KTH ROYAL INSTITUTE
OF TECHNOLOGY

Degree Project in Computer Science and Engineering

Second cycle, 30 credits

Keeping an Indefinitely Growing Audit Log

MÅNS ANDERSSON

Stockholm, Sweden, 2022

Keeping an Indefinitely Growing Audit Log

MÅNS ANDERSSON

Degree Programme in Computer Science and Engineering
Date: December 1, 2022

Supervisor: Hongyu Jin
Examiner: Panos Papadimitratos
School of Electrical Engineering and Computer Science
Host company: PrimeKey Solutions AB
Swedish title: En kontinuerligt växande audit log

Abstract

An audit log enables us to discover malfeasance in a system and to understand a security breach after it has happened. An audit log is meant to preserve information about important events in a system in a non-repudiable manner. Naturally, the audit log is often a target for malicious actors trying to cover the traces of an attack. The most common type of attack would be to try to remove or modify entries which contain information about some events in the system that a malicious actor does not want anyone to know about.

In this thesis, the state-of-the-art research on secure logging is presented together with a design for a new logging system. The new design has superior properties in terms of both security and functionality compared to the current EJBCA implementation. The design is based on a combination of two well-cited logging schemes presented in the literature.

Our design is an audit log built on a Merkle tree structