}_{sk_R}(D_\ell, ek'_\ell, t_\ell)$ to the log maintainer. Here, t_ℓ is the key creation time.

- By the next epoch, the log maintainer has inserted into the log all the device keys thus received. If a device does not send a new key during an epoch, the old key is retained in subsequent epochs until a period ζ has elapsed. At that time, keys of devices that did not send new keys are revoked.

- When a new key becomes valid, D_ℓ securely removes the old key in the device.

In other words, devices change their key every epoch, and if they don't do so (because the application is not invoked during a particular epoch) then their key is reused for a certain period, and then revoked. In this last case, the device can't be used until it re-registers.

5.3.3 Detecting attacks: examples

To provide intuition on how our protocol allows users to detect attacks, we explain some potential attack detection scenarios. We will present our formal security analysis in Section 5.5.

Attacks from a third party Suppose one of Robert's devices, say his phone, is infected with malware, allowing an adversary to mis-use all the keys stored on the device. The adversary may decrypt messages encrypted with ephemeral keys, and may create new signed ephemeral keys by using the phone's long term key and inserting them into the log. While the phone remains under the control of the attacker, the decryption activity is not detected. However, the long-term key usage is detected if the user notices unexpected usage of phone using the GUI of Figure 5.3. The figure shows the GUI displayed on another device of Robert's. It informs him that (so far in the current epoch) the keys corresponding to his phone and his ipad have been active. If Robert has not used his phone in the epoch, then he learns that it has been compromised. The GUI also confirms that the proofs about the usage statement have been verified.

Suppose Robert regains control of his phone, through routine malware scanning and software patching. If the adversary continues to use the phone's long-term key to create ephemeral keys, the phone can detect this activity via the log, and report it to the user.

Attacks on or by the log maintainer Suppose the log maintainer is malicious or compromised. It may provide fake proofs, or provide no proofs at all. This is readily detected by client software. It may maintain the log incorrectly, either by not correctly recording signed ephemeral keys or by incorrectly recording fake ephemeral keys. These attacks are detected when the key owner requests a complete proof of presence.

A more interesting attack arises if the log maintainer shows different versions of the log to different users. A receiver may see a version in which his ephemeral keys are correctly recorded, while the sender sees a version in which attacker-owned keys are present instead. This would allow the attacker to play man-in-the-middle attacks, preventing the sender and receiver ever exchanging information about the log digests they have. Such an attack would be extremely hard in practice, because the attacker would have to persistently control *all* the messages sent between a sender and receiver. (For that reason, similarly to [LLK13, Rya14], we ruled out an attacker that completely controls the network.) As mentioned in the last chapter, gossip protocols have been introduced to further reduce the feasibility of this attack.

5.3.4 Responding to attacks

If Robert detects unexpected activity on a device, or some verification fails, this is evidence of an attack. Robert's response should be to fix the software on his devices. He should generate a new long-term key, in order to prevent attacks occurring (and being detected) due to the disclosure of his current long-term key. The corresponding public key can be distributed using the method used in the bootstrapping phase. Furthermore, he can inform Sally that some of her recent messages to him may have been compromised.

Robert can also detect failure when he verifies the actions of the log maintainer. His response is to change to a different provider.

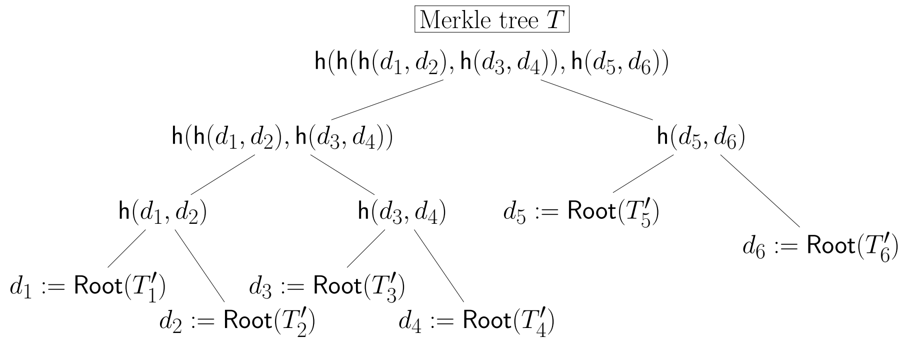
5.4 Detailed messaging implementation

This section presents the details of our proposal. We first present the log structure in Section 5.4.1. We then turn to describing the full protocol details in Section 5.4.2. The procedures that ensure that we detect malicious log maintainers are described in Section 5.4.3. After presenting the details, we discuss privacy concerns in Section 5.4.4.

5.4.1 Log structure

Similar to the public log employed in DTKI, the public log employed here is also organised as a tree of trees: the top-level tree is append-only, and its leaves are lexicographically ordered trees.

The ChronTree T (as shown in Figure 5.4) records the entire update history. Items in T are stored only in leaves and ordered chronologically, and each leaf is labelled by the root hash value of another Merkle tree T' (presented in Figure 5.5).



Merkle tree T
$\mathsf{h}(\mathsf{h}(\mathsf{h}(d_1, d_2), \mathsf{h}(d_3, d_4)), \mathsf{h}(d_5, d_6))$
$\mathsf{h}(\mathsf{h}(d_1, d_2), \mathsf{h}(d_3, d_4))$
$\mathsf{h}(d_5, d_6)$
$\mathsf{h}(d_1, d_2)$
$\mathsf{h}(d_3, d_4)$
$d_5 := \mathsf{Root}(T'_5)$
$d_6 := \mathsf{Root}(T'_6)$
$d_1 := \mathsf{Root}(T'_1)$
$d_3 := \mathsf{Root}(T'_3)$
$d_2 := \mathsf{Root}(T'_2)$
$d_4 := \mathsf{Root}(T'_4)$

Figure 5.4: An example of the log containing six updates $\{d_1, d_2, \ldots, d_6\}$. The log is maintained as an append-only Merkle tree T whose leaves are ordered chronologically.

Items in T' are also stored only in the leaves[2], but ordered according to user identity. Each leaf of T' is labelled by users' identity and a list of ephemeral certificates for different devices of the same user.

To recall how the proofs can be done with our log, we give some examples based on Figure 5.4 and 5.5. We will explain how to verify that the log is maintained correctly — i.e. the log maintainer only appends data in T, and items in every T' are ordered lexicographically — in §5.4.3.

Example of *proof of presence* To prove that data d'_2 for Bob is in T'_6 (see Figure 5.5), the log maintainer only needs to give the data needed to compute the label of parent node from d'_2 to the root of the tree.

$$\mathsf{PoP}(T'_6, d'_2) = [w, d'_1, h_{(3,4)}, h_{(5,7)}]$$

where $w = l \cdot l \cdot r$ is the path to d'_2, and l (resp. r) indicates the path to the left (resp. right) child. So, given d'_2, $\mathsf{Root}(T'_6)$, and the proof $\mathsf{PoP}(T'_6, d'_2)$, one can verify the proof by reconstructing the root value $h_T = \mathsf{h}(\mathsf{h}(\mathsf{h}(d'_1, d'_2), h_{(3,4)}), \mathsf{h}((5,7)))$. If $h_T = \mathsf{Root}(T'_6)$, then the proof is valid.

Example of *proof of currency* The proof of currency is the same as the proof of presence, but there is an extra constraint for the verifier to check, namely that the path to the root of the lexicographic tree only contains (an arbitrary number

[2]Note that the T' is implemented in a way that is slightly different to the LexTree in the previous chapter: in LexTree, data items are stored in both leaves and non-leaf nodes.

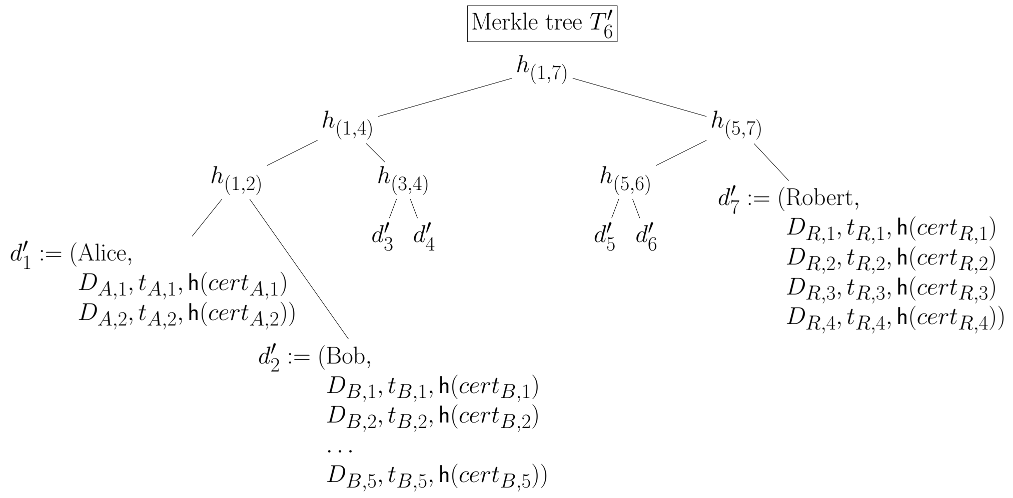

Merkle tree T'_6
$h_{(1,7)}$
$h_{(1,4)}$
$h_{(5,7)}$
$h_{(1,2)}$
$h_{(3,4)}$
$h_{(5,6)}$
$d'_7 :=$ (Robert, $D_{R,1}, t_{R,1}, \mathsf{h}(cert_{R,1})$ $D_{R,2}, t_{R,2}, \mathsf{h}(cert_{R,2})$ $D_{R,3}, t_{R,3}, \mathsf{h}(cert_{R,3})$ $D_{R,4}, t_{R,4}, \mathsf{h}(cert_{R,4})$)
d'_3 d'_4
d'_5 d'_6
$d'_1 :=$ (Alice, $D_{A,1}, t_{A,1}, \mathsf{h}(cert_{A,1})$ $D_{A,2}, t_{A,2}, \mathsf{h}(cert_{A,2})$)
$d'_2 :=$ (Bob, $D_{B,1}, t_{B,1}, \mathsf{h}(cert_{B,1})$ $D_{B,2}, t_{B,2}, \mathsf{h}(cert_{B,2})$ ... $D_{B,5}, t_{B,5}, \mathsf{h}(cert_{B,5})$)

Figure 5.5: An example of the data structure T' recording data in each update. Items in T' are ordered lexicographically. For all $a, b \in [1, 7]$, $h_{(a,b)}$ is the root hash value of a Merkle tree containing data from d'_a to d'_b. For example, $h_{(1,2)} = \mathsf{h}(d'_1, d'_2)$, and $h_{(1,7)} = \mathsf{h}(h_{(1,4)}, h_{(5,7)})$. Each leaf of T' is labelled by $(\mathsf{h}(ID),\ (D_j, t_j, \mathsf{h}(cert_j))_{j=1}^{n})$, such that $cert_j$ is a certificate on (D_j, ek_j, t_j) issued by ID, where D_j is the identity of the j^{th} device of ID, ek_j is an (ephemeral) public encryption key, and t_j is the issuing time.

of) "r".

Example of *proof of extension* To prove that the current version of the log represented by T (see Figure 5.4) is an extension of a previous version (T_{old}) containing four updates (i.e. $\mathsf{Root}(T_{old}) = \mathsf{h}(\mathsf{h}(d_1, d_2), \mathsf{h}(d_3, d_4))$ and $\mathsf{Size}(T_{old}) = 4$), the log maintainer gives $\mathsf{h}(d_5, d_6)$ as the proof. Given the two digests and this proof, the verifier can verify that T is extended from T_{old} by reconstructing $\mathsf{Root}(T)$.

Example of *Proof of absence* To prove that no certificates for user identity 'Bill' is included in T'_6 (see Figure 5.5), the log maintainer needs to prove that any node whose label containing Bill is absent from T'_6, by performing the following steps.

- Locate node A such that the user identity contained in its label is lexicographically the largest one smaller than Bill. In our example, the label of node A is d'_1 which contains user identity 'Alice'.

- Locate node B such that the user identity contained in its label is lexicographically the smallest one greater than Bill. In our example, the label of node B is d'_2 which contains user identity 'Bob'.

- Prove that d'_1 and d'_2 are present in T'_6, and they are siblings (so no node is placed in between of them). The former is proved by using proof of presence, and the latter one can be verified by checking the path to d'_1 and d'_2.

5.4.2 Messaging protocol details

We provide our main protocol with detailed message sequence here. It would help readers to see the exact exchanged messages. Also, it will be useful later to readers for understanding our formal model (presented in Appendix B) implemented in TAMARIN prover [MSCB13], which we used to give a rigorous formal machine-checked verification on the core security property of KUD (in the next section).

Enrolling a device (Figure 5.6)

We assume that all Robert's devices have shared his long-term signing key sk_R. To enrol a device D_ℓ, it generates a new ephemeral certificate, and publishes it in the log. In more detail, as presented in Figure 5.6:

- D_ℓ generates a new ephemeral key pair (dk_ℓ, ek_ℓ) for decryption and encryption, respectively. Then, D_ℓ issues a self-signed certificate $\mathsf{Cert}_{sk_R}(D_\ell, ek_\ell, t_\ell)$ on $(D_\ell, ek_\ell, t_\ell)$ by using sk_R, where t_ℓ is the key creation time; and sends the signed registration request
  $m_1 = (req_1, R, dg_{old}, \mathsf{Cert}_{sk_R}(D_\ell, ek_\ell, t_\ell))$ to the log, where req_1 is the request identity, R is the identity of Robert, and $dg_{old} = (\mathsf{Root}(T_{old}), \mathsf{Size}(T_{old}))$ is the digest of the log that Robert possibly has previously stored (it is likely to happen if Robert is re-enrolling his device D_ℓ).

- After the log maintainer receives the request, it verifies the signature and the certificate, and that t_ℓ is in the time interval of the current update epoch δ. If they are all valid, it stores the request, and issues a signed confirmation $\mathsf{sign}\{\mathsf{Root}(log), \mathsf{Size}(log), \mathsf{Cert}_{sk_R}(D_\ell, ek_\ell, t_\ell)\}_{sk_L}$, where log is organised as T, as explained in §5.4.1. If dg_{old} is provided, the log maintainer also generates a proof P of extension that the current log is extended from the log represented by dg_{old}, and sends the proof together with signed confirmation as the message m_2 to Robert.

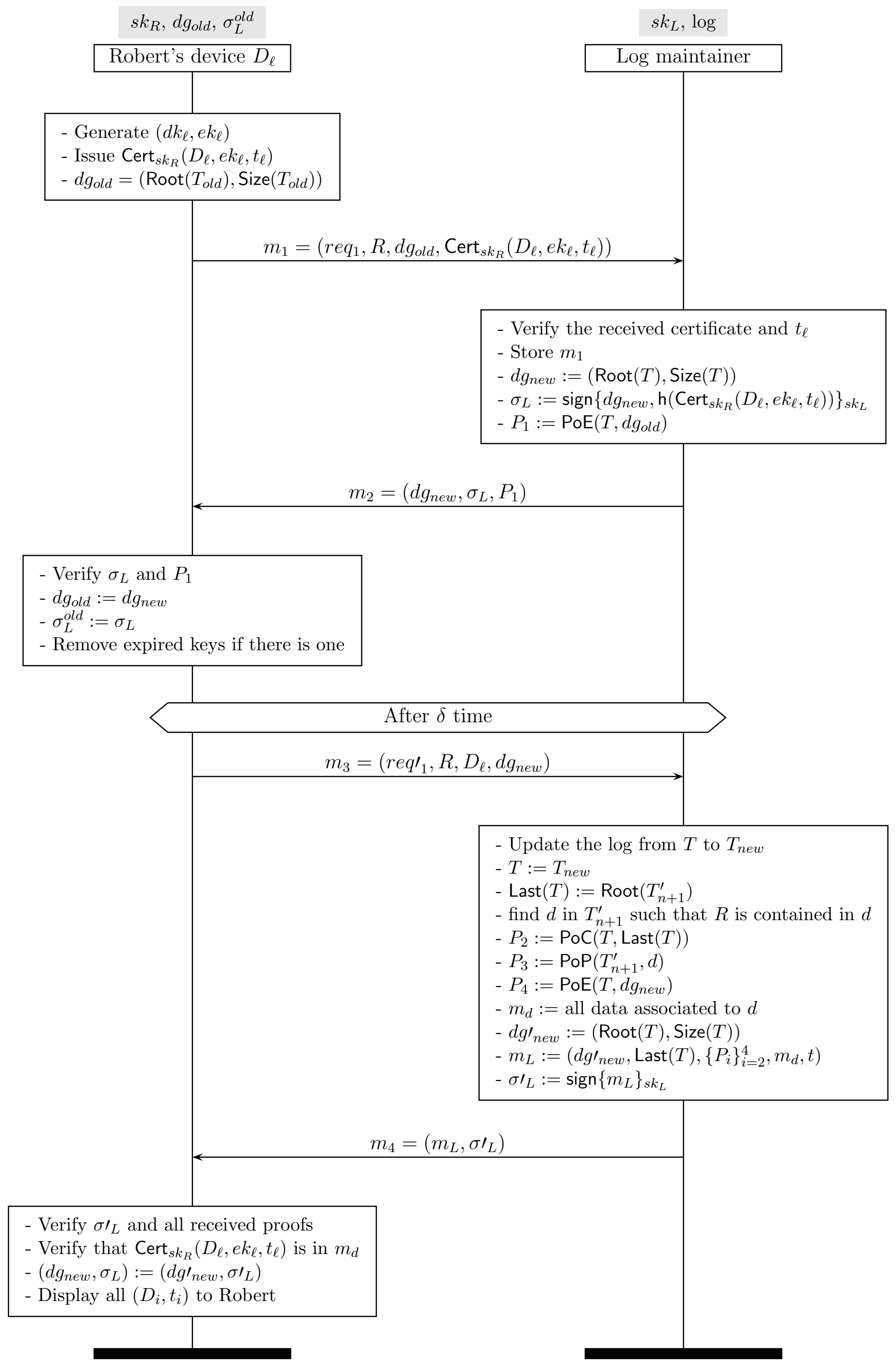

sk_R, dg_{old}, σ_L^{old}
Robert's device D_ℓ
sk_L, log
Log maintainer
- Generate (dk_ℓ, ek_ℓ)
- Issue $\mathsf{Cert}_{sk_R}(D_\ell, ek_\ell, t_\ell)$
- $dg_{old} = (\mathsf{Root}(T_{old}), \mathsf{Size}(T_{old}))$
$m_1 = (req_1, R, dg_{old}, \mathsf{Cert}_{sk_R}(D_\ell, ek_\ell, t_\ell))$
- Verify the received certificate and t_ℓ
- Store m_1
- $dg_{new} := (\mathsf{Root}(T), \mathsf{Size}(T))$
- $\sigma_L := \mathsf{sign}\{dg_{new}, \mathsf{h}(\mathsf{Cert}_{sk_R}(D_\ell, ek_\ell, t_\ell))\}_{sk_L}$
- $P_1 := \mathsf{PoE}(T, dg_{old})$
$m_2 = (dg_{new}, \sigma_L, P_1)$
- Verify σ_L and P_1
- $dg_{old} := dg_{new}$
- $\sigma_L^{old} := \sigma_L$
- Remove expired keys if there is one
After δ time
$m_3 = (req\prime_1, R, D_\ell, dg_{new})$
- Update the log from T to T_{new}
- $T := T_{new}$
- $\mathsf{Last}(T) := \mathsf{Root}(T'_{n+1})$
- find d in T'_{n+1} such that R is contained in d
- $P_2 := \mathsf{PoC}(T, \mathsf{Last}(T))$
- $P_3 := \mathsf{PoP}(T'_{n+1}, d)$
- $P_4 := \mathsf{PoE}(T, dg_{new})$
- $m_d :=$ all data associated to d
- $dg\prime_{new} := (\mathsf{Root}(T), \mathsf{Size}(T))$
- $m_L := (dg\prime_{new}, \mathsf{Last}(T), \{P_i\}_{i=2}^4, m_d, t)$
- $\sigma\prime_L := \mathsf{sign}\{m_L\}_{sk_L}$
$m_4 = (m_L, \sigma\prime_L)$
- Verify $\sigma\prime_L$ and all received proofs
- Verify that $\mathsf{Cert}_{sk_R}(D_\ell, ek_\ell, t_\ell)$ is in m_d
- $(dg_{new}, \sigma_L) := (dg\prime_{new}, \sigma\prime_L)$
- Display all (D_i, t_i) to Robert

Figure 5.6: The protocol for (re-)enrolling a device. In the protocol, if Robert is re-enrolling his device, then dg_{old} and σ_L^{old} are the previously stored digest and signature received from the log maintainer, respectively.

- D_ℓ verifies the received signature and proof, stores the new digest dg_{new} with signature σ_L, and sends the request m_3 containing a request identity req'_1, Robert and the device's identity (R, D_ℓ), and current observed digest to the log maintainer after δ time.

- After each period of length δ, the log maintainer updates the log according to the list of device enrollment requests received from its customers. The list of request should be in the form of

  $$(R_i, (\mathsf{Cert}_{sk_{R_i}}(D_{i,j}, ek_{i,j}, t_{i,j}))_{j=1}^{P})_{i=1}^{M}$$

  where R_i is the client identity, P is the number of devices that a client has requested to enrolling this update, and M is the total number of clients who has sent the enrollment request for this update.

  To update the log, the log maintainer retrieves the current T'_n such that $\mathsf{Root}(T'_n) = \mathsf{Last}(T)$, and creates T'_{n+1} by adding each request to the appropriate node of T'_n, where n is the size of the current log. It then extends T with a new rightmost node T'_{n+1}.

  In addition, the log maintainer proves that the list of certificates (including the ones in the enrollment request) for each participant R_i is complete, and current in the log. If R_i has previously observed a digest dg_{old} of the log, then log maintainer also generates a proof of extension that the current log is extended from the log represented by dg_{old}. To do so, the log maintainer locates the node labelled with d for R_i in T'_{n+1}, and generates:

  - $\mathsf{PoP}(T'_{n+1}, d)$ that d is present in T'_{n+1};
  - $\mathsf{PoC}(T, T'_{n+1})$ that the root hash value of T'_{n+1} is the label of the rightmost leaf in T; and
  - $\mathsf{PoE}(T, dg_{old})$ that the current log is extended from the log represented by dg_{old}.

  So R_i can verify that d — which contains a full list of certificates for his devices (including the newly enrolled ones) — is present in the latest update of the log.

- D_ℓ verifies the received proofs and signatures. Additionally, it displays the table (D_i, t_i) (for all $i \in [1, P]$) to Robert, so he can check that the devices mentioned are indeed recently used. If Robert sees a device mentioned that he has not recently used, it is evidence of an attack that an attacker who has used his long-term key without authorisation and has inserted a certificate for him.

The device is now ready to be used. A similar process will be used to un-register a device with the log maintainer.

Sending and receiving a message (Figure 5.7)

To send a message to Robert, Sally's device retrieves all the current device certificates for Robert from the log, and encrypts the messages with each of them. More precisely (as presented in Figure 5.7), to send a message:

- Sally sends request $m_1 = (req_2, R, r, dg_{old})$ to the log, where req_2[3] is the request identity, R is the identity of Robert, r is a random number, and where $dg_{old} = (\mathsf{Root}(T_{old}), \mathsf{Size}(T_{old}))$ is the digest of the log that Sally received in the last session.

- After receiving the request, the log maintainer locates the leaf whose label d contains R in the latest update T' (that is represented by the rightmost leaf of T), and generates the proof P_1 that $\mathsf{Root}(T')$ is current in T, proof P_2 that d is in T', and proof P_3 that the current log is an extension of the log that Sally has previously observed. It then sends m_2, which is the signed message ('CertResp', $dg_{new}, \mathsf{Last}(T), P_1, P_2, P_3, r, m_d, t$) to Sally, where 'CertResp' is a tag, $dg_{new} = (\mathsf{Root}(T), \mathsf{Size}(T))$, $m_d = (R,\ (D_j, t_j, ek_j, \mathsf{Cert}_j)_{j=1}^{P})$ is the data associated to d, and t is the time to identify the current epoch.

- After receiving the message from the log maintainer, Sally verifies if t is corresponds to the current epoch, and verifies the received signature, proofs, and certificates. If all verifications succeed, she replaces dg_{old} and σ_L^{old} by dg_{new} and σ_L, respectively, where σ_L is the signature from the log maintainer. Her device encrypts a copy of the message with a fresh symmetric key k, and encrypts k

[3]This request corresponds to the 'CertReq' in our TAMARIN code.

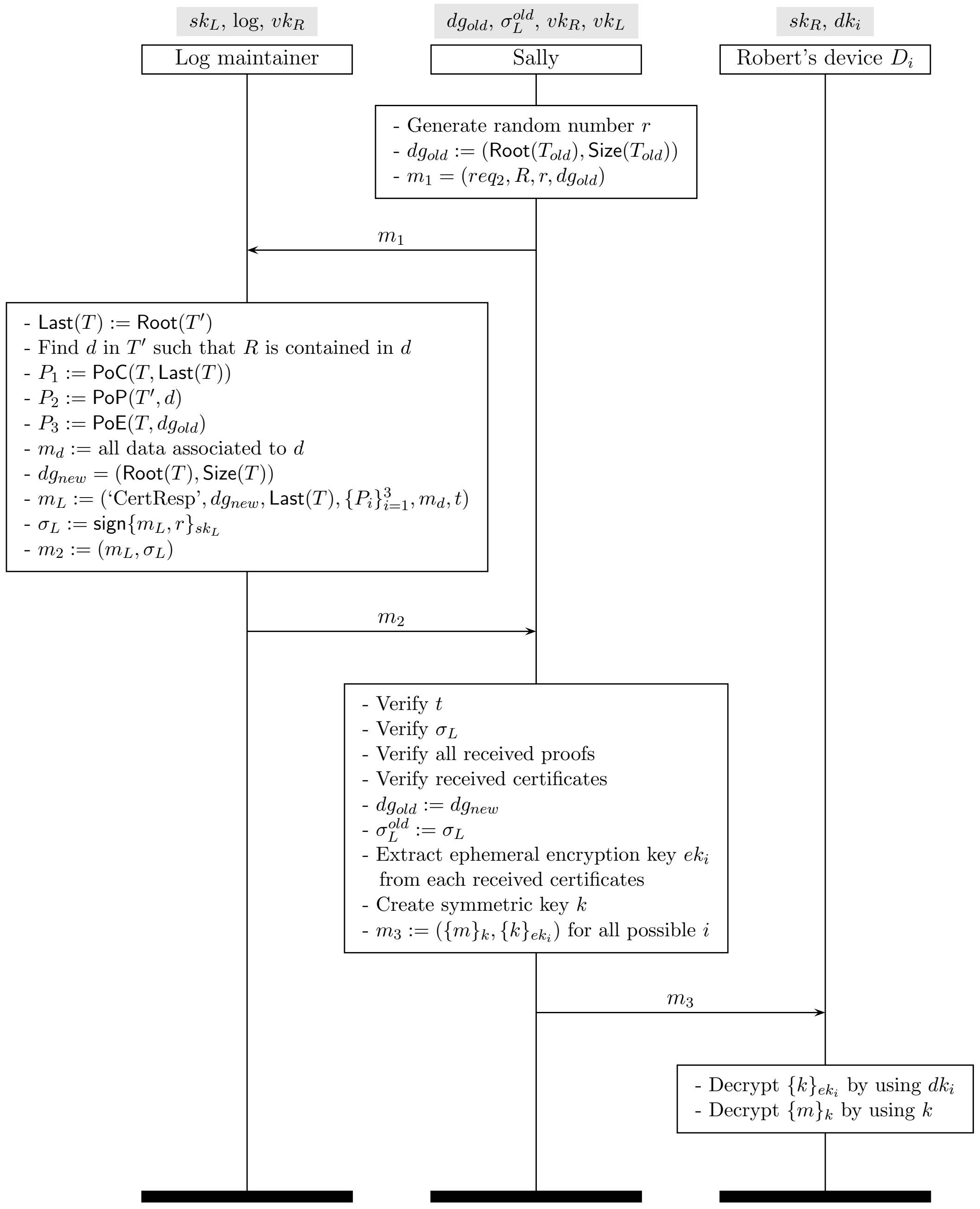

sk_L, log, vk_R
dg_{old}, σ_L^{old}, vk_R, vk_L
sk_R, dk_i
Log maintainer
Sally
Robert's device D_i
- Generate random number r
- $dg_{old} := (\mathsf{Root}(T_{old}), \mathsf{Size}(T_{old}))$
- $m_1 = (req_2, R, r, dg_{old})$
m_1
- $\mathsf{Last}(T) := \mathsf{Root}(T')$
- Find d in T' such that R is contained in d
- $P_1 := \mathsf{PoC}(T, \mathsf{Last}(T))$
- $P_2 := \mathsf{PoP}(T', d)$
- $P_3 := \mathsf{PoE}(T, dg_{old})$
- m_d := all data associated to d
- $dg_{new} = (\mathsf{Root}(T), \mathsf{Size}(T))$
- $m_L :=$ ('CertResp', $dg_{new}, \mathsf{Last}(T), \{P_i\}_{i=1}^{3}, m_d, t)$
- $\sigma_L := \mathsf{sign}\{m_L, r\}_{sk_L}$
- $m_2 := (m_L, \sigma_L)$
m_2
- Verify t
- Verify σ_L
- Verify all received proofs
- Verify received certificates
- $dg_{old} := dg_{new}$
- $\sigma_L^{old} := \sigma_L$
- Extract ephemeral encryption key ek_i from each received certificates
- Create symmetric key k
- $m_3 := (\{m\}_k, \{k\}_{ek_i})$ for all possible i
m_3
- Decrypt $\{k\}_{ek_i}$ by using dk_i
- Decrypt $\{m\}_k$ by using k

Figure 5.7: The protocol for sending and receiving a message. In which, σ_L^{old} is the signature received from the log maintainer in the last session. If any of the stated verification fails, the agent aborts the protocol.

with each received ek_i. It sends the encrypted message and together with the encrypted k to each of Robert's devices.

- Robert picks up any of his devices, receives the encrypted message, and decrypts it.

Note that in the protocol, if there is no certificate for Robert in the latest update, then a proof of absence that the identity of Robert is not in the latest update is provided to the user.

Remark 5.2. *The signed t is used to prevent attacks that replay a selected version of the log from the (compromised) log maintainer. Let x be the version of the log that Sally has previously observed, and z be the latest update. The replay attack is that the log maintainer picks and sends a version y of the log to Sally, such that $x < y < z$, and Robert's ephemeral key that is valid in the version y has been compromised by the attacker. In this case, if we do not have the signed t, then even with a gossip protocol, all verifications will succeed, because version y is also a genuine version of the log.*

Updating the keys (Figure 5.8)

Devices change their key every epoch w.r.t. ϵ, and if they don't do so (because the application is not invoked on a particular day), then their key will be reused for a certain period (e.g. a few more ϵ), and then will not be included in the log for the next further update epoch. In this last case, the device can't be used for receiving and reading messages until Robert uses the device again — it will re-register the device automatically. So, after Robert can use this device again in δ time (e.g. one hour). Note that if Robert has un-registered the device, then the device will not *automatically* re-register itself; and Robert has to re-register it *manually* in this case.

More precisely, whenever Robert invokes the messaging app on a device D_ℓ, the device checks to see if it is the first time it has run the app during that epoch w.r.t. ϵ. If so,

- D_ℓ creates a new ephemeral key pair (dk_ℓ, ek_ℓ), issues a certificate $\mathsf{Cert}_{sk_R}(D_\ell, ek_\ell, t_\ell)$, which will become the valid key in next epoch, where t_ℓ is the key creation

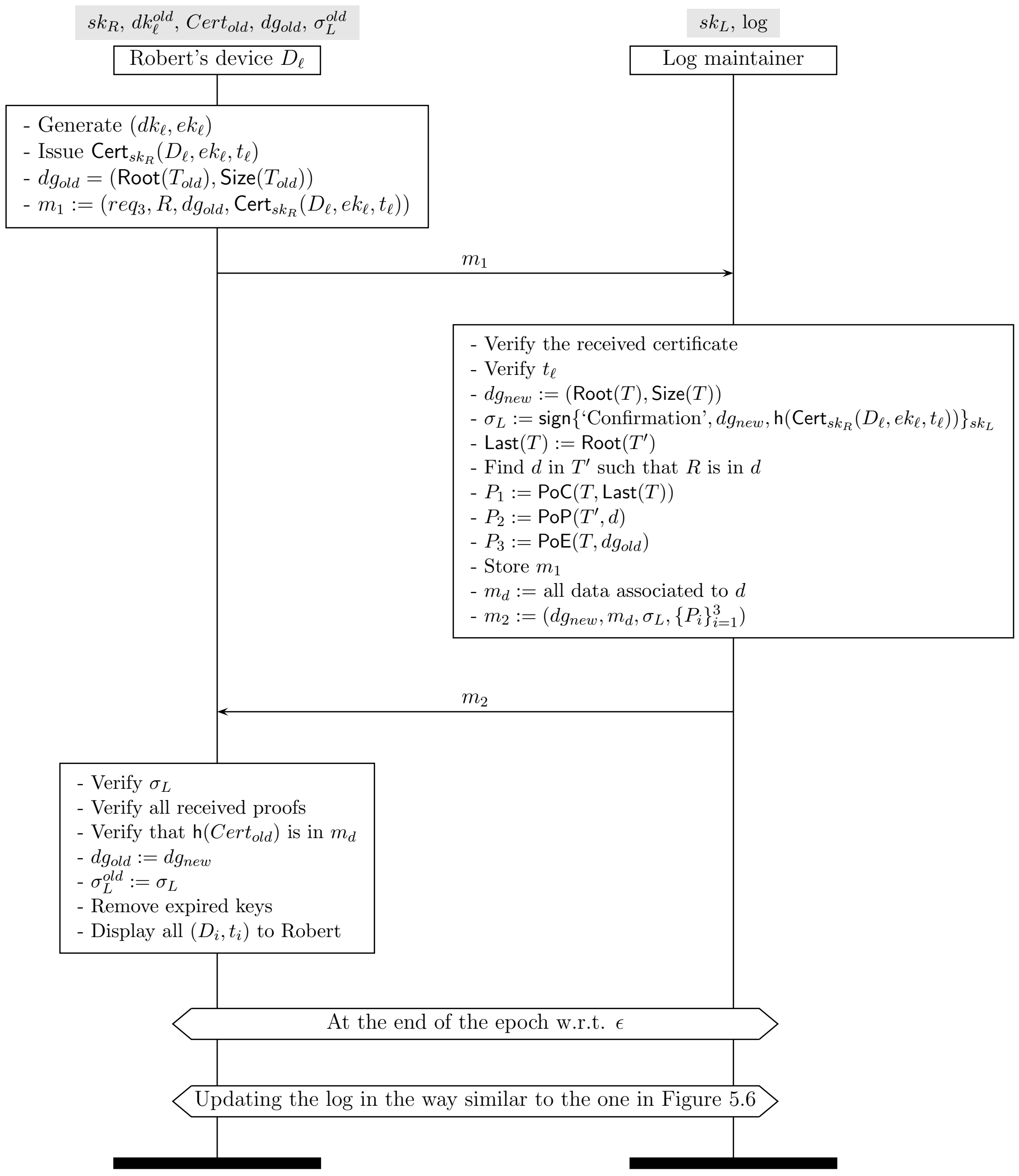

sk_R, dk_ℓ^{old}, $Cert_{old}$, dg_{old}, σ_L^{old}
sk_L, log
Robert's device D_ℓ
Log maintainer
- Generate (dk_ℓ, ek_ℓ)
- Issue $\mathsf{Cert}_{sk_R}(D_\ell, ek_\ell, t_\ell)$
- $dg_{old} = (\mathsf{Root}(T_{old}), \mathsf{Size}(T_{old}))$
- $m_1 := (req_3, R, dg_{old}, \mathsf{Cert}_{sk_R}(D_\ell, ek_\ell, t_\ell))$
m_1
- Verify the received certificate
- Verify t_ℓ
- $dg_{new} := (\mathsf{Root}(T), \mathsf{Size}(T))$
- $\sigma_L := \mathsf{sign}\{\text{'Confirmation'}, dg_{new}, \mathsf{h}(\mathsf{Cert}_{sk_R}(D_\ell, ek_\ell, t_\ell))\}_{sk_L}$
- $\mathsf{Last}(T) := \mathsf{Root}(T')$
- Find d in T' such that R is in d
- $P_1 := \mathsf{PoC}(T, \mathsf{Last}(T))$
- $P_2 := \mathsf{PoP}(T', d)$
- $P_3 := \mathsf{PoE}(T, dg_{old})$
- Store m_1
- $m_d :=$ all data associated to d
- $m_2 := (dg_{new}, m_d, \sigma_L, \{P_i\}_{i=1}^3)$
m_2
- Verify σ_L
- Verify all received proofs
- Verify that $\mathsf{h}(Cert_{old})$ is in m_d
- $dg_{old} := dg_{new}$
- $\sigma_L^{old} := \sigma_L$
- Remove expired keys
- Display all (D_i, t_i) to Robert
At the end of the epoch w.r.t. ϵ
Updating the log in the way similar to the one in Figure 5.6

Figure 5.8: The protocol for updating keys. In the protocol, dk_ℓ^{old} is the current valid ephemeral secret key, $Cert_{old}$ is the corresponding certificate, dg_{old} and σ_L^{old} are the digest and signature received from the log maintainer in the last session, respectively.

time. Then, he sends the signed request $m_1 = (req_3, R, dg_{old}, \mathsf{Cert}_{sk_R}(D_\ell, ek_\ell, t_\ell))$ to the log maintainer, where req_3[4] is the identity of update request, $dg_{old} = (\mathsf{Root}(T_{old}), \mathsf{Size}(T_{old}))$ is the digest of the log that he observed in the last session.

- After receiving the request, the log maintainer verifies the signature, time t_ℓ, and the received certificate. If they all valid, then it generates a commitment $\sigma_L = \mathsf{sign}\{\text{`Confirmation'}, dg_{new}, \mathsf{h}(\mathsf{Cert}_{sk_R}(D_\ell, ek_\ell, t_\ell))\}_{sk_L}$ that it will put the received new certificate in the log by the end of this epoch. The log maintainer locates the node d for Robert in the latest update of the log, and generates the proof P_1 that the root hash value of T' is the label of the rightmost leaf in T, proof P_2 that d is present in T', and the proof P_3 that T is an extension of the log that Robert has observed in the last session. Note that P_1 and P_2 together form the proof that d is the latest update for Robert in the log. The log maintainer sends the generated signature and proofs to D_ℓ.

- Upon receiving the response, D_ℓ verifies all signatures and proofs. Additionally, it verifies that the hashed certificate (contained in d) for D_ℓ in the latest update is indeed corresponding to the one it created and sent in the previous epoch. This verification ensures that no unauthorised request has been generated and recorded in the current log. (We will explain in the §5.4.3 that why we don't need to require D_ℓ to verify all history certificates for D_ℓ in the log are indeed generated by D_ℓ.) If all verifications succeed, D_ℓ removes any expired keys stored in D_ℓ, replaces the stored digest of the log with the new one, and displays the table (D_i, t_i) (for each possible i) to Robert, so he can check that the devices mentioned are indeed recently used. If Robert sees a device mentioned that he has not recently used, it is evidence of an attack.

- At the turn of the epoch, the log maintainer inserts all received update request into the log. Suppose in the current epoch, the log maintainer which maintains the log (represented by T of size n) has the tree T'_n containing

[4]This request corresponds to the 'UpdateReq' in our TAMARIN code.

$$\begin{aligned}
&(Alice,\ D_{A,1}, t_{A,1}, \mathsf{h}(cert_{A,1}) \\
&\qquad D_{A,2}, t_{A,2}, \mathsf{h}(cert_{A,2})), \\
&(Bob,\ D_{B,1}, t_{B,1}, \mathsf{h}(cert_{B,1}) \\
&\qquad D_{B,2}, t_{B,2}, \mathsf{h}(cert_{B,2}) \\
&\qquad \dots \\
&\qquad D_{B,5}, t_{B,5}, \mathsf{h}(cert_{B,5})), \\
&\dots\ \dots
\end{aligned}$$

and receives

$$(R_i, (\mathsf{Cert}_{sk_{R_i}}(D_{i,j}, ek_{i,j}, t_{i,j}))_{j=1}^{P'})_{i=1}^{M'}$$

for some identity R_i and certificates for its devices $D_{i,j}$, where P' is the number of a user's devices that has sent a key update request, and M' is the total number of clients who has sent the key update request in this epoch.

To up date the log, the log maintainer performs the following steps:

Step 1) creates a new tree T'_{n+1} by copying and pasting the entire T'_n;

Step 2) replaces the old certificates with the corresponding new ones in T'_{n+1};

Step 3) checks if any un-replaced certificate is older than ζ; if there is any, the log maintainer removes them from T'_{n+1};

Step 4) extends T with a new rightmost node $\mathsf{Root}(T'_{n+1})$.

Similar to the idea explained in §5.4.2, the log maintainer can provide the proof that the list of certificates (including the ones in the key update request) for R_i is complete, and current in the log; and the proof that the current log is an extension of the log that R_i has previously observed.

If a device has not updated ephemeral keys and has been excluded from the latest update by the log maintainer, then the device will automatically re-register itself when the owner has used the device again, so the device will be included in the log and be ready to receive and decrypt messages in δ time.

5.4.3 Crowd-sourced verification

Since we want to guarantee some security even when the log maintainer is not trusted, we need to monitor the log maintainer's behaviour to see if the log is maintained correctly. This can be easily verified if we introduce a trusted party to monitor the entire log. Alternately, similar to the approach used in the previous chapter, to avoid having a trusted party, we can use crowd-sourced verification by breaking the verification work into independent little pieces, and distribute each piece to different devices.

First, we need to verify that the log update history recorded in T is maintained in an append-only manner. This is achieved by verifying the proof of extension performed in each of above protocols, namely enrolling a device, updating the keys, and sending/receiving a message. Hence, there is no need for any additional verification.

Second, we need to verify that in each update T'_i, items are ordered lexicographically according to the user identity. It can be verified by asking each device to pick a random leaf in an update T'_i, and verify that the user identity recorded in its left (or right) neighbour leaf is lexicographically smaller (resp. greater) than the user identity in the picked leaf.

Third, in our protocol a device only checks its latest certificate in the log, instead of verifying all certificates recorded in the log. So, it cannot guarantee that no attacker-generated certificates have been previously included in the log. To detect such behaviour, we need to verify that the time of the key generation for the same device in different updates of the log is only going forward. To achieve this, each device picks a random leaf for a user in an update T_i, and verifies that either the record in an update is the same as the one in the previous update, or it is different and the time in the node for the same device of the user in the left (or right) neighbour update T_{i-1} (or T_{i+1}) is no greater (or no smaller) than the time in the picked leaf, respectively. Additionally, if the two times are equal, then the hash values of the certificates should also be equal. A missing associated record in a new update is evidence of misbehaviour. If no leaf for the user is included in the neighbour update, then a proof of absence that a node containing the user identity is not included in the update is provided.

Last, to ensure that the log maintainer did not show different logs to different users, users should exchange the digest of the log that they observed, for example, by using a gossip protocol.

5.4.4 Privacy considerations

The public log may cause some privacy concerns. For example, one may want to hide the user identities contained in a log, the total number of communications of a user, or the time distribution of a user's communications, etc.

In the above examples, to hide the user identity, the log maintainer can use a hash value of the signed user identity in the labels of leaves in each log update, rather than containing the user identity directly in the labels (see Figure 5.5). The signature scheme used should be deterministic and unforgeable, as suggested in [MBB$^+$14]. Hence, users that have the recipient's address can request the signed user identity from the log maintainer, and verify the log; but an attacker who has downloaded the entire log cannot recover the identity of users, based on the unforgeability of the chosen signature scheme. In this case, the nodes in each update tree T_i' will be ordered lexicographically according to the hash value of the signed user identity. In addition, users can also make the log to be only available to a fixed set of contacts. To hide the real number of communications associated to a given client of the log, the client can generate some noise — for example, the client can make 'spoof queries' to the log maintainer through an anonymous channel (e.g. Tor network).

There are many other possible solutions (e.g. server side access control). We do not detail them here as they are not the main focus of this thesis.

5.5 Security Analysis

We provide all input files required to understand and reproduce our security analysis at Appendix B. In particular, these include the complete KUD models.

5.5.1 Security properties

Our protocol achieves both classical security properties as well as novel ones. In a classical sense, Sally obtains the guarantee that if Robert's devices are not compromised, then the attacker will not learn the messages she sends.

The more interesting properties are achieved in the cases where Robert's devices get compromised. In this case, we cannot avoid that messages sent by Sally in the same epoch are also compromised. However, we prove that if any of Sally's messages from different epochs are compromised, then Robert will be able to detect this.

5.5.2 Formal analysis

We analyse the main security properties of the KUD protocol using the TAMARIN prover for a similar reason as explained in § 4.5.

Modeling aspects We used several abstractions during modeling. We model the Merkle hash trees as lists, similar to the verification of DTKI.

We model the protocol roles S (sender), R (receiver) and L (log maintainer) by a set of rewrite rules. Each rewrite rule typically models receiving a message, taking an appropriate action, and sending a response message. Our modeling approach is similar to most existing TAMARIN models. Our modeling of the roles directly corresponds to the protocol descriptions in the previous sections. TAMARIN provides built-in support for a Dolev-Yao style network attacker, i.e., one who is in full control of the network. We additionally specify rules that enable the attacker to compromise devices and learn their long and short-term secrets.

The final KUD model consists of 450 lines for the base model, and six main property specifications, examples of which we will give below.

Proof goals The first goal is a check for executability that ensures that our model allows for the successful transmission of a message. It is encoded in the following way.

```
lemma protocol_correctness:
 exists-trace
 " /* It is possible that */
   Ex d R skR dkR m #i.
     /* R received an encrypted message m on device d */
     MsgReceived(d, R, skR, dkR, m) @ #i
     /* without the adversary compromising any device. */
   & not (Ex d2 A ltk dkR #j.
            Compromise_Device(d2, A, ltk, dkR) @ #j) "
```

The property holds if the Tamarin model exhibits a behaviour in which one of R's devices received a message without the attacker compromising any device. This property mainly serves as a sanity check on the model. If it did not hold, it would mean our model does not model the normal (honest) message flow, which could indicate a flaw in the model. Tamarin automatically proves this property in a few seconds and generates the expected trace in the form of a graphical representation of the rule instantiations and the message flow.

We additionally proved several other sanity-checking properties to minimize the risk of modeling errors.

The second example goal is the core secrecy property with respect to a classical attacker, and expresses that unless the attacker compromises a key, he cannot learn any messages sent by Sally. Note that `K(m)` is a special event that denotes that the attacker knows m at this time.

```
lemma message_secrecy:
 "All R skR ekR m #i.
      /* If S sent a message m to R */
      ( MsgSent(R, skR, ekR, m) @ #i &
        /* without the adversary compromising any device */
        not (Ex #j d sk dkR.
                Compromise_Device(d, R, sk, dkR) @ #j)
      ) ==>
      ( /* then the adversary cannot know m */
        not ( Ex #j. K(m) @ #j)
      ) "
```

Tamarin also proves this property automatically.

The above result implies that if Robert receives a message that was sent by Sally, and the attacker did not compromise his device during the current epoch, then the attacker will not learn the message.

The next two properties encode the unique security guarantees provided by our protocol. If the attacker compromises Robert's device in an epoch, he can use the private ephemeral key to decrypt Sally's messages in that epoch. The first main property we prove is that if he uses the compromised long-term key of Robert to learn messages sent by Sally in other epochs, then he will be detected once Robert checks the log.

```
lemma detect_usage_S_sends:
 "All d skR dkR m #i1 #i2 #i3 detectionresult R k.
      /* If S sent to R an encrypted message m,
         where pk(dkR)=ekR */
      ( MsgSent(R, skR, pk(dkR), m) @ #i1 &
        /* and the adversary knows m */
        K(m) @ #i2 &
        /* and the ephemeral key used by the sender
           was not compromised, i.e., the compromise
           occurred in a different epoch
         */
        not (Ex #j sk .
                Compromise_Device(d, R, sk, dkR) @ #j ) &
        /* and Robert afterwards checks the log */
        CheckedLog(d, R, detectionresult, k ) @ #i3 &
        #i1 < #i3
      ) ==>
      ( /* then we detect a compromise */
        (detectionresult = 'bad')
      ) "
```

The property states that if Sally sends a message when Robert's device is not controlled by an attacker in the current epoch (but might have been compromised previously), and the attacker learns the message, then Robert detects the fact that his key was previously compromised when he next verifies the log.

The final property extends the previous for the messages that Robert actually receives from Sally, and shows that this also leads to detection of the key's abuse.

```
lemma detect_usage_R_receives:
 "All d skR dkR dkR2 m #i1 #i2 #i3 #i4 detectionresult R k.
      /* If S sent to R an encrypted message m,
         where pk(dkR)=ekR */
      ( MsgSent(R, skR, pk(dkR), m) @ #i1 &
        // /* and R receives it */
        MsgReceived(d, R, skR, dkR2, m) @ #i2 &
        /* and the adversary knows m */
        K(m) @ #i3 &
        /* and the ephemeral key used by the sender was
           not compromised, i.e., the compromise was in
           a different epoch then when m was sent.
         */
        not (Ex #j sk .
                Compromise_Device(d, R, sk, dkR) @ #j ) &
        /* and Robert afterwards checks the log */
        CheckedLog(d, R, detectionresult, k ) @ #i4 &
        #i2 < #i4
      ) ==>
```

```
( /* then we can detect a compromise */
  (detectionresult = 'bad')
) "
```

The last two properties encode the core security properties of our design. Both of them are proven automatically by the TAMARIN prover on a laptop within a few minutes.

Overall, the modeling effort was in the order of weeks, with several iterations to debug both the abstract model and the property specifications. The verification process helped us not only to prove, but also to refine the precise security properties of our protocol.

Our initial model and property specification could not be automatically verified by TAMARIN and we used the tool's interactive mode to determine the cause of non-termination. Ultimately, this enabled us to use TAMARIN's lightweight heuristics-influencing mechanism, which boils down to adding two lines of code per property, to guide the prover to find the proofs automatically and efficiently. This took several iterations and also revealed errors in earlier specifications, which made it clear that the complexity of the model required us to specify and prove several sanity checks.

5.6 Realization in practice

5.6.1 Estimating communication cost

To check if deployment might be feasible, we estimate the expected cost of our protocol design. As an example, we consider the following scenario. We assume that there are 10^9 users, each user has 5 devices, the log has been operating for 100 years, the log update period δ for registration request is 1 hour, and the log update epoch ϵ for certificate update is 1 day.

In this scenario, the size of T will be $100 \cdot 365 + 100 \cdot 365 \cdot 24 = 912500 < 2^{20}$, and the size of each T' is 10^9 which is less than 2^{30}. In addition, we assume that the size of a hash value is 256 bits (e.g. SHA256), the size of a signature is 64 Bytes (e.g. ECDSA), and the size of a certificate is 1.5 KB.

In addition, we assume that the size of a user (or device) identity is 12 Bytes, and time is in the 64-bit format, a random number is 28 bytes (recommended by

Table 5.1: The size of messages in different protocols. In which, size_P is the size of proofs in the corresponding message, and size_M is the maximum size of a message.

	Message	size_P	size_M
Enrolling a device			
	request	-	1.6 KB
	response	2.2 KB	2.5 KB
Total			4.1 KB
Fetching keys from log			
	request	-	78 B
	response	2.2 KB	9.9 KB
Total			10 KB
Updating the keys			
	request	-	1.5 KB
	response	2.2 KB	2.5 KB
Total			4 KB
crowd-sourced verification			
Total		5.3 KB	5.9 KB

TLS 1.2 [DR08]), each request identifier is 4 bits, and the size of a digest of a log is 300 bits.

The size of a proof of presence that some data is in T and is in T' will be at most 640 bytes and 960 bytes, respectively; the size of the proof that a version of the log is extended from a previous version is at most 640 bytes. We present the size of messages in the protocol in our example scenario in Table 5.1.

From Table 5.1 we can see that up to 5 KB data are needed to be transferred for both enrolling a device and updating keys. The protocol for fetching keys from the log is the most expensive one, as the sender has to download all certificates for different devices of the same users. In our example, the sender needs to download 5 certificates, the size of which is already 7.5 KB.

The results of our analysis indicate that the space cost of our system is acceptable.

5.6.2 Proof-of-concept log server prototype

To demonstrate the deployment of KUD in a real-world setting, we built a proof-of-concept prototype of the log server. We implemented a full log server implementation with interfaces, and client-side code for (a) adding users/devices, (b) rotating keys at the end of each epoch, and (c) sending messages. This involves all the operations to manipulate the log (consisting of a tree of trees), produce various proofs, and produce and verify the appropriate signatures. Anticipating a deployment on platforms such as Google's App Engine, we implemented our code in Python. We use basic caching mechanisms for previously computed results. On a quad-core 4 GHz Intel Core i7 with 32 GB of memory, we obtain the following times. The times are measured locally and therefore do not include network latency. Performing 100000 (1e05) enrollment requests from distinct users takes 1526 seconds, i.e., 15 milliseconds per request on average. When 100000 (1e05) users enroll 3 devices each, enrollment takes 1708 seconds, i.e., 5.7 milliseconds on average. The delay experienced by the user is therefore dominated by the network latency of transmitting 4.1 KB (Table 5.1), which is certainly less than a second.

When the tree contains 10000 (1e04) entries, the server produces 100000 (1e05) responses to message queries in 14.1 seconds, i.e., 0.14 milliseconds per message query. Updating a tree by simultaneously adding 10000 (1e04) entries takes about 1 second, which is mostly spent in creating the leaf data structures. Once again, the user's experience is mostly affected by the network latency, which is small because the data transferred is a few KB.

The memory usage when 100000 (1e05) users enroll one device is 410 MB (computed using "heapy" for the full process, not just reachable objects). If they enroll three devices each, memory usage increases to 900 MB.

Thus, even though our proof-of-concept implementation is not yet optimized for efficiency or storage, its performance already indicates our scheme is feasible.

5.7 Conclusion

We have presented a novel messaging protocol that offers security guarantees even when an attacker can access secret keys in a user's devices. In particular, (a) the protocol limits the impact of a compromise, since the attacker can only learn messages sent in the same epoch without being detected, and (b) if the attacker uses compromised long-term keys to impersonate users, then the protocol allows the participants to detect this, and therefore to take remedial action. Our protocol supports multiple devices per user, and the multiplicity of devices helps detect attacks by intuitive indicators to users about which (device) keys have recently been active.

The methods we introduce are not intended to replace existing methods used to keep keys safe. Existing technologies such as TPMs, smart-cards, and ARM TrustZone are all useful for securing keys. However, none of these technologies are completely secure. For example, even if hardware security is used, malware may be able to trigger usages of the key without having the ability to copy the key. Our methods can also detect such cases. Thus, KUD adds an additional layer of security that allows users to detect when other layers fail.

Part IV

Key compromise in secret distribution

CHAPTER 6

SELF-HEALING DISTRIBUTED STORAGE

6.1 Introduction

Cloud storage has been widely adopted to relieve the pain of maintaining dedicated hardware locally. For data owners, however, the confidentiality of outsourced data storage is a big concern. In particular, in the presence of system bugs, malware, and cyber attacks, the security of cloud servers cannot be guaranteed [Rya13, AKV15].

The data owner can encrypt the data before outsourcing it to the cloud. But this still leaves open the question of where to store the decryption keys. We would like a solution in which clients do not need to store any decryption keys. A client will authenticate itself to the server in order to request decryption.

One way to achieve such a solution is distributing the decryption keys of the encrypted data to many cloud servers. Secret sharing is a well-known technique to achieve this securely. In this case, an attacker would have to compromise a sufficient number of servers to break the security of the system. This makes the attacker's work more difficult. However, this solution is not good enough, as an attacker may be able to attack each server one-by-one over a long period. Eventually the attacker would be able to compromise sufficiently many servers and recover the decryption key.

To address this problem, we further develop the idea of storing secrets on several servers, with the aim of preventing gradual attacks over a long time. We assume that time is divided into *periods*. A server that is compromised in a time period may be repaired by its maintainers in the next time period [SCL^{+}15, YR15]. At the end of each time period, servers update all their secrets needed to recover the decryption key, before they start the next time period. The intuition is that, after a server has been repaired and has updated its secrets, the secrets learnt by an attacker in previous periods are rendered useless.

Unfortunately, the fact that the servers' secrets could be compromised and are changed periodically causes a problem for how a client should authenticate the servers. To solve this, we require a way for servers to have a public key for authentication that remains constant, even though the associated secrets held by the server are updated in each period. Achieving this requirement means that the clients do not need to be involved in (or even aware of) the server's regime for updating secrets.

In summary, we identify the following list of requirements:

1. The clients do not need to store any keys for performing decryption.
2. The system is secure even if all servers are compromised over a long time, provided that no more than a threshold number of servers are compromised in a given period.
3. The system provides a fixed public key for authentication, valid for all time periods.
4. The number of messages exchanged between the servers during the update period is independent of the number of data items stored.
5. As with any security protocol, there needs to be a well-defined adversary model, and a formal security proof.

This chapter presents a system based on bilinear pairings for distributed cloud storage, which we call "self-healing" cloud storage. The system satisfies the requirements mentioned above. One notable feature is that even though the service secrets change in each time period, the public key to be known by data owners remains constant. This feature could be used as a building block that allows us to tackle a more general server authentication issue, where the servers are compromisable cross time periods.

To formally prove the security guarantee of our self-healing scheme, we started our security verification first by using TAMARIN prover. However, we found (and reported) bugs in Tamarin prover when modelling bilinear paring based protocols. Since compared to DTKI and KUD protocols the scheme here has a heavy use of crypto, we proved this scheme by using classical game-based approach.

Before presenting this system in Section 6.3, we first formally define attacker model and security goals in Section 6.2. A rigorous formal security proof of the proposed system under the defined security model is presented in Section 6.4.

To the best of our knowledge, the proposed system is the first self-healing distributed storage, with a formal security model and a formal security proof.

6.2 Attacker model and security goal

This section first informal presents an attacker model and our security goal, in Section 6.2.1 and Section 6.2.2, respectively. It then defines the formal security model in Section 6.2.3.

We consider the scenario that an attacker wants to steal the sensitive data of users on cloud servers, by gradually breaking into servers of the system.

6.2.1 Attacker model

Suppose an attacker compromises a server. Then the attacker can fully control the server and has access to all its short-term and long-term secrets. Suppose sometime later, the maintainer of the server applies software patches and malware removal. Depending on the nature of the compromise, that action might restore the server into a secure state, or it might not.

As shown in Table 6.1, we use $\mathcal{S}_{PAC}$ to present the set of permanently attacker-controlled servers; $\mathcal{S}_{TAC}$ to present the set of temporarily attacker-controlled servers (as illustrated in Figure 6.3); and $\mathcal{S}_{Sec}$ to present the set of secure servers. For client side, we only consider two possibilities, namely the set $\mathcal{C}_{AC}$ of permanently attacker-controlled client and the set $\mathcal{C}_{Sec}$ of secure clients.

Figure 6.1 and Figure 6.2 show the possible transformation between different types of servers and clients, respectively. Generally, any secure server in $\mathcal{S}_{Sec}$ may become a temporarily attacker-controlled server; and any server in $\mathcal{S}_{TAC}$ may become a secure server; and any server in $\mathcal{S}_{Sec}$ or $\mathcal{S}_{TAC}$ may become a permanently attacker-controlled server. On the client side, we consider that any secure client may become

Table 6.1: The explanation on different types of participants.

Notation	Description
$\mathcal{S}_{PAC}$	The set of servers that are permanently controlled by attackers. Security actions, e.g. software patches and malware removal, can not succeed in restoring the servers to a secure state.
$\mathcal{S}_{TAC}$	The set of servers that are temporarily controlled by attackers. Security actions, e.g. software patches and malware removal, can succeed in restoring the servers to a secure state
$\mathcal{S}_{Sec}$	The set of servers that are currently secure.
$\mathcal{C}_{AC}$	The set of clients that are controlled by attackers.
$\mathcal{C}_{Sec}$	The set of clients that are currently secure.
$\mathcal{S}_{\text{Alice}}$	The set of servers selected by client Alice.
$\mathcal{S}$	The complete set of all servers, such that $\mathcal{S} = \mathcal{S}_{PAC} \cup \mathcal{S}_{TAC} \cup \mathcal{S}_{Sec}$
$\mathcal{C}$	The complete set of all clients, such that $\mathcal{C} = \mathcal{C}_{AC} \cup \mathcal{C}_{Sec}$
$\mathcal{P}$	The complete set of all participants, such that $\mathcal{P} = \mathcal{S} \cup \mathcal{C}$.

an attacker-controlled client.

6.2.2 Security goal

All servers update their secrets simultaneously at pre-determined times. We say T is an epoch if T starts from the beginning of the process for updating secrets, and ends at the beginning of the next process for updating secrets. Note that since we allow an adversary to corrupt servers at any moment during an epoch, if a server is corrupted during an update phase from epoch T to the next epoch T', we consider the attacker being able to obtain secrets in both the T-th and T'-th epochs.

Let $\mathcal{S}_{\text{Alice}}$ be set of servers selected by Alice. At a given epoch T, let $\mathcal{S}_{PAC}(T)$ be the number of permanently attacker-controlled servers in $\mathcal{S}_{\text{Alice}}$, and $\mathcal{S}_{TAC}(T)$ the number of temporarily attacker-controlled servers in $\mathcal{S}_{\text{Alice}}$. Our security goal is that an attacker cannot learn any secret of Alice, provided the total number of attacker-controlled servers in T and T' is less than the number of servers chosen by Alice, i.e. $\mathcal{S}_{PAC}(T') + \mathcal{S}_{TAC}(T) + \mathcal{S}_{TAC}(T') < |\mathcal{S}_{\text{Alice}}|$.

Remark 6.1. *Loosely speaking, it says that the system should be secure if the total number of compromised servers in two adjacent epochs is less than the number of*

$$\begin{tikzcd}
 & & \mathit{CreateSecureServer}() \arrow[d] \\
\mathcal{S}_{PAC} & \mathcal{S}_{TAC} \arrow[l, bend right, "\mathit{TakeOwnershipServer}()"'] \arrow[r, bend right, "\mathit{SecurityAction}()"'] & \mathcal{S}_{Sec} \arrow[l, bend right, "\mathit{CompromiseServer}()"'] \arrow[ll, bend left=40, "\mathit{TakeOwnershipServer}()"] \\
\mathit{CreateAttackerControlledServer}() \arrow[u] & &
\end{tikzcd}$$

Figure 6.1: A figure presenting the possible transformation between different types of servers, i.e. permanently attacker-controlled server $\mathcal{S}_{PAC}$; temporarily attacker-controlled server $\mathcal{S}_{TAC}$; and secure server $\mathcal{S}_{Sec}$. In our formal security model, these transformations can be achieved by using oracle queries as defined in Section 6.2.3.

$$\begin{tikzcd}
 & \mathit{CreateSecureClient}() \arrow[d] \\
\mathcal{C}_{AC} & \mathcal{C}_{Sec} \arrow[l, bend right, "\mathit{CompromiseClient}()"'] \\
\mathit{CreateAttackerControlledClient}() \arrow[u] &
\end{tikzcd}$$

Figure 6.2: A figure presenting the possible transformation between different states of clients, i.e. secure client C_{Sec} and attacker-controlled client C_{AC}. In our formal security model, these transformations can be achieved by using oracle queries as defined in Section 6.2.3.

servers chosen by Alice. Note that $\mathcal{S}_{PAC}(T)$ is the number of permanently compromised servers at epoch T, and these servers will be included in the set of permanently compromised servers in future epochs as well. So we have $\mathcal{S}_{PAC}(T) \leq \mathcal{S}_{PAC}(T')$. However, this is not true for $\mathcal{S}_{TAC}(\cdot)$.

6.2.3 Formal Model

We first define the scenario we are considering, i.e. attackers can periodically compromise cloud servers for storage. Then we formally define the ability of an attacker, and the security of a self-healing distributed cloud storage system.

Definition 6.1. *A **periodically compromised system environment (PCSE)** is an environment in which an attacker can periodically control honest participants of a protocol. It consists of*

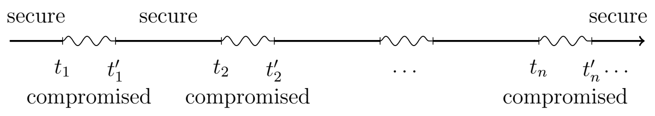


Figure 6.3: A timeline presenting a server's security state transformation between a temporarily attacker controlled server S_{TAC} and a secure server S_{Sec}. For all $i > 0$, we assume that the server is compromised in the time interval between t_i and t'_i, and is secure in the time interval between t'_i and t_{i+1}.

1. *Protocol* Π*: the underlying security protocol;*

2. *Security checking oracle SecurityCheck*(Π, S, t)*: given a server* $S \in \mathcal{S}$ *in protocol* Π *at time* t*, it outputs a boolean value* $V_{S,t}$ *to indicate if* S *is vulnerable at* t*. If* $V_{S,t} =$ *True, then an attacker is able to compromise* S*; otherwise,* S *is secure. This models the security status of a server.*

3. *Security action oracle SecurityAction*(Π, S, t)*: given a server* S *and time* t*, it outputs a strategy for* S *such that if* S *is a temporarily attacker-controlled server, i.e.* $S \in \mathcal{S}_{TAC}$*, and it executes the strategy at time* t*, then the server will become a secure server, i.e. SecurityCheck*$(\Pi, S, t') =$ *False, where* t' *is the time point right after* t*.*

We define our security model through a game with two participants, namely a challenger and a probabilistic polynomial time (PPT) adversary $\mathcal{A}$. The attacker's goal is to win the game that is initialised by the challenger. $\mathcal{A}$ is able to ask the following oracle queries.

1. $\mathcal{O}_1$: *Settings*(Π). By sending this query, the attacker is given all the public parameters of Π.

2. $\mathcal{O}_2$: *Execute*$(\Pi, \mathcal{P}')$. Upon receiving this query, the set of participants $\mathcal{P}' \subseteq \mathcal{P}$ executes protocol Π, if applicable. The exchanged messages will be recorded and sent to $\mathcal{A}$. This oracle query models an attacker's ability to eavesdrop communications between participants in Π.

3. $\mathcal{O}_3$: *CreateAttackerControlledClient*(Π, C). Upon receiving this query with a fresh identity C, the oracle creates an attacker-controlled client C in Π according to the attacker's choice. After this query has been made, we have

that $\mathcal{C}_{AC} := \mathcal{C}_{AC} \cup \{C\}$. We say an identity is "fresh" if and only if the identity is unique and has not been previously generated. This oracle models an attacker's ability to register a new client of its choice.

4. $\mathcal{O}_4$: $CreateAttackerControlledServer(\Pi, S)$. Upon receiving this query, the oracle creates a fresh server S, and sends the corresponding secret key and public key to the attacker. After this query has been made, we have that $\mathcal{S}_{PAC} := \mathcal{S}_{PAC} \cup \{S\}$. This oracle allows $\mathcal{A}$ to adaptively register permanently attacker-controlled servers of its choice.

5. $\mathcal{O}_5$: $CreateSecureClient(\Pi, C)$. Upon receiving this query, the oracle creates a fresh client C in Π. After this query has been made, we have that $\mathcal{C}_{Sec} := \mathcal{C}_{Sec} \cup \{C\}$. This oracle query allows an attacker to introduce more clients, which are initially secure.

6. $\mathcal{O}_6$: $CreateSecureServer(\Pi, S)$. Upon receiving this query, the oracle creates a fresh server S in Π. After this query has been made, we have that $\mathcal{S}_{Sec} := \mathcal{S}_{Sec} \cup \{S\}$. This oracle query allows an attacker to introduce more servers, which are initially secure.

7. $\mathcal{O}_7$: $CompromiseClient(\Pi, C)$. Upon receiving this query for some $C \in \mathcal{C}_{Sec}$ in Π, the oracle forwards all corresponding secrets of C to $\mathcal{A}$. From now on the attacker controls C so that $C \in \mathcal{C}_{AC}$ and $C \notin C_{Sec}$ after this query has been made. This oracle query allows $\mathcal{A}$ to adaptively and permanently compromise a client of its choice.

8. $\mathcal{O}_8$: $TakeOwnershipServer(\Pi, S)$. Upon receiving this query for some $S \in \mathcal{S}_{Sec}$ or $S \in \mathcal{S}_{TAC}$ in Π, the oracle forwards all corresponding secrets of S to $\mathcal{A}$, and from now on the attacker controls S. So, S is moved from its current set in to $\mathcal{S}_{PAC}$ after this query has been made. This oracle query allows $\mathcal{A}$ to adaptively and permanently compromise a server of its choice.

9. $\mathcal{O}_9$: $CompromiseServer(\Pi, S)$. Upon receiving this query, the oracle outputs all secrets of $S \in \mathcal{S}_{Sec}$ in Π. We have $S \notin S_{Sec}$ and $S \in S_{TAC}$ after this query has been made. This oracle query models $\mathcal{A}$'s ability to adaptively compromise a temporarily attacker-controlled server of its choice.

10. $\mathcal{O}_{10}$: $Dec(\Pi, Enc(M, PK_{\mathcal{S}_C}), C)$. Upon receiving this query for some client $C \in \mathcal{C}_{Sec}$ with secret M, the set $\mathcal{S}_C$ of servers collectively executes the decryption protocol to decrypt the encrypted message $Enc(M, PK_{\mathcal{S}_C})$, and sends the decryption result M to the attacker, where $PK_{\mathcal{S}_C}$ is the common public key of the set $\mathcal{S}_C$ servers selected by C for encryption/decryption.

We now consider the distributed cloud storage scenario. If a powerful attacker $\mathcal{A}$ can fully control a data owner's device when the device is creating or recovering a secret s, then $\mathcal{A}$ can easily learn s. As mentioned before, we do not consider this case, as there is nothing we can do and it is not interesting. To focus on the more interesting cases, we only consider that $\mathcal{A}$ cannot learn s by compromising the data owner's device during the secret creation or recovery time.

Definition 6.2. *A self-healing distributed storage protocol* Π *is* (k, n)-secure *if the advantage* $Adv_{\mathcal{A},n,k}(\lambda) = |Pr[b = b'] - \frac{1}{2}|$ *of* $\mathcal{A}$ *to win the following game, denoted* Game-PCSE*, is negligible in the security parameter* λ*.*

Game-PCSE:

- *Setup*(Π, λ)*. The challenger sets up protocol* Π *according to the security parameter* λ*. Initially,* $\mathcal{S} = \mathcal{C} = \emptyset$*.*

- *Query phase. The attacker can ask a polynomially bounded number of oracle queries* $\mathcal{O}_i$ *for* $i \in \{1, 2, \ldots, 10\}$*. Let* j_4*,* j_8*, and* j_9 *be counters counting the total number of* $\mathcal{O}_4$*,* $\mathcal{O}_8$*, and* $\mathcal{O}_9$ *queries asked by the attacker, respectively. We have that* $j_4 + j_8 + j_9 < k$*.*

- *Security action phase. The challenger makes security checking oracle queries on all servers, and then makes security action oracle queries on the servers that are temporarily controlled by the attacker. At the end of this phase, the counter* j_9 *will be reset to "0".*

- *The query phase and the security action phase are repeated a polynomially bounded number of times.*

- *Challenge*(C_b, C)*. The attacker selects a target client* C *who has not been asked through* $\mathcal{O}_i$ *for* $i \in \{3, 7\}$*, i.e.* $C \in \mathcal{C}_{Sec}$*; and selects two messages* M_0

and M_1, s.t. $|M_0| = |M_1|$. The attacker then sends them to the challenger. The challenger tosses a coin. Let $b \in \{0, 1\}$ be the result of the coin toss. The challenger then encrypts M_b according to Π, and sends the ciphertext $C_b = Enc(M_b, PK_{\mathcal{S}})$ back to the attacker.

- *The query phase and the security action phase are repeated a polynomially bounded number of times. Additionally, we require that the target client C cannot be asked through $\mathcal{O}_3$ and $\mathcal{O}_7$, and $Dec(\Pi, C_b, C)$ cannot be queried through $\mathcal{O}_{10}$.*

- *Guess(b). The attacker makes a guess b' of the value of b, and outputs b'. The attacker wins if $b = b'$.*

Remark 6.2. *In the game defined above, the execution of a query phase followed by a security action phase simulates an epoch of the protocol.*

Remark 6.3. *In a (k, n)-threshold cryptosystem, an attacker can break the security if the attacker is able to compromise k secrets/parties during the lifetime of the system. However, in the above defined (k, n)-secure system in the PCSE, an attacker cannot break the security even if the attacker can compromise all n parties in the lifetime of the system, provided at any time point t between two updates, at most $k - 1$ parties are compromised by the attacker.*

6.3 Our solution

We present our solution, first in a non-threshold form (i.e., we stipulate that the minimum number k of servers needed for performing decryption is equal to n, the total number of servers). Later, in section 6.5.1, we generalise it to a threshold-based solution where we allow one to choose $k < n$.

6.3.1 Basic idea

Alice selects a set of servers, and encrypts her secret by using the combined public key of the selected servers.

Setup
$S_A : (a, g^a), \quad S_B : (b, g^b), \quad S_C : (c, g^c), \quad PK = g^{abc}$
Zero-th Update
$S_A : (a_0, g^{a_0}), \quad S_B : (b_0, g^{b_0}), \quad S_C : (c_0, g^{c_0}), \quad H_0 = g^{(a_0 b_0 c_0/abc)}$
First Update
$S_A : (a_1, g^{a_1}), \quad S_B : (b_1, g^{b_1}), \quad S_C : (c_1, g^{c_1}), \quad H_1 = g^{(a_1 b_1 c_1/abc)}$
The j-th Update
$S_A : (a_j, g^{a_j}), \quad S_B : (b_j, g^{b_j}), \quad S_C : (c_j, g^{c_j}), \quad H_j = g^{(a_j b_j c_j/abc)}$
Encryption at any time
$C = (\alpha = g^{abck}, \beta = sZ^k)$, for some secret s and random number k
Decryption at the j-th epoch
Compute $\gamma = e(\alpha, H_j)$, then decrypt (β, γ) by using (a_j, b_j, c_j)

Figure 6.4: The data associated with the servers S_A, S_B and S_C at different stages of the protocol, and the encryption and decryption computations.

Time is divided into epochs. At the end of each epoch, the servers execute a protocol during which they generate new decryption keys and destroy the old ones. If a server is compromised in an epoch, the attacker obtains all its (shares of) decryption keys. However, the protocol ensures that decryption keys from a server in one epoch cannot be used together with decryption keys from a server in a different epoch. Each change of epoch renders useless the decryption keys obtained by the attacker in previous epochs. Thus, to decrypt the secret, and attacker would have to compromise a threshold number of servers *within the same* epoch.

6.3.2 Abstract construction

We explain the protocol with three servers, S_A, S_B, and S_C. Let G_1, G_2 be two cyclic groups of a sufficiently large prime order p, such that $|p| = \lambda$, with a bilinear map $e : G_1^2 \to G_2$, and $g \in G_1$ is a generator and $Z = e(g, g) \in G_2$ (as defined in Chapter 2). The data associated with the servers at different stages of the protocol are presented in Figure 6.4.

Setup and zero-th epoch. S_A generates a private key $a \in \mathbb{Z}_p$, and a public key g^a. Similarly, S_B and S_C generate (b, g^b) and (c, g^c), such that $b, c \in \mathbb{Z}_p$. Then S_A, S_B, S_C collectively compute and publish their joint public key g^{abc}.

Next, S_A generates a new key $a_0 \in \mathbb{Z}_p$ and public key g^{a_0}, and similarly S_B and S_C generate (b_0, g^{b_0}) and (c_0, g^{c_0}), such that $b_0, c_0 \in \mathbb{Z}_p$. Then S_A, S_B, S_C collectively compute and publish helper data $H_0 = g^{(a_0/a)\cdot(b_0/b)\cdot(c_0/c)}$ with proofs that they have correctly performed the computation. They destroy the secrets a, b, c.

At the end of the $(j-1)$-th epoch. The servers replace their decryption keys a_{j-1}, b_{j-1}, and c_{j-1} with new ones a_j, b_j, and c_j. Then S_A, S_B, S_C collectively compute and publish helper data $H_j = g^{(a_j/a)\cdot(b_j/b)\cdot(c_j/c)}$ with proofs that they have correctly performed the computation. The values a, b, c are not required to compute H_j. They destroy the secrets a_{j-1}, b_{j-1}, c_{j-1}.

Encryption of secret s. At any time during the server lifecycle (i.e. any epoch j), a client Alice can encrypt her data with the (unchanging) public key g^{abc}. To encrypt secret s, she selects a new random k and computes $C = (\alpha = g^{abck}, \beta = sZ^k)$.

Decryption of ciphertext (α, β) at the j-th epoch.

After authenticating client Alice's request for decryption, the servers can collectively decrypt a ciphertext (α, β) during any epoch. To decrypt (α, β), the servers compute $\gamma = e(\alpha, H_j) = Z^{a_j b_j c_j k}$. Then the servers use their secrets a_j, b_j, and c_j to collectively compute Z^k, and then they can recover the secret s from $\beta = sZ^k$. Note that duing this decryption process, Alice should apply masking to the γ to prevent servers from learning the plaintext. More details are presented in the next section.

Remark 6.4. *Note that the public key used by clients for encryption remains constant regardless of the secret updates on the server side (requirement 3 in Section 6.1). Also, the update procedure is independent of the number of stored ciphertexts (requirement 4). That is because in the update phase the servers need only collectively compute the helper data. The ciphertext (α, β) of each data item remains unchanged.*

6.3.3 Detailed construction

Initialisation *Setup.* Let $S_1, S_2, \ldots, S_n$ be the servers selected by Alice. S_i creates setting-up key pair (s_i, g^{s_i}), for some $s_i \in \mathbb{Z}_p$, respectively. They also compute a common public key $PK = g^{\prod s_i}$, which is available to the client in an authentic matter, e.g., via a certificate. This key can be established as follows:

- Each S_i computes and publishes $P_i = g^{s_i}$.

- S_2 computes $PK_{12} = (P_1)^{s_2}$. This computation can be verified by checking $e(PK_{12}, g) = e(P_1, P_2)$.
- S_i computes $PK_{1\dots i} = (PK_{1\dots(i-1)})^{s_i}$, which again can be verified by checking $e(PK_{1\dots i}, g) = e(PK_{1\dots(i-1)}, P_i)$.

Now, each S_i has a secret key s_i and a common public key PK.

Zero-th epoch. This epoch is to generate the first decryption keys. Each S_i chooses another secret key s_{i0}, computes $u_{i0} = s_{i0}/s_i$, computes and publishes $P_{i0} = g^{s_{i0}}$, $P'_{i0} = g^{u_{i0}}$ and deletes s_i. The correctness of these values can be checked as $e(P'_{i0}, P_i) = e(P_{i0}, g)$.

By using u_{i0}, S_i works with other servers to get $H_0 = g^{\prod s_{i0}/s_i}$ in the same way as computing PK, and then deletes u_{i0}.

At the end of the initialisation, S_i only holds s_{i0} at secret. This value can be used for decryption (if needed) and is used for the next decryption key update. In addition, S_i also holds two public values, namely a helper data H_0 and a common public key PK.

Note that the common public key is used for data encryption by the clients. This implies that the clients do not have to follow the server key updating processes, and they will keep using the key PK for a reasonably long time.

Updating the decryption keys The decryption key update process is similar to the computation of the first decryption keys presented in the previous phase. At the end of the $(j-1)$-th epoch for some $j \geq 1$, the servers replace their decryption keys $s_{i(j-1)}$ with new ones, s_{ij}. This is achieved as follows.

- With the input $s_{i(j-1)}$, S_i chooses s_{ij}, computes $u_{ij} = s_{ij}/s_{i(j-1)}$, computes and publishes $P_{ij} = g^{s_{ij}}$ and $P'_{ij} = g^{u_{ij}}$, and deletes $s_{i(j-1)}$. The correctness of these values can be checked as $e(P'_{ij}, P_{i(j-1)}) = e(P_{ij}, g)$.
- By using u_{ij}, S_i works with other servers to get $H_j = H_{j-1}^{\prod s_{ij}/s_{i(j-1)}} = g^{\prod s_{ij}/s_i}$ and then deletes u_{ij}. The correctness of these values should also be verified in the same way as computing PK.

At the end of $(j-1)$-th update, S_i only holds s_{ij}. This value is used for both decryption and update in the (j)-th epoch.

Encryption To encrypt a secret s, Alice selects a new random k, and computes $PK^k = g^{k \cdot \prod s_i}$ and sZ^k. Alice sends $(\alpha = PK^k, \beta = sZ^k)$ to each server.

Servers only accept (α, β) as some encrypted data from Alice if a valid proof of knowledge of s (or k) is provided. This is used to prevent replay attacks in which an attacker who has observed (α, β) sets up an account with the servers, and provides (α, β) as the attacker's encrypted data, then requests servers to decrypt it for the attacker. Any secure zero knowledge proof of knowledge (ZKPK) can be used. For example, the proof can be a Schnorr ZKPK of k, where the prover knows k and the verifier knows PK^k. If the prover shows knowledge of k, this implies that she also knows s.

At the end, Alice destroys s and k after all servers are convinced and accepted the ciphertext.

Decryption An abstract protocol for decrypting an encrypted secret is presented in Figure 6.5.

In more detail, in the j-th epoch for some $j \geq 0$, Alice sends a request to a selected server for retrieving the encrypted data. After successfully authenticating Alice, the server calculates $\gamma = e(\alpha, H_j) = Z^{k \cdot \prod s_{ij}}$, and sends (β, γ) to Alice.

Alice selects a new random $k' \in \mathbb{Z}_p$, sends $Z^{k'}$ to each of the servers as her commitment on k', computes $\gamma^{k'} = Z^{kk' \cdot \prod s_{ij}}$, and asks each server to remove its s_{ij} from the exponent. The final output should be $Z^{kk'}$. She then can recover Z^k by computing $\left(Z^{kk'}\right)^{1/k'}$, and thus be able to decrypt sZ^k.

Before a server decrypts some message requested by a user, the server expects a proof that the requested decryption is indeed a step to help the user to recover a key that the user actually owns, i.e. to prove that

$$(Z, Z^{k'}, Z^{k \cdot \prod s_{ij}}, Z^{kk' \cdot \prod s_{ij}})$$

is a DDH tuple. This can be done by using classic non-interactive ZKPK schemes, for proving that (g, g^x, g^y, g^{xy}) is a DDH tuple (e.g. Chaum-Pedersen protocol [CP92],

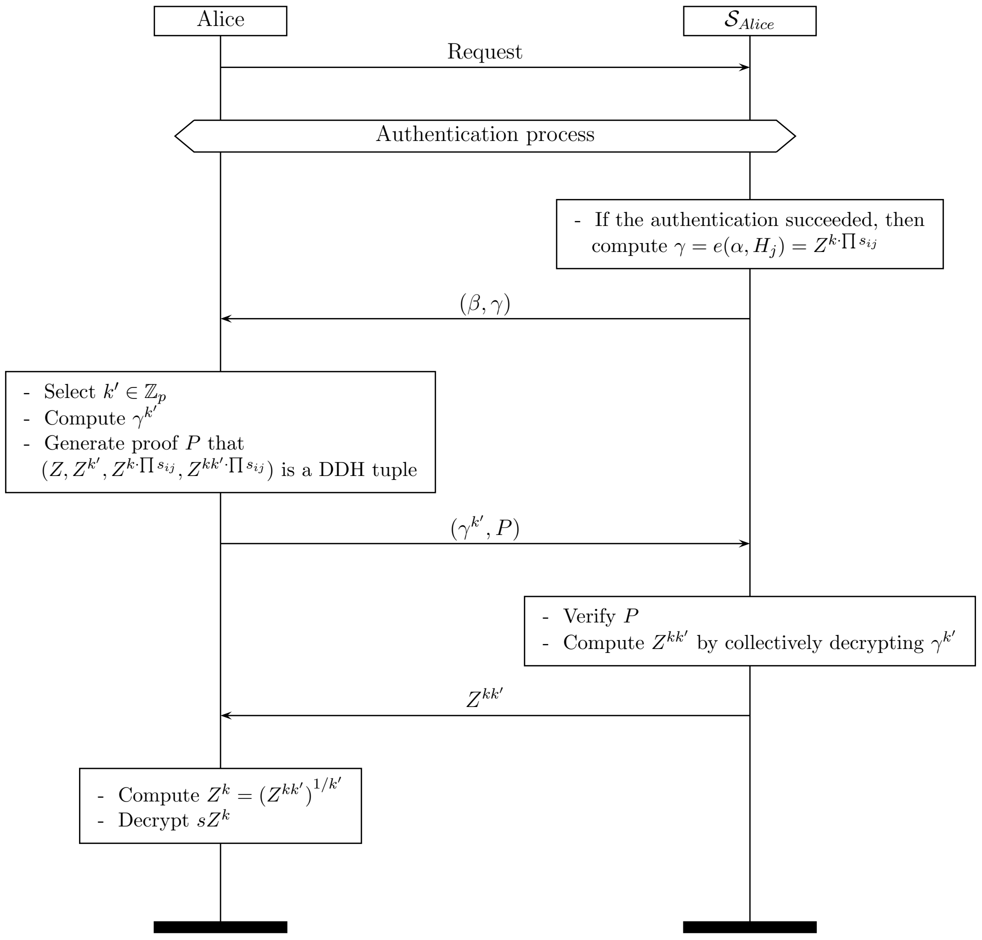

Alice
$\mathcal{S}_{Alice}$
Request
Authentication process
- If the authentication succeeded, then compute $\gamma = e(\alpha, H_j) = Z^{k \cdot \prod s_{ij}}$
(β, γ)
- Select $k' \in \mathbb{Z}_p$
- Compute $\gamma^{k'}$
- Generate proof P that $(Z, Z^{k'}, Z^{k \cdot \prod s_{ij}}, Z^{kk' \cdot \prod s_{ij}})$ is a DDH tuple
$(\gamma^{k'}, P)$
- Verify P
- Compute $Z^{kk'}$ by collectively decrypting $\gamma^{k'}$
$Z^{kk'}$
- Compute $Z^k = (Z^{kk'})^{1/k'}$
- Decrypt sZ^k

Figure 6.5: An abstract presentation of the protocol for decrypting a distributed and encrypted secret.

as reviewed in Chapter 2). Each server also needs to check the received values from other servers in the same way.

6.4 Security analysis

Our goal is to prove the security of our protocol per Definition 6.2; that is, we prove that the advantage that a probabilistic polynomial-time (PPT) adversary $\mathcal{A}$ has to win the *Game-PCSE* is negligible.

We first introduce a new assumption, called "modified decisional bilinear Diffie-Hellman inversion (M-DBDHI)", and show how this assumption is related to our system. We then define a game based on the M-DBDHI assumption. The game has multiple rounds. We call such a game with j rounds as *j-round modified decisional bilinear Diffie-Hellman inversion* game, denoted *Game-j-R-MDBDHI.*

We then prove that if an adversary can win *Game-j-R-MDBDHI* with a non-negligible advantage, then we can break the M-DBDHI assumption.

Finally, we simulate our protocol and adversary model by using *Game-j-R-MDBDHI*, and prove in our theorem that if an adversary can win *Game-PCSE* with a non-negligible advantage ϵ, then we can make use of this adversary to win *Game-j-R-MDBDHI* with advantage $\frac{(1+2\epsilon)(N-N'')(N-N')}{8N^2}$, which is also non-negligible. (Here, the values N, N' and N'' are quantities of servers participating in the protocol.)

6.4.1 Hardness assumption and discussion

We first define the Modified Decisional Bilinear Diffie-Hellman Inversion (M-DBDHI) assumption as follows.

Definition 6.3 (M-DBDHI Assumption). *Given $(g, g^a, g^b, g^x, g^{1/x}, g^{x/b})$, where $g \in G_1$ is a generator, $a, b, x \in \mathbb{Z}_p$, it is hard to distinguish $e(g,g)^{a/b}$ from random.*

Remark 6.5. *Roughly speaking, the connection with our protocol is the following. Let k be an element in $\mathbb{Z}_p$ such that $g^a = g^{bk}$ and let the common public key be $PK = g^b$. Then the ciphertext of a message m is*

$$(\alpha, \beta) = (PK^k, m \cdot e(g,g)^k) = (g^a, m \cdot e(g,g)^{a/b})$$

So, if an adversary is able to distinguish $e(g,g)^{a/b}$ from random, then we can make use of the adversary to determine whether $(g^a, m \cdot e(g,g)^{a/b})$ is a correct encryption of m or not.

As far as we know, the M-DBDHI assumption is not provable from other similar assumptions in the literature. However, a weaker form of it follows from the q-Decisional Bilinear Diffie-Hellman Inversion (q-DBDHI) assumption (see Chapter 2).

We show how to prove the weaker assumption as follows.

Lemma 6.1. *Assume q-DBDHI. Given (g, g^a, g^b), where $g \in G_1$ is a generator, $a, b \in \mathbb{Z}_p$, it is hard to distinguish $e(g,g)^{a/b}$ from random.*

Proof. It can be proved hard by contradiction: If there exists an adversary $\mathcal{A}$ which has a non-negligible advantage ϵ to distinguish $e(g,g)^{a/b}$ from random, then we can construct a PPT Turing machine $\mathcal{B}$ to break the q-DBDHI assumption.

Let $(g, g^x, g^{x^2}, \ldots, g^{x^q}) \in (G_1)^{q+1}$ be the given $(q+1)$-tuple in q-DBDHI, and the challenge for $\mathcal{B}$ is to decide if a given challenge Q is $e(g,g)^{1/x}$. We now explain how to make use of $\mathcal{A}$ to break q-DBDHI assumption.

To generate a challenge for $\mathcal{A}$, $\mathcal{B}$ first randomly picks $a \in \mathbb{Z}_p$, and sends (g, g^a, g^x) and challenge Q^a to $\mathcal{A}$. $\mathcal{A}$ should send a decision to state if $Q^a = e(g,g)^{a/x}$ or not. $\mathcal{B}$ then uses the received decision as his decision on whether $Q = e(g,g)^{1/x}$ or not. So, the advantage that $\mathcal{B}$ can break the q-DBDHI assumption is also ϵ, which is non-negligible. This contradicts our assumption. $\square$

The M-DBDHI assumption is stronger because, in addition to (g, g^a, g^b) which are the elements that an attacker can learn in a single epoch of the protocol, the adversary is given the extra elements $(g^x, g^{1/x}, g^{x/b})$ in the given tuple. In fact, these elements are the knowledge that an attacker can learn from other epochs of our protocol (more details can be found in the proof of Lemma 6.2).

Intuitively, the M-DBDHI is expected to be a hard problem, as the extra information $(g^x, g^{1/x}, g^{x/b})$ does not help one to distinguish $e(g,g)^{a/b}$, due to the discrete logarithm problem and Decisional Diffie-Hellman Problem on G_2. How to formally prove the relationship between M-DBDHI and any well-known hard problem is an open question.

6.4.2 Formal security analysis

We define the *j-round modified decisional bilinear Diffie-Hellman inversion* game (Game-j-R-MDBDHI) as follows.

Game-j-R-MDBDHI

1. The challenger sets $j = 1$, selects $r_0 \in \{0,1\}$ uniformly at random, and generates a tuple $(g, g^{a_0}, g^{b_0}, Q_0)$ according to security parameter λ, where $g \in G_1$ is a generator, $a_0, b_0 \in \mathbb{Z}_p$ such that $|p| = \lambda$. If $r_0 = 0$, then $Q_0 = e(g,g)^{a_0/b_0}$; otherwise Q_0 is randomly chosen from G_2. The challenger sends the tuple to the adversary.

2. Query phase. The adversary selects and makes one of the following two requests to the challenger.

   a) **Update.** Upon receiving this request, the challenger selects new random $a_j, b_j \in \mathbb{Z}_p$, and outputs $g^{b_j/b_{j-1}}$, selects $r_j \in \{0,1\}$ uniformly at random, and an updated tuple $(g, g^{a_j}, g^{b_j}, Q_j)$. If $r_j = 0$, then $Q_j = e(g,g)^{a_j/b_j}$; otherwise Q_j is randomly chosen from G_2.

   b) **Reveal and update.** Upon receiving this request, the challenger outputs (a_{j-1}, b_{j-1}), and updates the tuple as presented above.

   After a challenger answers a request, the challenger sets $j = j + 1$.

3. The query phase is repeated a polynomially bounded number of times.

4. The adversary outputs a decision on whether $Q_i = e(g,g)^{a_i/b_i}$, for any $i \in \{0, 1, \ldots, j\}$, such that the value of (a_i, b_i) and (a_{i+1}, b_{i+1}) is not revealed to the adversary through request (b), if applicable. The adversary wins the game if the decision is correct.

5. After the adversary outputs a decision, the current (a_j, b_j) will be revealed to the adversary.

Remark 6.6. *In the above game, the request (a) will be used indirectly to help us to simulate the protocol for updating decryption keys; and the request (b) will be used indirectly to help us to simulate oracles of compromising a server's decryption key in* Game-PCSE*. The last step, i.e. revealing (a_j, b_j), only happens after the adversary has made a decision. So this has no value for the game. However, it will also indirectly help us to simulate the protocol in the* Game-PCSE.

Lemma 6.2. *Assuming M-DBDHI, an adversary can win* Game-j-R-MDBDHI *only with a negligible advantage.*

Proof. This lemma can be proved by contradiction. Let the j'-th round challenge be the target challenge, for some $j' \in [0, j]$. Apart from the initial knowledge $(g, g^{a_0}, g^{b_0}, Q_0)$, an attacker also has the following extra knowledge:

- all secrets (i.e. (a_i, b_i)) associated to the i-th challenge can be learnt by the attacker, for all i such that $i \in \{0, 1, 2, \ldots, j'-1, j'+2, \ldots\}$. In other words, the j'-th and $(j'+1)$-th secret cannot be learnt by the attacker;
- the attacker can also learn $g^{b_i/b_{i-1}}$ for all $i \in \{1, 2, \ldots, j', \ldots\}$.

Since the secrets related to the j'-th tuple are selected from random, it is independent of the first $(j'-2)$ rounds. The connection between j'-th tuple and $(j'-1)$-th tuple is that when the request (b) is asked on the $(j'-1)$-th tuple, then the attacker learns $a_{j'-1}, b_{j'-1}, g^{b'_j/b_{j'-1}}$. It is easy to see that the attacker will not gain any extra information associated to the j'-th round, due to the discrete logarithm assumption and Diffie-Hellman assumption.

For the secret revealed after $(j'+1)$-th round, the extra knowledge an attacker has is $(a_{j'+2}, b_{j'+2}, g^{b_{j'+2}/b_{j'+1}}, a_{j'+3}, b_{j'+3}, g^{b_{j'+3}/b_{j'+2}}, \ldots)$. Since the secrets generated after $(j'+2)$-th round are also random, and they are not linked to the j'-th round in any format, they are independent of the j'-th round. So, the actual related knowledge the attacker has is

$$(g, g^{a'_j}, g^{b'_j}, g^{b_{j'+1}}, g^{b_{j'+1}/b'_j}, b_{j'+2}, g^{b_{j'+2}/b_{j'+1}}, Q'_j)$$

We now prove that given the above tuple, an adversary cannot determine if $Q'_j = e(g, g)^{a'_j/b'_j}$ with non-negligible advantage. This can be proved by contradiction, namely, if an adversary can determine if $Q'_j = e(g, g)^{a'_j/b'_j}$ with non-negligible advantage, then we can construct a PPT Turing machine $\mathcal{B}$ to break M-DBDHI assumption.

Given a challenge $(g, g^a, g^b, g^x, g^{1/x}, g^{x/b}, Q)$ as stated in M-DBDHI assumption, $\mathcal{B}$ selects a new random r, and computes $(g^{1/x})^r$, then sends $(g, g^a, g^b, g^x, g^{x/b}, r, g^{r/x}, Q)$, which is of the same form as the knowledge the attacker has related to the challenge given in the j'-th round. If the adversary can determine whether $Q = e(g, g)^{a/b}$ or not with non-negligible advantage, then $\mathcal{B}$ can re-direct the adversary's answer to break M-DBDHI assumption with non-negligible advantage, which forms a contradiction.

Thus, if an adversary can win *Game-j-R-MDBDHI* with a non-negligible advantage, we can make use of it to break M-DBDHI assumption. This forms a contradiction. □

Theorem 6.3. *Assuming M-DBDHI, the proposed system is secure in the sense of Definition 6.2.*

Proof. We prove our theorem by contradiction. If an attacker $\mathcal{A}$ is able to win *Game-PCSE* (Definition 6.2), with non-negligible advantage, then we can construct a PPT Turing machine $\mathcal{B}$ to win *Game-j-R-MDBDHI*.

We now explain how to construct $\mathcal{B}$ to make use of $\mathcal{A}$ to win *Game-j-R-MDBDHI*, in the following steps.

Let λ be the security parameter used in this game.

$\mathcal{B}$, as an adversary, starts *Game-j-R-MDBDHI* with a challenger Chl, and obtains $(g, g^{a_0}, g^{b_0}, Q_0)$ from Chl. Then, $\mathcal{B}$, as a challenger, sets up *Game-PCSE* with $\mathcal{A}$ as follows.

- $\mathcal{B}$ sets up our proposed protocol through $Setup(\Pi, \lambda)$. $\mathcal{A}$ declares the number N of servers that will be created.

- **Query phase.** We present how $\mathcal{B}$ answers the following oracle queries made by $\mathcal{A}$. To answer $\mathcal{O}_1$ (i.e. $Settings(\Pi)$), $\mathcal{B}$ outputs current public parameters. We will present how to answer O_2, i.e. to execute the protocol, later. To answer $\mathcal{O}_i$ for $i \in \{3, 4, 5, 6\}$, i.e. create new participants, $\mathcal{B}$ creates a participant accordingly, and if $i \in \{3, 4\}$, then $\mathcal{B}$ sends the corresponding key pair to $\mathcal{A}$. $\mathcal{B}$ can also answer $\mathcal{O}_i$ for $i \in \{7, 8, 9\}$, i.e. compromise or take ownership of participants, since $\mathcal{B}$ creates them and knows all associated secrets. Similarly, $\mathcal{B}$ can answer the decryption oracle $\mathcal{O}_{10}$. Note that there is a special case on answering the $\mathcal{O}_8$ and $\mathcal{O}_9$, which we will discuss after we present how to answer $\mathcal{O}_2$.

To answer $\mathcal{O}_2$, we need to present how to simulate our protocol. In the setup phase, all participants are created through the adversary's create-participant oracle. For the first received $\mathcal{O}_6$ ($CreateSecureServer(\Pi, S_i)$), $\mathcal{B}$ will use g^{b_0} provided by Chl as S_i's initial public key g^{s_i}. We call this server a "trap" server. From $\mathcal{A}$'s point of view, there is no difference between the trap server and other servers. When all N servers are created, $\mathcal{B}$ creates the common public key PK.

To generate the first decryption key, $\mathcal{B}$ can directly generate decryption keys for all the servers he created, since $\mathcal{B}$ knows all corresponding initial secrets. For the trap server, $\mathcal{B}$ does not know the corresponding secret b_0, however, $\mathcal{B}$ can still generate the first decryption key by making request (a) in *Game-j-R-MDBDHI*. The output of request (b) is $(g, g^{a_1}, g^{b_1}, Q_1)$ and g^{b_1/b_0}. The g^{b_1} will be used as the public value associated to the first decryption key b_1, and the value g^{b_1/b_0} is in fact the $g^{u_{i0}}$ associated to the trap server S_i in our protocol, and this value will be used to calculate helper data H_0.

At the end of the j-th epoch, the decryption keys of servers (apart from the trap server) can be easily updated, since $\mathcal{B}$ creates them and knows all the secrets. For the trap server, $\mathcal{B}$ can simulate the decryption key update in the same way as generating the first decryption key. $\mathcal{B}$ asks the request (a) in *Game-j-R-MDBDHI*, i.e. update the tuple from $(g, g^{a_j}, g^{b_j}, Q_j)$ to $(g, g^{a_{j+1}}, g^{b_{j+1}}, Q_{j+1})$ for the $(j+1)$-th epoch.

The encryption process does not need the knowledge of b_j, so $\mathcal{B}$ can pick a random $k \in \mathbb{Z}_p$, and simulates the encryption. $\mathcal{B}$ will store the value of k in order to be able to simulate the decryption oracle without knowing b_j.

Now we can see that the special case of answering $\mathcal{O}_8$ and $\mathcal{O}_9$ is when the trap server is being asked, since $\mathcal{B}$ does not know the value of b_j. In the j-th epoch for some $j \neq 1$, when $CompromiseServer(\Pi, S_i)$ is being made on the trap server S_i, $\mathcal{B}$ asks Chl to reveal the current secret through request (b). Chl reveals b_j, and updates the tuple as stated in *Game-j-R-MDBDHI.* $\mathcal{B}$ then redirects b_j to $\mathcal{A}$.

If at the first epoch the $CompromiseServer(\Pi, S_i)$ is being made on the trap server, or if at any epoch the $TakeOwnershipServer(\Pi, S_i)$ is being made on the trap server S_i, then $\mathcal{B}$ make a random guess as the decision on the currently challenge Q_j from Chl, and the game *Game-j-R-MDBDHI* is over. Note that $\mathcal{B}$ is still able to continue with $\mathcal{A}$, as b_j is revealed at the end of *Game-j-R-MDBDHI*.

- **Security action phase.** $\mathcal{B}$ makes security action oracle queries on all temporarily attacker-controlled servers, and executes the received strategy for the associated server. Then $\mathcal{B}$ updates the decryption keys of all servers as described above in the query phase.

- **Challenge phase.** After receiving two messages M_1 and M_2 from $\mathcal{A}$, $\mathcal{B}$ tosses a coin, and $b \in \{0, 1\}$ is the result of the coin tossing. $\mathcal{B}$ computes PK' such that PK' is the result of replacing b_0 in PK by using a_0. $\mathcal{B}$ can do this because that $\mathcal{B}$ knows g^{a_0} and all other initial secrets. $\mathcal{B}$ then sends $(PK', M_b \cdot Q_0)$ back to $\mathcal{A}$ as the ciphertext C_b.

- If $\mathcal{B}$ has received a correct guess from $\mathcal{A}$, then $\mathcal{B}$ says to Chl that $Q_0 = e(g, g)^{a_0/b_0}$. Otherwise, $\mathcal{B}$ makes a random guess. Note that as previously explained, we can think of the a_0/b_0 as the random number k picked by a client in the encryption phase. So, if $(PK', M_b \cdot Q_0)$ is a correct encryption of message M_b under public key $PK_{\mathcal{S}}$, then we have that $Q_0 = e(g, g)^{a_0/b_0}$.

Note that the above simulation opts out the case that an attacker requests the servers to decrypt the target ciphertext. The attacker can do it by providing the ciphertext to servers for decryption. However, our protocol prevents an attacker to do so by using the two secure zero knowledge proof schemes: one is used for verifying the ownership during the encryption process, and one is used to verify that the to-be-decrypted ciphertext is the one belongs to a client during the decryption process.

Let N' be the number of $TakeOwnershipServer(\Pi, S_i)$ queries made by $\mathcal{A}$ in *Game-PCSE*; and let N'' be the number of $CompromiseServer(\Pi, S_i)$ queries made by $\mathcal{A}$ in the first epoch.

The probability that $\mathcal{B}$ wins the game associated to Q_j with Chl is analysed as follows.

- The probability that the adversary chooses to perform $CompromiseServer(\Pi, S_i)$ in the first epoch on the trap server is $\frac{N''}{N}$. In this case, the probability that $\mathcal{B}$ wins *Game-j-R-MDBDHI* is $\frac{1}{2}$, as $\mathcal{B}$ can only make random guess. This is due to the fact that if the oracle query has been asked at the first epoch, then (a_1, b_1) will be revealed $\mathcal{B}$, and Q_0 is not a valid challenge anymore.

- The probability that $CompromiseServer(\Pi, S_i)$ is not made in the first epoch on the trap server, and $TakeOwnershipServer(\Pi, S_i)$ is performed on the trap server in *Game-PCSE*, is $(1 - \frac{N''}{N}) \cdot \frac{N'}{N}$. In this case, the probability that $\mathcal{B}$ wins *Game-j-R-MDBDHI* is $\frac{1}{2}$, as $\mathcal{B}$ can only make random guess. This is due to the fact that all secrets associated to the *Game-j-R-MDBDHI* will be revealed to $\mathcal{B}$ in order to simulate the $TakeOwnershipServer(\Pi, S_i)$ oracle query, so $\mathcal{B}$ cannot make use of $\mathcal{A}$ to win *Game-j-R-MDBDHI*.

- The probability that $CompromiseServer(\Pi, S_i)$ is not made in the first epoch on the trap server, and trap server has not been asked through $TakeOwnershipServer(\Pi, S_i)$ in *Game-PCSE* is $(1 - \frac{N''}{N}) \cdot (1 - \frac{N'}{N})$. In this case, the probability that $\mathcal{B}$ wins *Game-j-R-MDBDHI* has two cases:

  – if $Q_j = e(g, g)^{a_j/b_j}$, then the probability that $\mathcal{B}$ wins is

  $$(\frac{1}{2} + \epsilon) \cdot 1 + (\frac{1}{2} - \epsilon)) \cdot \frac{1}{2}$$

  (Recall that if $\mathcal{A}$ wins (the probability is $\frac{1}{2} + \epsilon$), then $\mathcal{B}$ will win with probability 1; and if $\mathcal{A}$ does not win, then $\mathcal{B}$ makes a random guess.)

  – if $Q_j \neq e(g, g)^{a_j/b_j}$, then the probability of $\mathcal{B}$ wins is $\frac{1}{2}$, as $\mathcal{B}$ does not have any advantage by using $\mathcal{A}$, since the encryption is not in a correct format.

  These two cases occur with equal probability.

So the advantage $Adv_{\mathcal{B},N,N}(\lambda)$ $\mathcal{B}$ has to win the game associated to Q_j with *Chl* is

$$
\begin{aligned}
Adv_{\mathcal{B},N,N}(\lambda) = & \\
& \frac{N''}{N} \cdot \frac{1}{2} \\
& + (1 - \frac{N''}{N}) \cdot \frac{N'}{N} \cdot \frac{1}{2} \\
& + (1 - \frac{N''}{N}) \cdot (1 - \frac{N'}{N}) \cdot \frac{1}{2}((\frac{1}{2} + \epsilon) \cdot 1 + (1 - (\frac{1}{2} + \epsilon)) \cdot \frac{1}{2}) \\
& + (1 - \frac{N''}{N}) \cdot (1 - \frac{N'}{N}) \cdot \frac{1}{2} \cdot \frac{1}{2} \\
& - \frac{1}{2} \\
= & \frac{(1+2\epsilon)(N-N'')(N-N')}{8N^2}
\end{aligned}
$$

Since ϵ is non-negligible, $N - N' \geq 1$, and $N - N'' \geq 1$, so we have that the advantage $\mathcal{B}$ has to win *Game-j-R-MDBDHI* is non-negligible. This contradicts our assumption. □

6.5 Discussion and related work

6.5.1 Extension to a threshold system

As mentioned in previous sections, our system requires the presence of all servers for recovering a secret. Inspired by [Rab98], the proposed system can be easily extended to a threshold-based system, by using any classical (verifiable) secret sharing schemes to back-up all ephemeral secret keys of servers.

To be more precise, let “key servers” be the servers in our standard protocol and “back-up servers” be the secret sharing servers. Each time a new key of a key server is generated, the key will be distributed to a set of back-up servers through secret sharing schemes, and the shares associated to the old keys will be destroyed. So, when a key server is dead, our system can still continue by recovering the dead server’s secret keys from shares, and take actions from there to re-build the server.

Intuitively, the extended threshold system is secure even if we additionally allow an attacker to compromise less than a threshold number of back-up servers at any epoch, provided that the set of back-up servers are not overlapped with the set of key servers, and all key servers uses the same threshold with the same set of back-up servers for sharing their keys. Loosely speaking, since the shares of different epochs are completely independent to each other, the compromise of shares in an epoch does not help an attacker to recover secrets shared in other epochs.

In fact, we can easily improve the security guarantee of the extended threshold system by letting key servers use different sets of back-up servers for sharing their keys. In this way, an attacker would need to compromise a threshold number of back-up servers to only obtain the secret of a single key server, rather than being able to recover all key servers' secrets. A more rigorous security analysis of the extended threshold system will be our future work.

6.5.2 Related work

A similar adversary model, called "mobile adversary", was introduced by Ostrovsky and Yung [OY91]. The mobile adversary considers viruses that migrate between computers. A system that is secure against a mobile adversary is called proactively secure. Research [CH94, HJKY95, NN04, HJJ^{+}97, FGMY97b, FGMY97a, FMY01, FMY99, Rab98] on proactively secure systems is mainly focusing on secret sharing schemes and signature schemes.

Proactive secret sharing (PSS) (e.g., [HJKY95, NN04, SLL10, BDLO14]) is a technique for sharing a secret among a set of servers; it is secure against an attacker that can compromise servers, one by one, over a long period. In PSS, as in our protocol, time is divided into epochs. In each epoch, the servers that hold shares of the secret engage in a protocol to update their shares. An attacker may compromise some servers in a given epoch, but the learnt secrets are useless in other epochs. Thus, even if all the servers are eventually compromised over different epochs, the secret remains intact provided that in each epoch there was at least one server that remained honest.

One might try to solve the problem of this paper by treating the service's secret key as the secret to be shared among multiple servers. This does not solve the problem, however, because decrypting a message encrypted with the service's public key would require reconstructing the secret key. An attacker that compromises the server holding the secret key at that point would obtain the secret key.

Proactively secure cryptographic systems apply the ideas of proactively secure secret sharing to sharing decryption or signing secrets among several servers. Such systems have been achieved by combining a proactively secure secret sharing scheme with an encryption or signature scheme (e.g., [FGMY97b, FGMY97a, FMY99, FMY01, CKLS02, ADN06]). However, these constructions make use of a trusted dealer, who creates the secret key and distributes shares of some secrets to the servers. Unfortunately, the creation of the secret key in a single location by the dealer prevents the decentralisation required and achieved in our protocol. Although it is mentioned in a number of papers that the function of the trusted dealer in these schemes can be done by the servers, it is well known that both distributing a secret in Shamir's secret sharing scheme and creating and distributing an RSA key, amongst multiple players without a trusted dealer, are complicated and costly. In this paper, we propose a decentralised distributed decryption scheme, which does not have such a trusted dealer and is efficient.

6.6 Conclusion

Increasing numbers of attacks on cloud servers challenge the security of distributed storage. We have introduced a provably secure self-healing distributed storage as a security-enhanced approach to this challenge. It does not require data owners to store decryption keys, and is secure even if all the servers are compromised over a long time, provided that no more than a threshold number of servers are compromised in a single epoch. In addition, the system has a feature of updating decryption keys on the server, while the public key used by data owners remains unchanged. This solves the problem of how to authenticate servers when they are compromisable.

Part V

Conclusion

CHAPTER 7

CONCLUSION

7.1 Introduction

Cryptographic key management is arguably the hardest part of cryptography, and having a secure way to manage cryptographic keys is one of the core assumptions of cryptosystems. If the key management is not secure, then the security of cryptosystems will fail.

The study was set out to explore solutions to defend against attacks on the key management, with a focus on the unauthorised uses of private keys. In particular, this thesis sought to tackle the following challenges:

Challenge 1. How to ensure the authenticity of public keys when an attacker is able to compromise certificate authorities?

Challenge 2. How to mitigate the damage caused by the compromised private keys in secure messaging applications?

Challenge 3. How to provide a better guarantee on the confidentiality of a distributed secret when an attacker is able to gradually compromise all distributed storage servers?

The main findings to the above challenges are part specific and were summarised within the respective parts (i.e. Part II, Part III, and Part IV, respectively). This chapter summarises our findings to the above challenges: it discusses each of our solutions to the challenges, and gives several thoughts on the possible directions of further research.

7.2 Key compromise in web PKI

Many alternatives are proposed to address the challenge of protecting the authenticity of public keys against compromised certificate authorities. They each have different pros and cons, and it is hard to judge which one is better and whether they have addressed the problem completely or not.

To understand the current state of the art, Part II of this thesis first identified 15 fundamental criteria. Second, based on the nature of the proposals' design concept, we classified these proposals into three categories, namely *difference observation*, *scope restriction*, and *certificate management transparency*. Third, we provided an analysis on each of the proposals based on our identified criteria. We observed that although no system satisfies all criteria, systems in the category of certificate management transparency provide more desired features than systems in other categories.

Based on the observations, we presented our new research result on a distributed and transparent key infrastructure, which we call "DTKI". Compared to its predecessors, DTKI has the advantage of minimising the oligopoly of certificate management, and is the first to prevent attacks when all service providers collude together.

However, DTKI does not provide offline verification. As a result, DTKI introduces extra network latency. In addition, the avoidance of oligopoly provided by DTKI requires an international panel to serve as the mapping log maintainer (MLM). In practice, there might be concerns around this requirement, such as how to decide which party should be the MLM, how to organise the governing representatives from many countries to minimise the oligopoly, and how the log maintainers should be funded. To deploy DTKI, further research addressing these concerns is needed.

Overall, we provided what we hope is a clear picture on the current status of the research on solving the first challenge. We contributed a framework for evaluating the potential solutions, presented our improvement (i.e. DTKI) on the current web PKI alternatives, and formally modelled and verified security properties of DTKI using the TAMARIN prover.

7.3 Key compromise in secure communication

Part III developed two key usage detection (KUD) protocols to mitigate the private key compromise in secure communication.

The first protocol is a basic protocol used to explain the concept of KUD, and it requires the recipient being online when a sender wants to send a message. The second protocol is a more developed protocol for secure messaging. It does not require the sender and the recipient being online at the same time, supports multiple devices per user, and the multiplicity of devices helps detect attacks by reporting device activities to the device owner.

Assuming compromised devices can be made secure again, rather than completely lose the game, our protocols could additionally either guarantee the confidentiality of messages sent to a device, or (under certain conditions) allow the victim to detect that confidentiality failed. We formally modelled the detailed KUD protocol, and proved its security properties by using Tamarin prover.

Intuitively, the concept of KUD can also detect situations in which an attacker has access to a key rather than retains a copy of it. For example, it could detect the abuse of keys that are protected using a Trusted Platform Module. However, the detailed use cases and security guarantee requires further study.

Overall, although KUD protocols do not completely solve the problem of key compromise in secure communication, it raises the bar for the attacker, and provides an extra level of security guarantee to the existing messaging systems.

7.4 Key compromise in secret distribution

Part IV aimed to provide a better security guarantee on the system for distributing secrets when an attacker is able to gradually compromise all distributed storage servers, without requiring the secret owner to store any keys for performing decryption.

We developed a security-enhanced self-healing scheme based on bilinear pairings for distributing a secret. With the proposed system, data are encrypted by using a common public key derived from the public keys of the set of servers selected by the owner. Servers update their part of decryption keys periodically, while the common public key remains constant. The system provides a proactive security guarantee: an attacker can learn a secret only if s/he can compromise all servers simultaneously in a short period. We proved the security guarantee by using game-based approach.

Our scheme requires the involvement of all selected servers to perform secret recovery, although there is an easy way to extend it to a threshold-based system as discussed in Section 6.5, further research on the exact security guarantee of the threshold-based scheme and its performance is needed.

Overall, although our solution still allows an attacker to launch attacks, it makes the attacker's work a lot difficult by rendering useless the decryption keys obtained by the attacker in previous epochs.

7.5 Research questions and directions

We list some research questions emerging from the work presented in this thesis, and these questions might lead to possible research directions associated to our presented research.

Eliminating the use of gossip protocols. Both DTKI and KUD are log-based systems, and they require a gossip protocol to exchange their view (i.e. the digest) of the log to prevent "bubble" attacks. One research question is that is it possible to use distributed logs (such as the blockchain used by Bitcoin) to avoid the use of gossip protocol? Also, if this is possible, then what is the security assumption the new system relies on, and what is the advantage of the new system over the one using a gossip protocol?

Providing transparency to all security systems. Both DTKI and KUD provide a transparent key management by using public logs. It forces attackers to leave evidence of their attacks, and enables victims to verify behaviours of participants. Loosely speaking, if an attacker is fully malicious, then the log-based system might help victims to detect attacks; if an attacker is malicious but cautious, which means that the attacker would not launch attacks if the attack will be detected, then it would prevent attacks from the attacker. A research question is that can we apply this concept to all other security systems to make the behaviours of participants transparent for achieving a better security guarantee?

Applying KUD to other systems. This thesis only explored the messaging application of our KUD concept in the settings of public key cryptography. Intuitively, our KUD concept can be applied to the symmetric key cryptosystem as well. Future research could explore more on what are other possible applications. (For example, can we apply KUD to TLS protocol or to security protocols for the Internet of Things?) The related research questions are that can we apply the KUD concept to other cryptosystems, and can we improve the system to achieve a better security guarantee such as attack prevention rather than merely detecting attacks?

Improving the self-healing scheme. This thesis explored how to securely distribute secrets through a self-healing system. The proposed system enables the secret owner to recover the secret when needed. A security concern is that when the secret is being recovered, if the data owner's device is compromised, then the secrets would be exposed to the attacker.

Solutions to address the above concern are future research directions. For example, one research direction is designing a self-healing storage system in the way that a secret can only be accessed for performing some computation (such as message decryption, signature generation, and user authentication), but cannot be recovered directly. This would limit the damage caused by compromised data owner devices.

7.6 Conclusion

Key management is often the most vulnerable part of cryptosystems. This chapter concluded our research on mitigating the private key compromise in three cases, namely private key compromise in web PKI, private key comprise in secure communication, and private key compromise in secret distribution. In addition, to drive the research further, this chapter asked several research questions and indicated possible future research directions.

Bibliography

[ADN06] Jesús F. Almansa, Ivan Damgård, and Jesper Buus Nielsen. Simplified threshold RSA with adaptive and proactive security. In *Proceedings of the Advances in Cryptology - EUROCRYPT 2006, 25th Annual International Conference on the Theory and Applications of Cryptographic Techniques, St. Petersburg, Russia, May 28 - June 1, 2006*, pages 593–611, 2006.

[AF13] Devdatta Akhawe and Adrienne Porter Felt. Alice in warningland: A large-scale field study of browser security warning effectiveness. In *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pages 257–272, 2013.

[AGT01] Aris Anagnostopoulos, Michael T. Goodrich, and Roberto Tamassia. Persistent authenticated dictionaries and their applications. In *Proceedings of the 4th International Conference on Information Security, ISC 2001, Malaga, Spain, October 1-3, 2001*, pages 379–393, 2001.

[AK09] Mansoor Alicherry and Angelos D. Keromytis. Doublecheck: Multipath verification against man-in-the-middle attacks. In *Proceedings of the 14th IEEE Symposium on Computers and Communications (ISCC 2009), July 5-8, Sousse, Tunisia*, pages 557–563, 2009.

[AKV15] Mazhar Ali, Samee U. Khan, and Athanasios V. Vasilakos. Security in cloud computing: Opportunities and challenges. *Information Sciences*, 305:357–383, 2015.

[App11] J. Appelbaum. Detecting certificate authority compromises and web browser collusion. Tor Blog, 2011. `https://blog.torproject.org/`

`blog/detecting-certificate-authority-compromises-and-web-browser-collusion`.

[Art11] C. Arthur. Rogue web certificate could have been used to attack Iran dissidents. The Guardian, 2011. `http://www.theguardian.com/technology/2011/aug/30/faked-web-certificate-iran-dissidents`.

[Bar11a] R. Barnes. Use cases and requirements for DNS-based authentication of named entities (DANE). RFC 6394 (Informational), October 2011.

[Bar11b] Richard L Barnes. DANE: Taking TLS authentication to the next level using DNSSEC. *IETF journal, October*, 7(2), 2011. `http://www.internetsociety.org/articles/dane-taking-tls-authentication-next-level-using-dnssec`.

[BB04a] Dan Boneh and Xavier Boyen. Efficient selective-id secure identity-based encryption without random oracles. In *Proceedings of The Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004*, pages 223–238, 2004.

[BB04b] Dan Boneh and Xavier Boyen. Efficient selective-id secure identity-based encryption without random oracles. In *Proceedings of the Advances in Cryptology - EUROCRYPT, Interlaken, Switzerland, May 2-6, 2004*, pages 223–238, 2004.

[BCK$^+$14] David A. Basin, Cas Cremers, Tiffany Hyun-Jin Kim, Adrian Perrig, Ralf Sasse, and Pawel Szalachowski. ARPKI: attack resilient public-key infrastructure. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*. ACM, 2014.

[BDLO14] Joshua Baron, Karim El Defrawy, Joshua Lampkins, and Rafail Ostrovsky. How to withstand mobile virus attacks, revisited. In *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014*, pages 293–302, 2014.

[BF03] Dan Boneh and Matthew K. Franklin. Identity-based encryption from the weil pairing. *SIAM Journal on Computing*, 32(3):586–615, 2003.

[Bla12] Black tulip report of the investigation into the DigiNotar certificate authority breach. Fox-IT, Technical Report, 2012.

[Bon98] Dan Boneh. The decision diffie-hellman problem. In *Proceedings of the Third International Symposium on Algorithmic Number Theory, ANTS-III, Portland, Oregon, USA, June 21-25, 1998*, pages 48–63, 1998.

[Cer] Certificate patrol. `http://patrol.psyced.org`.

[CH94] Ran Canetti and Amir Herzberg. Maintaining security in the presence of transient faults. In *Proceedings of the Advances in Cryptology - CRYPTO '94, 14th Annual International Cryptology Conference, Santa Barbara, California, USA, August 21-25*, pages 425–438, 1994.

[CKLS02] Christian Cachin, Klaus Kursawe, Anna Lysyanskaya, and Reto Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security, CCS 2002, Washington, DC, USA, November 18-22, 2002*, pages 88–97, 2002.

[CO99] Ran Canetti and Rafail Ostrovsky. Secure computation with honest-looking parties: What if nobody is truly honest? (extended abstract). In *Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing, May 1-4, 1999, Atlanta, Georgia, USA*, pages 255–264, 1999.

[CP92] David Chaum and Torben P. Pedersen. Wallet databases with observers. In *Proceedings of the Advances in Cryptology - CRYPTO '92, 12th Annual International Cryptology Conference, Santa Barbara, California, USA, August 16-20, 1992*, pages 89–105, 1992.

[CSF$^+$08] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile. RFC 5280 (Proposed Standard), May 2008. Updated by RFC 6818.

[CSP$^+$15] Laurent Chuat, Pawel Szalachowski, Adrian Perrig, Ben Laurie, and Messeri Eran. Efficient Gossip Protocols for Verifying the Consistency of Certificate Logs. In *Proceedings of the IEEE Conference on Communications and Networks Security (CNS)*. IEEE, 2015.

[CVE] Common vulnerabilities and exposures. `https://cve.mitre.org/cve/index.html`, Retrieved Feb. 2015.

[CvO13] Jeremy Clark and Paul C. van Oorschot. SoK: SSL and HTTPS: Revisiting past challenges and evaluating certificate trust model enhancements. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 511–525, 2013.

[DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, November 1976.

[DMS04] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *In Proceedings of the 13th USENIX Security Symposium*, pages 303–320, 2004.

[Dom14] The domain name industry brief. Verisign, 2014. `http://www.verisigninc.com/en_US/innovation/dnib/index.xhtml`.

[DR08] T. Dierks and E. Rescorla. The transport layer security (TLS) protocol version 1.2. RFC 5246 (Proposed Standard), August 2008. Updated by RFCs 5746, 5878, 6176.

[DY05] Yevgeniy Dodis and Aleksandr Yampolskiy. A verifiable random function with short proofs and keys. In *Public Key Cryptography - PKC, Les Diablerets, Switzerland, January 23-26, 2005*, pages 416–431, 2005.

[Eck11] Peter Eckersley. Iranian hackers obtain fraudulent HTTPS certificates: How close to a web security meltdown did we get? Electronic Frontier Foundation, 2011. `https://www.eff.org/deeplinks/2011/03/iranian-hackers-obtain-fraudulent-https`.

[Eck12] P. Eckersley. Sovereign key cryptography for internet domains. Internet Draft, 2012.

[EPS14] C. Evansm, C. Palmer, and R. Sleevi. Internet-draft: Public key pinning extension for http. Draft 21, October 2014.

[FFC$^+$11] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 3–14. ACM, 2011.

[FGMY97a] Yair Frankel, Peter Gemmell, Philip MacKenzie, and Moti Yung. Optimal resilience proactive public-key cryptosystems. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22*, pages 384–393, 1997.

[FGMY97b] Yair Frankel, Peter Gemmell, Philip D. MacKenzie, and Moti Yung. Proactive RSA. In *Proceedings of the Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21*, pages 440–454, 1997.

[FMC11] N. Falliere, L. O. Murchu, and E. Chien. W32.stuxnet dossier. Symantec Corporation, Technical report., 2011. `https://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf`.

[FMY99] Yair Frankel, Philip D. MacKenzie, and Moti Yung. Adaptively-secure optimal-resilience proactive RSA. In *Proceedings of the Advances in Cryptology - ASIACRYPT '99, International Conference on the Theory and Applications of Cryptology and Information Security, Singapore, November 14-18*, pages 180–194, 1999.

[FMY01] Yair Frankel, Philip MacKenzie, and Moti Yung. Adaptive security for the additive-sharing based proactive RSA. In *Proceedings of the Public Key Cryptography, 4th International Workshop on Practice and Theory in Public Key Cryptography, PKC 2001, Cheju Island, Korea, February 13-15*, pages 240–263, 2001.

[FS86] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Proceedings of the Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986*, pages 186–194, 1986.

[FY92] Matthew K. Franklin and Moti Yung. Communication complexity of secure computation (extended abstract). In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing, May 4-6, 1992, Victoria, British Columbia, Canada*, pages 699–710, 1992.

[HDWH12] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Mining your Ps and Qs: detection of widespread weak keys in network devices. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 35–35, Berkeley, CA, USA, 2012. USENIX Association.

[HJJ$^+$97] Amir Herzberg, Markus Jakobsson, Stanislaw Jarecki, Hugo Krawczyk, and Moti Yung. Proactive public key and signature systems. In *Proceedings of CCS '97, Proceedings of the 4th ACM Conference on Computer and Communications Security, Zurich, Switzerland, April 1-4*, pages 100–110, 1997.

[HJKY95] Amir Herzberg, Stanislaw Jarecki, Hugo Krawczyk, and Moti Yung. Proactive secret sharing or: How to cope with perpetual leakage. In *Proceedings of the Advances in Cryptology - CRYPTO '95, 15th Annual International Cryptology Conference, Santa Barbara, California, USA, August 27-31*, pages 339–352, 1995.

[HL08] Carmit Hazay and Yehuda Lindell. Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. In *Theory of Cryptography, Fifth Theory of Cryptography Conference, TCC 2008, New York, USA, March 19-21, 2008.*, pages 155–175, 2008.

[HS12] P. Hoffman and J. Schlyter. The DNS-based authentication of named entities (DANE) transport layer security (TLS) protocol: TLSA. RFC 6698 (Proposed Standard), August 2012.

[Int14] Internet users. Internet Usage Statistics, 2014. `http://www.internetlivestats.com/internet-users/`.

[Jou02] Antoine Joux. The weil and tate pairings as building blocks for public key cryptosystems. In *Proceedings of the 5th International Symposium on Algorithmic Number Theory, ANTS-V, Sydney, Australia, July 7-12, 2002*, pages 20–32, 2002.

[JVG$^+$07] Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten Van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems (TOCS)*, 25(3):8, 2007.

[KHP$^+$13] Tiffany Hyun-Jin Kim, Lin-Shung Huang, Adrian Perrig, Collin Jackson, and Virgil Gligor. Accountable key infrastructure (AKI): A proposal for a public-key validation infrastructure. In *22nd International World Wide Web Conference, WWW '13, Rio de Janeiro, Brazil, May 13-17, 2013*, 2013.

[Koc98] P. Kocher. A quick introduction to certificate revocation trees. unpublished work, 1998.

[KWH13] J. Kasten, E. Wustrow, and J. A. Halderman. CAge: Taming certificate authorities by inferring restricted scopes. In *Financial Cryptography and Data Security - 17th International Conference, FC 2013, Okinawa, Japan, April 1-5, 2013, Revised Selected Papers*, 2013.

[Lan11] A. Langley. Public-key pinning. *ImperialViolet* (blog), 2011. `https://www.imperialviolet.org/2011/05/04/pinning.html`.

[Lan12] A. Langley. Revocation checking and Chrome's CRL. *ImperialViolet* (blog), 2012. `https://www.imperialviolet.org/2012/02/05/crlsets.html`.

[Ley12] John Leyden. Trustwave admits crafting SSL snooping certificate: Allowing bosses to spy on staff was wrong, says security biz. The Register, 2012. `www.theregister.co.uk/2012/02/09/tustwave_disavows_mitm_digital_cert`.

[LK12] Ben Laurie and Emilia Kasper. Revocation transparency. Google Research, `http://www.links.org/files/RevocationTransparency.pdf`, September 2012.

[LLK13] B. Laurie, A. Langley, and E. Kasper. Certificate Transparency. RFC 6962 (Experimental), 2013.

[Mar11] M. Marlinspike. SSL and the future of authenticity. In *Black Hat, USA*, 2011.

[MB03] Petros Maniatis and Mary Baker. Authenticated append-only skip lists. *CoRR*, cs.CR/0302010, 2003.

[MBB$^+$14] Marcela S. Melara, Aaron Blankstein, Joseph Bonneau, Michael J. Freedman, and Edward W. Felten. CONIKS: A privacy-preserving consistent key service for secure end-to-end communication. *IACR Cryptology ePrint Archive*, 2014.

[Mer87] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *Proceedings of the Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987*, pages 369–378, 1987.

[MP12] M. Marlinspike and T. Perrin. Internet-draft: Trust assertions for certificate keys (TACK), 2012. `https://tools.ietf.org/html/draft-perrin-tls-tack-02`.

[MS0] MS01-017: Erroneous Verisign-issued digital certificates pose spoofing hazard. Microsoft Support. `http://support.microsoft.com/kb/293818`.

[MSCB13] Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. The TAMARIN prover for the symbolic analysis of security protocols. In *Proceedings of the Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013.*, pages 696–701, 2013.

[NN98] Kobbi Nissim and Moni Naor. Certificate revocation and certificate update. In *Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, USA, January 26-29, 1998*, 1998.

[NN04] Ventzislav Nikov and Svetla Nikova. On proactive secret sharing schemes. In *Selected Areas in Cryptography, 11th International Workshop, SAC 2004, Waterloo, Canada, August 9-10*, pages 308–325, 2004.

[Nor14a] Linus Nordberg. Transparency gossip. INTERNET-DRAFT, 2014.

[Nor14b] Linus Nordberg. Transparency gossip HTTPS transport. INTERNET-DRAFT, 2014.

[OY91] Rafail Ostrovsky and Moti Yung. How to withstand mobile virus attacks (extended abstract). In *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, August 19-21*, pages 51–59, 1991.

[Rab98] Tal Rabin. A simplified approach to threshold and proactive RSA. In *Proceedings of the Advances in Cryptology - CRYPTO '98, 18th Annual International Cryptology Conference, Santa Barbara, California, USA, August 23-27*, pages 89–104, 1998.

[Riv98] Ronald L Rivest. Can we eliminate certificate revocation lists? In *Proceedings of the Financial Cryptography, Second International Conference, FC'98, Anguilla, British West Indies, February 23-25, 1998*, pages 178–183. Springer, 1998.

[Rob11] Paul Roberts. Phony SSL certificates issued for Google, Yahoo, Skype, others. Threat Post, March 2011. `http://threatpost.com/phony-ssl-certificates-issued-google-yahoo-skype.-others-032311`.

[Rya13] Mark Dermot Ryan. Cloud computing security: The scientific challenge, and a survey of solutions. *Journal of Systems and Software*, 86(9):2263–2268, 2013.

[Rya14] Mark Ryan. Enhanced certificate transparency and end-to-end encrypted mail. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014.

[Sch89] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In *Proceedings of the Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989*, pages 239–252, 1989.

[Sch96] Bruce Schneier. *Applied cryptography - protocols, algorithms, and source code in C (2. ed.).* Wiley, 1996.

[SCL$^+$15] Frank Stajano, Bruce Christianson, T. Mark A. Lomas, Graeme Jenkinson, Jeunese Payne, Max Spencer, and Quentin Stafford-Fraser. Pico without public keys. In *Proceedings of the Security Protocols XXIII - 23rd International Workshop, Cambridge, UK, March 31 - April 2*, pages 195–211, 2015.

[Sha79] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[SLL10] David A. Schultz, Barbara Liskov, and Moses Liskov. MPSS: mobile proactive secret sharing. *ACM Trans. Inf. Syst. Secur.*, 13(4):34, 2010.

[SS11] Christopher Soghoian and Sid Stamm. Certified lies: Detecting and defeating government interception attacks against SSL. In *Financial Cryptography and Data Security - 15th International Conference, FC 2011, Gros Islet, St. Lucia, February 28 - March 4, 2011, Revised Selected Papers*, pages 250–259, 2011.

[Ste11] Toby Sterling. Second firm warns of concern after Dutch hack. Yahoo! News, September 2011. `http://news.yahoo.com/second-firm-warns-concern-dutch-hack-215940770.html`.

[The] The EFF SSL Observatory. `https://www.eff.org/observatory`.

[TP11] S. Turner and T. Polk. Prohibiting secure sockets layer (SSL) version 2.0. RFC 6176 (Proposed Standard), March 2011.

[WAP08] Dan Wendlandt, David G. Andersen, and Adrian Perrig. *Perspectives:* improving SSH-style host authentication with multi-path probing. In *Proceedings of the USENIX Annual Technical Conference, Boston, MA, USA, June 22-27, 2008*, pages 321–334, 2008.

[WB13] S. Weiler and D. Blacka. Clarifications and Implementation Notes for DNS Security (DNSSEC). RFC 6840 (Proposed Standard), February 2013.

[YR15] Jiangshan Yu and Mark Dermot Ryan. Device attacker models: Fact and fiction. In *Proceedings of the Security Protocols XXIII - 23rd International Workshop, Cambridge, UK, March 31 - April 2*, pages 158–167, 2015.

APPENDIX A

TAMARIN CODE FOR DTKI

The version of the Tamarin prover we use to perform our proofs is *git commit f1a215550d95d57d3e59267a4fb268dfbf6e826e.* One can run the following command to check the well-formedness of our modeling.

```
$ tamarin-prover file-name.spthy
```

To verify properties, run the following command.

```
$ tamarin-prover --prove file-name.spthy
```

DTKI.spthy

```
theory DTKI

begin

builtins: multiset

functions: adec/2, aenc/2, fst/1, h/1, pair/2, pk/1, sdec/2, senc/2,
  sign/2, snd/1, false/0, true/0, verify/3
equations:
    adec(aenc(x.1, pk(x.2)), x.2) = x.1,
    fst(<x.1, x.2>) = x.1,
    sdec(senc(x.1, x.2), x.2) = x.1,
    snd(<x.1, x.2>) = x.2,
    verify(sign(x.1, x.2), x.1, pk(x.2)) = true


/*Initialisation: Mapping log maintainer. */

rule INIT_MLM:
    [ Fr(~ltkM) ]
  --[ Is_Type('MLM', $M),
      Only_One('INITMLM') // The MLM is unique in DTKI
    ]->
    [
      !Ltk($M, ~ltkM),
      !Pk($M, pk(~ltkM)),
      Out(pk(~ltkM))
    ]

/*Initialisation: Certificate log maintainer CLM1. */

rule INIT_CLM_RGX_1:
    [ Fr(~ltkC) ]
  --[ Is_Type('CLM', $C),
      Only_One('INITCLM_RGX_1') // We only allow two CLMs in this
                                // proof, it can be extended to many
                                // CLMs. We predefined two regular
                                // expressions, namely RGX1 and RGX2,
                                // and each CLM is authorised for
                                // either of them. This one is
                                // authorised for regular expression
                                // 1.
    ]->
    [ !Ltk($C, ~ltkC),
      !Pk($C, pk(~ltkC)),
      Rgx($C, pk(~ltkC), 'RGX1'),
      StStateInitCLM($C, pk(~ltkC), 'RGX1'),
      Out(pk(~ltkC))
    ]

/*Initialisation: Certificate log maintainer CLM2. */

rule INIT_CLM_RGX_2:
    [ Fr(~ltkC) ]
  --[ Is_Type('CLM', $C),
      Only_One('INITCLM_RGX_2') // We only allow two CLMs in this
                                // proof, it can be extended to many
                                // CLMs. We predefined two regular
                                // expressions, namely RGX1 and RGX2,
                                // and each CLM is authorised for
                                // either of them. This one is
                                // authorised for regular expression
                                // 2.
    ]->
    [ !Ltk($C, ~ltkC),
      !Pk($C, pk(~ltkC)),
      Rgx($C, pk(~ltkC), 'RGX2'),
      StStateInitCLM($C, pk(~ltkC), 'RGX2'),
      Out(pk(~ltkC))
    ]

/*Initialisation: Mapping log. */
```

```
rule INIT_MLOG:
     [
      !Ltk($M, ltkM),
      Fr(~mlogid)
     ]
   --[ Is_Type('MLM', $M),
       Only_One('INITMLOG') // Each log maintainer has only one log.
     ]->
     [
      L_Mlog($M, ~mlogid, 'nil') // 'nil' is just a normal constant.
                                 // I use this constant to fill this
                                 // position when no record is available.
     ]


/*Initialisation: Certificate log. */

rule INIT_CLOG_RGX1:
     [
      !Ltk($C, ltkC),
      StStateInitCLM($C, pk(ltkC), 'RGX1'),
      Fr(~clogid)
     ]
   --[ Is_Type('CLM', $C),
       Only_One('CLOG1') // Each log maintainer has only one log.
     ]->
     [
      L_Clog($C, ~clogid, 'RGX1', 'nil') // the log maintained by the
                                          // CLM is for domains whose
                                          // name is an instance of
                                          // regular expression RGX1
     ]

rule INIT_CLOG_RGX2:
     [
      !Ltk($C, ltkC),
      StStateInitCLM($C, pk(ltkC), 'RGX2'),
      Fr(~clogid)
     ]
   --[ Is_Type('CLM', $C),
       Only_One('CLOG2') // Each log maintainer has only one log.
     ]->
     [
      L_Clog($C, ~clogid, 'RGX2', 'nil') // the log maintained by the
                                         // CLM is for domains whose
                                         // name is an instance of
                                         // regular expression RGX2
     ]
```

```
/* Adding a new Clog into Mlog. Phase 1, a CLM sends a request to the MLM.*/

rule MLOG_ADD_NEW_CLOG_PHASE_1_CLM:
        let Request = sign(<'AddCLMRequest', $C, clogid, rgx, pk(ltkC)>, ltkC)
in
        [
         !Ltk($C, ltkC),
         Rgx($C, pk(ltkC), rgx),
         L_Clog($C, clogid, rgx, clog)
        ]
      --[
         Is_Type('CLM', $C)
        ]->
        [
         L_Clog($C, clogid, rgx, clog),
         Out(Request)
        ]

/* Adding a new Clog into Mlog. Phase 2, the MLM records the verified request in the mlog.*/

rule MLOG_ADD_NEW_CLOG_PHASE_2_MLM:
        let
        Request = sign(<'AddCLMRequest', $C, clogid, rgx, ltpkC>, ltkC)
        Proof_MLM = sign( <'AddedByMLM', $C, clogid, rgx, ltpkC>, ltkM)
          in
[
    In(Request),
    !Ltk($M, ltkM),
    !Pk($C, ltpkC),
    L_Mlog($M, mlogid, mlog)
]
--[
   Is_Type('CLM', $C),
   Is_Type('MLM', $M),
   Eq(verify(Request, <'AddCLMRequest', $C, clogid, rgx, ltpkC>, ltpkC), true ),
   Eq(ltpkC,pk(ltkC)),
   Start_Role(<'MANCP2M','M',rgx>, <$M,$C,rgx>) // To find the basic
                                              // trace and spead
                                              // up the proof, we
                                              // only have one
                                              // instance of each
                                              // role.
   ]->
  [
   L_Mlog($M, mlogid, mlog + <$C, clogid, rgx, ltpkC>),
   Out(Proof_MLM)
  ]

/* The MLM can modify its own log */
```

```
  rule MODIFY_MLOG:
let record = <$C, clogid, rgx, ltpkC>
  in
          [
           In(record),
           !Ltk($M, ltkM),
           StCompromisedMLM($M,ltkM),//only a malicious log maintainer
                                      //would modify logs in this way.
           L_Mlog($M, mlogid, mlog)
           ]
         --[
            Is_Type('CLM', $C),
            Is_Type('MLM', $M),
            Modify_Mlog($M, mlogid, ltkM)
           ]->
           [L_Mlog($M, mlogid, mlog + record)]


  /* Adding a new Clog into Mlog. Phase 3, generating a proof for
     convincing other participants. This also enables a malicious MLM
     to create any fake proof about the mlog.*/

  rule MLOG_ADD_NEW_CLOG_PHASE_3_CLM:
   let
          Proof_MLM = sign( <'AddedByMLM', $C, clogid, rgx, ltpkC>, ltkM)
   in

    [
     In(Proof_MLM),
     !Pk($M,ltpkM)
    ]
  --[
     Is_Type('CLM', $C),
     Is_Type('MLM', $M),
     Eq(verify(Proof_MLM, <'AddedByMLM', $C, clogid, rgx, ltpkC>, ltpkM), true )
    ]->
    [
     !Mapping(Proof_MLM)
    ]

  /* Initialisation: Certificate authorities*/

  rule INIT_CA:
  [ Fr(~ltkCA) ]
  --[
     Is_Type('CA', $CA),
     Only_One('CA')
     ]->

  [ !Ltk($CA, ~ltkCA),
    !Pk($CA, pk(~ltkCA)),
    Out(pk(~ltkCA))
  ]


  /* Compromise a CA */

  rule COMPROMISE_CA:
          [ !Ltk($CA, ltkCA)
          ]
  --[
     Is_Type('CA', $CA),
     Compromise_CA($CA,ltkCA)
     ]->

  [
   Out(ltkCA)
  ]

  /* Compromise the MLM */

  rule COMPROMISE_MLM:
          [ !Ltk($M, ltkM)
            ]
  --[
     Is_Type('MLM', $M),
     Compromise_MLM($M,ltkM)
     ]->

  [
   Out(ltkM),
   StCompromisedMLM($M,ltkM)
    ]

  /* Compromise a CLM */

  rule COMPROMISE_CLM:
          [ !Ltk($C, ltkC)
          ]
  --[
     Is_Type('CLM', $C),
     Compromise_CLM($C,ltkC)
     ]->

  [
   Out(ltkC),
   StCompromisedCLM($C,ltkC)
  ]

/*Initialisation: domain server Did, s.t. Did is an instance of RGX1. */
```

```
rule INIT_DOMAIN_RGX1:
     [ Fr(~ltkD),
       Fr(~Did)
     ]
  --[ Is_Type('Domain', $D)]->
  [ !Ltk($D, ~ltkD),
    !Pk($D, pk(~ltkD)),
    !DomainInfo($D, ~Did, 'RGX1', pk(~ltkD)),
    L_LocalRecord($D, ~Did, 'RGX1', 'nil'),
    MasterKey($D, ~Did, 'RGX1', pk(~ltkD)),
    Out(pk(~ltkD))
    ]

/*Initialisation: domain server Did, s.t. Did is an instance of RGX2. */

rule INIT_DOMAIN_RGX2:
     [ Fr(~ltkD),
       Fr(~Did)
     ]
  --[ Is_Type('Domain', $D)]->
  [ !Ltk($D, ~ltkD),
    !Pk($D, pk(~ltkD)),
    !DomainInfo($D, ~Did, 'RGX2', pk(~ltkD)),
    L_LocalRecord($D, ~Did, 'RGX2', 'nil'),
    MasterKey($D, ~Did, 'RGX2', pk(~ltkD)),
    Out(pk(~ltkD))
    ]

  /* Request a master certificate from CAs */
rule REQUEST_MASTER_CERT:
          let request= <'RequestMasterCert', $D, Did, rgx, ltpkD>
            in
     [
      !DomainInfo($D, Did, rgx, ltpkD),
      MasterKey($D, Did, rgx, ltpkD)
     ]
  --[
     Is_Type('Domain', $D),
     Is_Type('CA', $CA)
     ]->
    [
     Out(request)
    ]


  /* A CA certifies a master key. The verification of domain name and
     key is done by linking !Pk($D, ltpkD) to the ($D, ltpkD) in the
     request.*/
```

```
rule CREATE_MASTER_CERT_PHASE_1:
  let request= <‘RequestMasterCert’, $D, Did, rgx, ltpkD>
      Cert_M= sign( <‘MasterCert’, $D, Did, rgx, ltpkD>, ltkCA)
  in
     [
      In(request),
      !Ltk($CA, ltkCA),
      !Pk($D, ltpkD)
     ]
  --[
     Is_Type(‘Domain’, $D),
     Is_Type(‘CA’, $CA),
     Start_Role(<‘CMCP’,‘CA’>,<$CA, $D>)
     ]->
    [
     Out(Cert_M)
    ]

  /* If a statement is signed by a CA, then this is a certificate. The
     reason that we don’t put !MasterCert(Cert_M) in the last rule,
     namely CREATE_MASTER_CERT_PHASE_1, is that the last rule would have to
     link the certificate to the real pk since !Pk is used. However,
     in the real world, a CA is able to certify any public key. Thus
     we have this extra rule to give the CA’s ability to certify any
     pk for any domain.*/

  rule CREATE_MASTER_CERT_PHASE_2:
   let
       Cert_M= sign( <‘MasterCert’, $D, Did, rgx, ltpkD>, ltkCA)
   in

    [
     In(Cert_M),
     !Pk($CA,ltpkCA)
    ]
  --[
     Is_Type(‘Domain’, $D),
     Is_Type(‘CA’, $CA),
     Eq(ltpkCA,pk(ltkCA)),
     Eq(verify(Cert_M, <‘MasterCert’, $D, Did, rgx, ltpkD>, ltpkCA), true )
    ]->
    [!MasterCert(Cert_M)]

/*Generate TLS keys. Any domain owner with a secret key corresponding
  to a master certificate can issue TLS certificates.*/

rule UPDATE_DOMAIN_TLSKEY_PHASE_1:
  let
    Cert_M= sign( <‘MasterCert’, $D, Did, rgx, ltpkD>, ltkCA)
    Cert_TLS=sign( <‘TLSCert’, $D, Did, rgx, ltpkD, pk(~stkD)>, ltkD)
  in
     [ !Ltk($D, ltkD),
       !MasterCert(Cert_M),
       L_LocalRecord($D, Did, rgx, record),
       Fr(~stkD) // a new short term (TLS) key
       ]
   --[ Is_Type('Domain', $D),
       D_Key($D, Did, rgx, ltkD, ~stkD),
       Eq(pk(ltkD),ltpkD),
       Start_Role(<'UDTP','D',rgx>, <$D,$C>)
     ]->
     [!Stk($D, Did, ~stkD),
      L_LocalRecord($D, Did, rgx, record+<$D, Did, ltpkD, pk(~stkD)>),
      Out(Cert_TLS),
      Out(pk(~stkD))
     ]

/*For the reason similar to generating master certificates, we have
  two phases for enabling anyone who has the secret corresponding to a
  master certificate to generate TLS certificates.*/

rule UPDATE_DOMAIN_TLSKEY_PHASE_2:
  let
    Cert_M= sign( <'MasterCert', $D, Did, rgx, ltpkD>, ltkCA)
    Cert_TLS=sign( <'TLSCert', $D, Did, rgx, ltpkD, stpkD>, ltkD)
  in
    [In(Cert_TLS),
     !MasterCert(Cert_M),
     !Pk($CA,ltpkCA)
     ]
  --[
     Is_Type('Domain', $D),
     Is_Type('CA', $CA),
     Eq(ltpkCA,pk(ltkCA)),
     Eq(ltpkD, pk(ltkD))
     ]->
  [!TLSCert(Cert_TLS)]


  /* To publish domain information into clog, a domain first needs to
     query the mapping. This can be done by supplying the Proof_MLM
     directly. */

  rule REQUEST_MAPPING_DOMAIN:
          let Request = <'MappingRequest', rgx, ~n>
  in
          [
           Fr(~n),
           !DomainInfo($D, Did, rgx, ltpkD)
          ]
        --[
           Is_Type('Domain', $D),
           Start_Role(<'RMD','D',rgx>,<$D,rgx>)
          ]->
          [
           StAskMapping(Request),
           Out(Request)
          ]

  rule PROVE_MAPPING:
  let
          Request = <'MappingRequest', rgx, n>
  Proof_MLM = sign( <'AddedByMLM', $C, clogid, rgx, ltpkC, n>, ltkM)

  in
   [
      In(Request),
      !Ltk($M, ltkM),
      !Mapping(Proof_MLM)
  ]
  --[
     Is_Type('CLM', $C),
     Is_Type('MLM', $M),
     Start_Role(<'PM','M',rgx>,<$M,$D,rgx>)
     ]->
    [
     Out(Proof_MLM)
    ]


  rule UP_CLOG_ADD_PHASE_1_Domain:

   let
      Request1 = <'MappingRequest', rgx, n>
      Cert_M= sign( <'MasterCert', $D, Did, rgx, pk(ltkD)>, ltkCA)
      Cert_TLS=sign( <'TLSCert', $D, Did, rgx, ltpkD, stpkD>, ltkD)
      Request2=<'AddDomainRequest', Cert_M, Cert_TLS>
      Proof_MLM = sign( <'AddedByMLM', $C, clogid, rgx, ltpkC, n>, ltkM)
   in
  [
   StAskMapping(Request1),
   In(Proof_MLM),
   !Pk($M,ltpkM),//It is built into browsers.
   !MasterCert(Cert_M),
   !TLSCert(Cert_TLS),
   L_Mlog($M, mlogid, mlog)
  ]
  --[
     Is_Type('MLM', $M),
     Is_Type('CLM', $C),
     Eq(verify(Proof_MLM, <'AddedByMLM', $C, clogid, rgx, ltpkC, n>, ltpkM), true ),
     IsIn(<$C, clogid, rgx, ltpkC>, mlog)
    ]->
  [
   Out(Request2),
   L_Mlog($M, mlogid, mlog)
  ]

   rule UP_CLOG_ADD_PHASE_2_CLM:

  let
     Cert_M= sign( <'MasterCert', $D, Did, rgx, ltpkD>, ltkCA)
     Cert_TLS=sign( <'TLSCert', $D, Did, rgx, ltpkD, stpkD>, ltkD)
     Request2=<'AddDomainRequest', Cert_M, Cert_TLS>
     Proof_MLM = sign( <'AddedByMLM', $C, clogid, rgx, ltpkC>, ltkM)
     Proof_CLM= sign( <'AddedByCLM', Cert_M, Cert_TLS>, ltkC)
  in
  [
   In(Request2),
   !Ltk($C, ltkC),
   !Pk($CA, ltpkCA),
   !Pk($M, ltpkM),
   !Mapping(Proof_MLM),
   L_Clog($C, clogid, rgx, clog)
   ]
  --[ Is_Type('Domain', $D),
      Is_Type('CLM', $C),
      Is_Type('MLM', $M),
      Eq(ltpkC, pk(ltkC)),
      Eq(verify(Cert_M, <'MasterCert', $D, Did, rgx, ltpkD>, ltpkCA), true),
      Eq(verify(Cert_TLS, <'TLSCert', $D, Did, rgx, ltpkD, stpkD>, ltpkD), true),
      Eq(verify(Proof_MLM, <'AddedByMLM', $C, clogid, rgx, ltpkC>, ltpkM), true ),
      Start_Role(<'UCAP','C',rgx>, <$D,$C>)
      ]->
  [
   L_Clog($C, clogid, rgx, clog + <$D, Did, ltpkD, stpkD>),
   Out(Proof_CLM)
  ]

  /* A CLM can modify its own log */
    rule MODIFY_CLOG:
  let record = <$D, Did, ltpkD, stpkD>
  in
          [
           In(record),
           !Ltk($C, ltkC),
           StCompromisedCLM($C,ltkC),//only malicious log maintainers
                                     //would modify logs in this way.
           L_Clog($C, clogid, rgx, clog)
           ]
```

```
       --[
          Is_Type('Domain', $D),
          Is_Type('CLM', $C),
          Modify_Clog($C, clogid, rgx, ltkC)
          ]->
         [L_Clog($C, clogid, rgx, clog + record)]

/* The CLM can provide fake information about Domains*/
rule UP_CLOG_ADD_2:
 let
    Cert_M= sign( <'MasterCert', $D, Did, rgx, pk(ltkD)>, ltkCA)
Proof_CLM= sign( <'AddedByCLM', Cert_M, Cert_TLS>, ltkC)
 in
[
 In(Proof_CLM),
 !Pk($C,ltpkC)
]
--[
   Is_Type('CLM', $C),
   Eq(verify(Proof_CLM, <'AddedByCLM',  Cert_M, Cert_TLS>, ltpkC), true )
  ]->
[!TLS_Cert_In_Clog(Proof_CLM)]

/* Start secure communication */

rule SECURE_COMMUNICATION_USER_1:
        let
    Cert_M= sign( <'MasterCert', $D, Did, rgx, ltpkD>, ltkCA)
    Cert_TLS=sign( <'TLSCert', $D, Did, rgx, ltpkD, stpkD>, ltkD)
    Proof_MLM = sign( <'AddedByMLM', $C, clogid, rgx, ltpkC>, ltkM)
    Proof_CLM= sign( <'AddedByCLM', Cert_M, Cert_TLS>, ltkC)
in
  [
   Fr(~n),
   !Pk($C, ltpkC),
   !Pk($M,ltpkM),
   !MasterCert(Cert_M),//if the random verification is successfully
                      //verified with positive result, then all
                      //users will see the same master certificate
                      //of the same domain.
   !Mapping(Proof_MLM),
   !TLS_Cert_In_Clog(Proof_CLM),
   L_Clog($C, clogid, rgx, clog),
   L_Mlog($M, mlogid, mlog)
   ]
--[
   Is_Type('Domain', $D),
   Is_Type('CLM', $C),
   Is_Type('MLM', $M),
   Eq(verify(Proof_MLM, <'AddedByMLM', $C, clogid, rgx, pk(ltkC)>, ltpkM), true ),
     Eq(verify(Proof_CLM, <‘AddedByCLM’, Cert_M, Cert_TLS>, ltpkC), true ),
     Eq(verify(Cert_TLS, <‘TLSCert’, $D, Did, rgx, ltpkD, stpkD>, ltpkD), true ),
     IsIn(<$D, Did, ltpkD, stpkD>, clog),
     IsIn(<$C, clogid, rgx, ltpkC>, mlog),
     Start_Role(<‘SCU’,‘U’>,<$D>)
     ]->
     [
      Out(aenc{‘1’, ~n}stpkD),
      StSend($D, Did, ~n, rgx, ltpkD, stpkD),
      L_Clog($C, clogid, rgx, clog),
      L_Mlog($M, mlogid, mlog)
     ]

  rule SECURE_COMMUNICATION_DOMAIN_1:
          let m=aenc{‘1’, n}stpk
              Cert_TLS=sign( <‘TLSCert’, $D, Did, rgx, ltpkD, stpkD>, ltkD)
          in
    [
     In(m),
     !Stk($D, Did, stkD),
     !Ltk($D, ltkD),
     !TLSCert(Cert_TLS)
     ]
  --[
     Is_Type(‘Domain’, $D),
     Eq(fst(adec(m, stkD)), ‘1’),
     Start_Role(<‘SCD’,‘D’>,<$D>)
     ]->
     [
      Out( h(snd(adec(m, stkD))) )
      ]

rule SECURE_COMMUNICATION_USER_2:
  [ StSend($D, Did, n, rgx, ltpkD, stpkD),
    In( h(n) )       // Receive hashed secret from network
    ]
  --[ Com_Done($D, Did, n, rgx, ltpkD, stpkD) ]-> // It states that the secret ‘n’
                                                  // was sent to domain $D
    [
     StDone($D, Did, n, rgx, ltpkD, stpkD)
     ]

  /* Domain owners should verify that their master certificate are
     correctly recorded in the log */

  rule DOMAIN_CHECK_MASTER_CERTIFICATE:
let
  Cert_M= sign( <‘MasterCert’, $D, Did, rgx, pk(ltkD)>, ltkCA)
  in
  [
   !Pk($D,ltpkD),
   !MasterCert(Cert_M),
   !DomainInfo($D, Did, rgx, ltpkD)
   ]
  --[
     Is_Type('Domain', $D),
     Eq(pk(ltkD),ltpkD),
     VerifiedMasterCert($D, Did, rgx, ltpkD)
    ]->
    [
     Master_Cert_Verified(Cert_M)
    ]

    /* Detection */

    rule DOMAIN_PERIODICAL_VERIFICATION_GOOD:
  [
    L_LocalRecord($D, Did, rgx, record),
    L_Clog($C, clogid, rgx, clog)
    ,StDone($D, Did, n, rgx, ltpkD, stpkD)
  ]
  --[
     SubsetEq(clog, record),// clog is a subset of D's local
                             // history. It means that all records
                             // about D in the clog are generated
                             // by the domain owner D.
     CheckedLog($D, Did, rgx, 'nil', 'good', 'nil')
  ]->
  [
    L_LocalRecord($D, Did, rgx, record),
    L_Clog($C, clogid, rgx, clog)
  ]

rule DOMAIN_PERIODICAL_VERIFICATION_BAD:
  let
  Cuckoo = <$D, Did, ltpkD, stpkD>
  clog = Cuckoo+rest
  in
  [
   L_LocalRecord($D, Did, rgx, record),
   L_Clog($C, clogid, rgx, clog)
   ,StDone($D, Did, n, rgx, ltpkD, stpkD)
  ]
--[
   NotIn(Cuckoo, record),
   CheckedLog($D, Did, rgx, ltpkD, 'bad', stpkD)  // a bad key is found in
                                                  // clog
  ]->
  [
   L_LocalRecord($D, Did, rgx, record),
   L_Clog($C, clogid, rgx, clog)
  ]

axiom only_one:
  "(All x #i #j. (Only_One(x)@i & Only_One(x)@j) ==> (#i = #j))"

axiom types_distinct:
  "(All t1 t2 x #i #j. (Is_Type(t1,x)@i & Is_Type(t2,x)@j) ==> (t1 = t2))"

axiom eq_check_succeed:
  "All x y #i. Eq(x,y) @ i ==> x = y"

axiom neq_check_succeed:
  "All x y #i. Neq(x,y) @ i ==> not (x = y)"

axiom notin_check_succeed:
  "All x l #i. NotIn(x,l) @ i ==> not (Ex rest. x+rest = l)"

axiom in_check_succeed:
  "All x l #i. IsIn(x,l) @ i ==> Ex rest. x+rest = l"



axiom subseteq_check_succeed:
  "All l m #i. SubsetEq(l,m) @ i ==> ( ( l = m ) | (Ex rest. l+rest = m) )"

axiom one_role_instance:
  "(All #i #j role param1 param2 .
     (
     Start_Role(role,param1) @ i &
     Start_Role(role,param2) @ j
     )
     ==>
     ( #i = #j )
   )"

 /* We can run this protocol correctly without having any compromised party*/

lemma protocol_correctness:
 exists-trace
 " /* It is possible that */
   Ex D Did n rgx ltpkD stpkD #i1.

   /* The user received a confirmation, i.e. hashed secret the user
   has sent, from the network */

   Com_Done(D, Did, n, rgx, ltpkD, stpkD) @ #i1

     /* without the adversary compromising any party. */
   &   not (Ex #i2 CA ltkCA.
                Compromise_CA(CA,ltkCA) @ #i2)
   &   not (Ex #i3 C ltkC.
                Compromise_CLM(C,ltkC) @ #i3)

   &   not (Ex #i4 M ltkM.
                Compromise_MLM(M,ltkM) @ #i4)

"
lemma message_secrecy_no_compromised_party:
 "
 All D Did m rgx ltpkD stpkD #i1.

   /* If a user received a confirmation, i.e. hashed secret the user
   has sent, from the network */

   (Com_Done(D, Did, m, rgx, ltpkD, stpkD) @ #i1

/* and no party has been compromised */
   &   not (Ex #i2 CA ltkCA.
                Compromise_CA(CA,ltkCA) @ #i2)

   &   not (Ex #i3 C ltkC.
                Compromise_CLM(C,ltkC) @ #i3)

   &   not (Ex #i4 M ltkM.
                Compromise_MLM(M,ltkM) @ #i4)
      )
      ==>
      ( /* then the adversary cannot know m */
       not (Ex #i5. K(m) @ #i5)
      )
 "

lemma message_secrecy_compromise_all_domain_verified_master_cert:
 "
 All D Did m rgx ltpkD stpkD #i1.

   /* If a user received a confirmation, i.e. hashed secret the user
   has sent, from the network */

   (Com_Done(D, Did, m, rgx, ltpkD, stpkD) @ #i1

/* and at an earlier time, the domain server has verified his master
certificate */

     & Ex #i2.
     VerifiedMasterCert(D, Did, rgx, ltpkD) @ #i2 &
        #i2 < #i1
/* and all parties can be compromised*/
      )
      ==>
      ( /* then the adversary cannot know m */
       not (Ex #i3. K(m) @ #i3)
      )
 "
  /* Sanity check on verification: can finish trace with good log*/

lemma protocol_can_finish_with_good_log:
 exists-trace
 "/* It is possible that */
   Ex D Did n rgx ltpkD stpkD #i1 #i2.

   /* The user received a confirmation, i.e. hashed secret the user
   has sent, from the network */

   Com_Done(D, Did, n, rgx, ltpkD, stpkD) @ #i1

   /* and we check the log afterwards and find no fake records */
   &  #i1 < #i2
   &  CheckedLog(D, Did, rgx, ‘nil’, ‘good’, ‘nil’) @ #i2

   /* and the adversary did not compromise any party */
   &   not (Ex #i3 CA ltkCA.
                Compromise_CA(CA,ltkCA) @ #i3)

   &   not (Ex #i4 C ltkC.
                Compromise_CLM(C,ltkC) @ #i4)

   &   not (Ex #i5 M ltkM.
                Compromise_MLM(M,ltkM) @ #i5)

 "

lemma detect_bad_records_in_the_log_when_master_cert_not_verified:

 "
 All D Did m rgx ltpkD flag stpkD #i1 #i2 #i3.

   /* If a user received a confirmation, i.e. hashed secret the user
   has sent, from the network */

   (Com_Done(D, Did, m, rgx, ltpkD, stpkD) @ #i1

/* and all parties can be compromised*/

/* and the master certificate of the domain was not initially verified */

/* and the adversary knows m */

     &  K(m) @ #i2
/* and we afterwards check the log */
     &  CheckedLog(D, Did, rgx, ltpkD, flag, stpkD) @ #i3

    &   #i1 < #i3)
      ==>
      ( /* then we can detect a fake record in the log */
        (flag = 'bad')
      )
 "

end
```

APPENDIX B

TAMARIN CODE FOR KUD

The complete work for modeling and proving the detailed KUD protocol contains seven files:

- Makefile (see § B.1).
  The Makefile is used to generate "spthy" files from each "m4" file. To run Makefile, put all files in the same directory, and run "make" in the terminal.
- kud-base.inc (see § B.2).
  This file contains the modeling of the detailed KUD protocol, as presented in § 5.
- kud-correctness.m4 (see § B.3).
  The corresponding spthy file proves the correctness of our modeling.
- kud-secrecy.m4 (see § B.4).
  The corresponding spthy file proves the basic secrecy property.
- kud-trace-log-good.m4 (see § B.5).
  The corresponding spthy file provides a sanity check on our verification, to show that it can find trace with a non-corrupted log.
- kud-trace-log-bad.m4 (see § B.6).
  The corresponding spthy file provides a sanity check on our verification, to show that the detection of attacks will raise an alert.
- kud-detect.m4 (see § B.7).
  The corresponding spthy file proves the key usage detection property.

B.1 Makefile

```
SOURCES := $(shell find . -name '*.m4')
OBJECTS := $(SOURCES:%.m4=%.spthy)
DEPENDS := kud-base.inc

all: $(OBJECTS)

%.spthy: %.m4 $(DEPENDS)
        m4 $< >$@

clean:
        \rm -f $(OBJECTS)
```

B.2 kud-base.inc

```
//theory KUD
begin

builtins: multiset
functions: adec/2, aenc/2, fst/1, h/1, pair/2, pk/1, sdec/2, senc/2,
  sign/2, snd/1, false/0, true/0, verify/3
equations:
    adec(aenc(x.1, pk(x.2)), x.2) = x.1,
    fst(<x.1, x.2>) = x.1,
    sdec(senc(x.1, x.2), x.2) = x.1,
    snd(<x.1, x.2>) = x.2,
    verify(sign(x.1, x.2), x.1, pk(x.2)) = true

/*Initialisation: key owner. */

rule INIT_R:
     [ Fr(~ltkR),
       Fr(~devid) ]
   --[ Is_Type('KeyOwner', $R),
       R_Key($R, pk(~ltkR), ~ltkR),
       LogDeviceStatus(~devid, $R, 'nil'),
       Start_Role('R',~devid)
     ]->
     [ !Ltk($R, ~ltkR),
       !Pk($R, pk(~ltkR)),
       Out(pk(~ltkR)),
       Device(~devid, $R, ~ltkR, 'secure'), // a user's device is initially secure,
                                            // i.e. no vulnerability found so far.
       LogDevice(~devid, $R, 'nil')  // the log stored in Robert's device
     ]

/*Initialisation: log server. */
```

```
rule INIT_L:
    [ Fr(~ltkL) ]
  --[ Is_Type('LogMaintainer', $L),
      Only_One('INITL') // we only allow one log maintainer $L
    ]->
    [ !Ltk($L, ~ltkL),
      !Pk($L, pk(~ltkL)),
      Out(pk(~ltkL))
    ]


rule INIT_LOG:
     [
       Fr(~lcid)
     ]
   --[ Is_Type('LogMaintainer', $L),
       Is_Type('KeyOwner', $R),
       Only_One('INITLOG'), // each user can only have one log with the
                            // log maintainer; which is a reasonable
                            // restriction.
       LogCloudStatus(~lcid, $R, 'nil','start')
     ]->
     [
      LogCloud(~lcid, $R, 'nil') // the log maintained by the log maintainer
                                 // in the cloud
     ]

/* Robert generates short-term keys, signs them, sends a request to the
   log maintainer.*/

rule UPDATE_EPHEMERAL_KEY_R_1:
  let cert_R = sign( <'cert', pk(~stk), $R>, ltkR)
      Req = sign( <'UpdateRequest', cert_R, h(LogD) >, ltkR)
  in
    [ Fr(~stk), //stk is the short term (ephemeral) key
      !Ltk($R, ltkR),
      LogDevice(devid, $R, LogD)
    ]
  --[  Is_Type('LogMaintainer', $L),
       Is_Type('KeyOwner', $R),
       DeviceSecret(devid, $R, ~stk),
       Start_Role(<'UE','B'>,<$R,$L>)
    ]->
    [ LogToAdd($R, cert_R),
      LogDevice(devid, $R, LogD),
      StUpdateEphKeyR1(~stk, $R, ltkR, $L, ~stk, devid),
      Out(Req),
      Out(<pk(~stk),h(LogD)>) // [USS] Unfold potential secrets in signature
    ]
  /*  The log maintainer verifies the signed request, signs the
    confirmation that he has inserted the data into the log; */

rule UPDATE_EPHEMERAL_KEY_L_1:
  let
      cert_R = sign( <'cert', stpk, $R> , ltkR)
      LogCNew = LogC+<$R, cert_R>
      Req = sign( <'UpdateRequest', cert_R, h(LogD) >, ltkR)
      Confirmation = sign( (< 'Confirmation', $R, $L, h(LogCNew), cert_R> ), ltkL)
  in
    [ In(Req),
      !Ltk($L, ltkL),
      !Pk($R, ltpkR),
      LogCloud(lcid, $R, LogC)
    ]
  --[ Eq(verify(cert_R, <'cert',stpk, $R>, ltpkR), true ),
      Eq(verify(Req, <'UpdateRequest', cert_R, h(LogD)>, ltpkR), true ),
      Is_Type('LogMaintainer', $L),
      Is_Type('KeyOwner', $R),
      LogCloudStatus(lcid, $R, LogCNew, 'update'),
      Start_Role(<'UE','L'>,<$R,$L>)
    ]->
    [
      Out(Confirmation),
      CurrentCert($R, cert_R), // this fact shows the current valid
                               // certificate of Robert
      LogCloud(lcid, $R, LogCNew),
      Out(h(LogCNew)) // [USS] Unfold potential new secrets in signature
    ]

/* Robert verifies the confirmation, and updates keys and log in the
   device; */

rule UPDATE_EPHEMERAL_KEY_R_2:
  let
      cert_R = sign(<'cert',stpk, $R>, ltkR)
      Req = sign( <'UpdateRequest', cert_R, h(LogD)>, ltkR)
      Confirmation = sign( <'Confirmation', $R, $L, h(LogC), cert_R>, ltkL)
      LogDNew = LogD+<$R, cert_R>
  in
    [ StUpdateEphKeyR1(tid, $R, ltkR, $L, stk, devid),
      In(Confirmation),
      !Pk($L, ltpkL),
      LogToAdd($R, cert_R),
      LogDevice(devid, $R, LogD)
    ]
  --[ Is_Type('KeyOwner', $R),
      Is_Type('LogMaintainer', $L),
      Eq(verify(Confirmation, <'Confirmation', $R, $L, h(LogC), cert_R>, ltpkL), true),
      Eq(h(LogC), h(LogD+<$R, cert_R>)),
      Eq(stpk,pk(stk)),
      LogDeviceStatus(devid, $R, LogDNew)
    ]->
    [
     StoresKey(devid, $R, stk, cert_R), //this fact represents that Robert
                                        //stores the short-term key in his
                                        //device, the associated certificate
                                        //is cert_R
     LogDevice(devid, $R, LogDNew) // append the new cert into
                                   // his log
    ]

/* An attacker is allowed to compromised a device, and then get the
   long term secret and short term secret stored in it.
   We also add two individual rules to simplify some of the property
   specifications.
 */

rule COMPROMISE_DEVICE_BOTH:
   let
     cert_R = sign(<'cert',pk(stk), R>, ltkR)
   in
    [ !Ltk(R, ltkR),
      StoresKey(devid, R, stk, cert_R),
      Device(devid, R, ltkR, 'secure')
    ]
  --[ Compromise_Device(devid, R, ltkR, stk),
      Is_Type('KeyOwner', R),
      Start_Role('Compromise',<R,devid>)
    ]->
    [ Out(<stk,ltkR>),
      StoresKey(devid, R, stk, cert_R),
      Device(devid, R, ltkR, 'compromised')
    ]

rule COMPROMISE_DEVICE_LTKEY:
    [ !Ltk(R, ltkR),
      Device(devid, R, ltkR, 'secure')
    ]
  --[ Compromise_Device(devid, R, ltkR, 'none'),
      Is_Type('KeyOwner', R),
      Start_Role('Compromise',<R,devid>)
    ]->
    [ Out(ltkR),
      Device(devid, R, ltkR, 'compromised')
    ]

/* Device can be patched. To reduce the verification space and the
   number of open chains, we only allow to fix the device once. In
   other words, rather than having a repected cycle

   Secure--compromised--fixed--compromised again -- fixed...

   we have only one part of the cycle, namely
   secure--compromised--fixed--compromised
   and the device will remain to be broken. In theory, the
   verification should remain the same.*/

rule FIX_DEVICE:
       [ !Ltk(R, ltkR),
         Device(devid, R, ltkR, 'compromised')
       ]
     --[ Is_Type('KeyOwner', R),
         Only_One('FIXDEVICE')
       ]->
       [ Device(devid, R, ltkR, 'secure') ]

/* Sally gets a short-term public key stpk of Robert from the log,
   verifies the signature, encrypts message m by using stpk, then
   sends the ciphertext to Robert. */

rule SECURE_COMMUNICATION_S_1: // Sally requests cert
  let
  m_1=<'CertReq', $R, ~N>
  in
  [ Fr(~N),
    !Pk($R, ltpkR),
    !Pk($L, ltpkL)
  ]
--[ Is_Type('KeyOwner', $R),
    Is_Type('LogMaintainer', $L),
    Start_Role(<'SC','S'>,<$R,$L>)
  ]->
  [ Out(m_1),
    StSecureCommS1(~N, $L, $R, ~N, ltpkR, ltpkL)
  ]

rule SECURE_COMMUNICATION_L_1: // Log maintainer finds the current valid
                               // cert, and gives it to Sally.
  let
  m_1=<'CertReq', $R, N>
  cert_R = sign(<'cert',stpk, $R>, ltkR)
  m_2=sign(<'CertResp',$R, cert_R, N>, ltkL)
  in
  [ In(m_1),
    CurrentCert($R, cert_R),
    !Ltk($L, ltkL)
  ]
--[ Is_Type('KeyOwner', $R),
    Is_Type('LogMaintainer', $L),
    Start_Role(<'SC','L'>,<$R,$L>)
  ]->
  [ Out(m_2)
  ]

rule SECURE_COMMUNICATION_S_2: // Sally verifies the cert, and sends m
                               // encrypted with the associated stpk
  let
  cert_R = sign(<'cert',stpk, $R>, ltkR)
  m_2 = sign(<'CertResp',$R, cert_R, N>, ltkL)
  m_3 = aenc{~m}stpk
  in
  [ Fr(~m),
    StSecureCommS1(tid, $L, $R, N, ltpkR, ltpkL),
    In(m_2)
  ]
--[ Eq(verify(m_2, <'CertResp',$R, cert_R, N>, ltpkL), true ),
    Eq(verify(cert_R, <'cert',stpk, $R>, ltpkR), true ),
    Is_Type('KeyOwner', $R),
    Is_Type('LogMaintainer', $L),
    MsgSent($R, ltkR, stpk, ~m)
  ]->
  [
   Out(m_3)
  ]

rule SECURE_COMMUNICATION_B_1: // Robert uses the current valid short
                               // term key to decrypt the received
                               // message.
  let
  cert_R = sign(<'cert',stpk, $R>, ltkR)
  m_3 = aenc{m}stpk
  in
    [ !Ltk($R, ltkR),
      StoresKey(devid, $R, stk, cert_R),
      In(m_3)
    ]
  --[ Is_Type('KeyOwner', $R),
      MsgReceived(devid, $R, ltkR, stk, m),
      Start_Role(<'SC','B'>,<$R,$L>)
    ]->
    [ ]


rule PERIODICAL_VERIFICATION_GOOD:
  [
    LogCloud(lcid, $R, LogC),
    LogDevice(devid, $R, LogD)
  ]
```

```
--[ CheckedLog(devid, $R, 'good', 'null'),
    SubsetEq(LogC, LogD) // LogC is a subset of LogD
  ]->
  [
    LogCloud(lcid, $R, LogC),
    LogDevice(devid, $R, LogD)
  ]

rule PERIODICAL_VERIFICATION_BAD:
  let
  cert_R = sign( <'cert',stpk, $R> , ltkR)
  Cuckoo = <$R, cert_R>
  LogC = Cuckoo+rest
  in
  [ LogCloud(lcid, $R, LogC),
    LogDevice(devid, $R, LogD)
  ]
--[ NotIn(Cuckoo, LogD),
    CheckedLog(devid, $R, 'bad', stpk)   // a bad key is in LogC but not in LogD
  ]->
  [
    LogCloud(lcid, $R, LogC),
    LogDevice(devid, $R, LogD)
  ]

axiom only_one:
  "(All x #i #j. (Only_One(x)@i & Only_One(x)@j) ==> (#i = #j))"

axiom types_distinct:
  "(All t1 t2 x #i #j. (Is_Type(t1,x)@i & Is_Type(t2,x)@j) ==> (t1 = t2))"

axiom eq_check_succeed:
  "All x y #i. Eq(x,y) @ i ==> x = y"

axiom neq_check_succeed:
  "All x y #i. Neq(x,y) @ i ==> not (x = y)"

axiom notin_check_succeed:
  "All x l #i. NotIn(x,l) @ i ==> not (Ex rest. x+rest = l)"

axiom subseteq_check_succeed:
  "All l m #i. SubsetEq(l,m) @ i ==> ( ( l = m ) | (Ex rest. l+rest = m) )"

// vi:ft=spthy
```

B.3 kud-correctness.m4

```
/* Provide hints for Tamarin's heuristics.
```

```
   In this case, we tell Tamarin to delay (make "Last", "L_") finding the
   sources for the LogDevice fact, since it doesn’t provide useful information
   and can lead to non-termination. It is similar for LogCloud, but the trace
   is found faster if we don’t use L_ for that. */
define(‘LogCloud’,‘L_LogCloud($*)’)
define(‘LogDevice’,‘L_LogDevice($*)’)

theory kud_protocol_correctness

/* Include kud model */
include(‘kud-base.inc’)

// To find the basic trace faster, we first exclude the verification protocol
axiom no_checks:
  "not (Ex #i devid A f k.
     CheckedLog(devid, A, f, k) @ i )"

/* The protocol can run correctly. */

lemma protocol_correctness:
 exists-trace
 " /* It is possible that */
   Ex d R skR dkR m #i.
     /* R received an encrypted message m on device d */
     MsgReceived(d, R, skR, dkR, m) @ #i
     /* without the adversary compromising any device. */
   & not (Ex d2 A ltk dkR #j.
             Compromise_Device(d2, A, ltk, dkR) @ #j)
 "

end

// vi:ft=spthy
```

B.4 kud-secrecy.m4

```
/* Provide hints for Tamarin’s heuristics.
   In this case, we tell Tamarin to delay (make "Last", "L_") finding the
   sources for the LogCloud and LogDevice facts, since they don’t provide useful information
   and can lead to non-termination. */
define(‘LogCloud’,‘L_LogCloud($*)’)
define(‘LogDevice’,‘L_LogDevice($*)’)
define(‘Device’,‘F_Device($*)’)

theory kud_secrecy

/* Include kud model */
include(‘kud-base.inc’)
```

```
/* Basic secrecy property */
lemma message_secrecy:
 "All R skR ekR m #i.
      /* If S sent a message m to R */
      ( MsgSent(R, skR, ekR, m) @ #i &
        /* without the adversary compromising any device */
        not (Ex #j d sk dkR.
                Compromise_Device(d, R, sk, dkR) @ #j)
      )
      ==>
      ( /* then the adversary cannot know m */
        not ( Ex #j. K(m) @ #j)
      )
 "

end

// vi:ft=spthy
```

B.5 kud-trace-log-good.m4

```
/* Provide hints for Tamarin's heuristics.
   In this case, we tell Tamarin to delay (make "Last", "L_") finding the
   sources for the LogCloud and LogDevice facts, since they don't provide useful information
   and can lead to non-termination. */
define(`LogCloud',`L_LogCloud($*)')
define(`LogDevice',`L_LogDevice($*)')
//
changequote`'changequote(`',`')dnl
define(DeviceStatus,F_DeviceStatus($*))
define(Cert,sign(<'cert',pk($2),$1>,$3)) // Cert(R,stk,ltk)
// Fix vim syntax coloring mixup: ')

theory kud_log_good_trace

/* Include kud model */
include(kud-base.inc)

// To find the basic trace faster, we only have one instance of each
// role.
axiom one_role_instance:
  "(All #i #j role param1 param2 .
     (
     Start_Role(role,param1) @ i &
     Start_Role(role,param2) @ j
     )
     ==>
     ( #i = #j )
   )"
```

```
// Similarly, we restrict to only one log check
axiom one_check:
  "(All #i1 #i2 devid1 devid2 A1 A2 f1 f2 k1 k2.
     (
     CheckedLog(devid1, A1, f1, k1) @ i1  &
     CheckedLog(devid2, A2, f2, k2) @ i2
     )
     ==>
     ( #i1 = #i2 )
   )"

/* Sanity check on verification: can finish trace with good log
 */
lemma log_good_trace2:
 exists-trace
 "Ex devid ltkR stk m #i1 #i2 R.
      ( /* If S sent to R an encrypted message m */
        MsgReceived(devid, R, ltkR, stk, m) @ #i1 &
        /* and we check the log afterwards and find no compromise */
        ( #i1 < #i2 ) &
        CheckedLog(devid, R, 'good', 'null' ) @ #i2 &
        /* and the adversary did not compromise anything */
        not (Ex #j devid2 R2 ltkR2 stk2.
                Compromise_Device(devid2, R2, ltkR2, stk2) @ j
            )
      )
 "

end

// vi:ft=spthy
```

B.6 kud-trace-log-bad.m4

```
/* Provide hints for Tamarin's heuristics.
   In this case, we tell Tamarin to delay (make "Last", "L_") finding the
   sources for the LogCloud and LogDevice facts, since they don't provide useful information
   and can lead to non-termination. */
define(`LogCloud',`L_LogCloud($*)')
define(`LogDevice',`L_LogDevice($*)')
//
changequote`'changequote(`',`')dnl
define(DeviceStatus,F_DeviceStatus($*))
define(Cert,sign(<'cert',pk($2),$1>,$3)) // Cert(R,stk,ltk)
// Fix vim syntax coloring mixup: ')

theory kud_log_bad_trace
```

```
/* Include kud model */
include(kud-base.inc)

// To find the basic trace faster, we only have one instance of each
// role.
axiom one_role_instance:
  "(All #i #j role param1 param2 .
     (
     Start_Role(role,param1) @ i &
     Start_Role(role,param2) @ j
     )
     ==>
     ( #i = #j )
   )"

// Similarly, we restrict to only one log check
axiom one_check:
  "(All #i1 #i2 devid1 devid2 A1 A2 f1 f2 k1 k2.
     (
     CheckedLog(devid1, A1, f1, k1) @ i1  &
     CheckedLog(devid2, A2, f2, k2) @ i2
     )
     ==>
     ( #i1 = #i2 )
   )"

axiom one_compromise:
 "(All #i1 #i2 devid1 devid2 R1 R2 ltkR1 ltkR2 stk1 stk2.
     ( Compromise_Device(devid1, R1, ltkR1, stk1) @ #i1 &
       Compromise_Device(devid2, R2, ltkR2, stk2) @ #i2 )
     ==>
     ( ( #i1 = #i2 ) & ( stk1 = 'none') )
  )
 "

/* Sanity check on verification: detection may raise an alert
 */
lemma log_bad_trace2:
 exists-trace
 " Ex devid ltkR stpk m #i1 #i2 #i3 #i4 R k stkalt.
      ( /* If S sent to R an encrypted message m */
        MsgSent(R, ltkR, stpk, m) @ #i1 &
        /* and the adversary knows m */
        K(m) @ #i2 &
        /* compromise the long-term key, not the specific ephemeral
           key */
        Compromise_Device(devid, R, ltkR, stkalt) @ #i3  &
        not (pk(stkalt) = stpk) &
        /* and we check the log and find a compromise */
        CheckedLog(devid, R, 'bad', k ) @ #i4
      )
 "

end

// vi:ft=spthy
```

B.7 kud-detect.m4

```
/* Provide hints for Tamarin's heuristics.
   In this case, we tell Tamarin to delay (make "Last", "L_") finding the
   sources for the LogCloud and LogDevice facts, since they don't provide useful information
   and can lead to non-termination. */
//xdefine(`LogCloud',`L_LogCloud($*)')
define(`LogDevice',`L_LogDevice($*)')
//
changequote`'changequote(`',`')dnl
define(Device,F_Device($*))
define(StoresKey,F_StoresKey($*))
define(Cert,sign(<'cert',pk($2),$1>,$3)) // Cert(R,stk,ltk)
// Fix vim syntax coloring mixup: ')

theory kud_detect

/* Include kud model */
include(kud-base.inc)

axiom only_one_device:
  " All devid1 devid2 #i #j.
    ( Start_Role('R',devid1) @ i &
      Start_Role('R',devid2) @ j )
    ==>
    ( #i = #j )
  "

axiom only_one_Lora:
  " All l1 l2 #i #j.
    ( Is_Type('LogMaintainer',l1) @ i &
      Is_Type('LogMaintainer',l2) @ j )
    ==>
    ( l1 = l2 )
  "

axiom at_most_two_of_a_role:
  " All r p1 p2 p3 #i1 #i2 #i3 .
    ( Start_Role(r,p1) @ #i1 &
      Start_Role(r,p2) @ #i2 &
      Start_Role(r,p3) @ #i3 )
    ==>
    ( ( #i1 = #i2 ) | ( #i1 = #i3 ) | ( #i2 = #i3 ) )
  "

axiom limit_receives:
  " All devid a1 a2 b1 b2 c1 c2 d1 d2 #i1 #i2 .
    ( MsgReceived(devid, a1, b1, c1, d1) @ #i1 &
      MsgReceived(devid, a2, b2, c2, d2) @ #i2 )
    ==>
    ( #i1 = #i2 )
  "

// We want this to hold even if there is only one compromise
axiom one_compromise:
  "(All #i1 #i2 devid1 devid2 A1 A2 ltk1 ltk2 k1 k2.
     (
     Compromise_Device(devid1, A1, ltk1, k1) @ i1  &
     Compromise_Device(devid2, A2, ltk2, k2) @ i2
     )
     ==>
     ( #i1 = #i2 )
   )"

// Similarly, we restrict to only one check
axiom one_check:
  "(All #i1 #i2 devid1 devid2 A1 A2 f1 f2 k1 k2.
     (
     CheckedLog(devid1, A1, f1, k1) @ i1  &
     CheckedLog(devid2, A2, f2, k2) @ i2
     )
     ==>
     ( #i1 = #i2 )
   )"

/* Sanity check: can we actually reach the situation we aim for in the
 * 'detect' check? */
lemma detect_usage_trace_flag2:
  exists-trace
  " Ex devid ltkR stk m #i1 #i3 #i4 flag R k.
        ( /* If an honest paty sent to R an encrypted message m */
          ( MsgSent(R, ltkR, pk(stk), m) @ #i1 &
            /* and the adversary knows m */
            K(m) @ #i3 &
            /* and this stk was not specifically compromised */
            /* in other words, the compromise was in a different epoch
             */
            not (Ex #j ltk . Compromise_Device(devid, R, ltk, stk) @ j ) &
            /* and we afterwards check the log */
            CheckedLog(devid, R, flag, k ) @ #i4 &
            #i1 < #i4
          )
        )
       "

/* Basic secrecy property, proven elsewhere */
axiom message_secrecy2:
  " All R ltkR stpk m #i.
        ( /* If S sent a message m to R */
          ( MsgSent(R, ltkR, stpk, m) @ i &
            /* without the adversary having compromised a device of R at
            some point */
            not (Ex #r devid stk. Compromise_Device(devid, R, ltkR, stk) @ r)
          )
          ==>
          ( /* then the adversary does not know it */
            not ( Ex #j. K(m) @ j)
          )
        )
       "



/* Main detection of usage property.
 */
lemma detect_usage_S_sends:
 "All d skR dkR m #i1 #i2 #i3 detectionresult R k.
      /* If S sent to R an encrypted message m,
         where pk(dkR)=ekR */
      ( MsgSent(R, skR, pk(dkR), m) @ #i1 &
        /* and the adversary knows m */
        K(m) @ #i2 &
        /* and the ephemeral key used by the sender
           was not compromised, i.e., the compromise
           occurred in a different epoch
         */
        not (Ex #j sk .
                Compromise_Device(d, R, sk, dkR) @ #j ) &
        /* and Robert afterwards checks the log */
        CheckedLog(d, R, detectionresult, k ) @ #i3 &
        #i1 < #i3
      )
      ==>
      ( /* then we detect a compromise */
        (detectionresult = 'bad')
      )
 "

/* Main detection of usage property.
 */
lemma detect_usage_R_receives:
 "All d skR dkR dkR2 m #i1 #i2 #i3 #i4 detectionresult R k.
      /* If S sent to R an encrypted message m,
         where pk(dkR)=ekR */
      ( MsgSent(R, skR, pk(dkR), m) @ #i1 &
        MsgReceived(d, R, skR, dkR2, m) @ #i2 &
        /* and the adversary knows m */
        K(m) @ #i3 &
        /* and the ephemeral key used by the sender was
           not compromised, i.e., the compromise was in
           a different epoch then when m was sent.
         */
        not (Ex #j sk .
                Compromise_Device(d, R, sk, dkR) @ #j ) &
        /* and Robert afterwards checks the log */
        CheckedLog(d, R, detectionresult, k ) @ #i4 &
        #i2 < #i4
      )
      ==>
      ( /* then we can detect a compromise */
        (detectionresult = 'bad')
      )
 "

end

// vi:ft=spthy
```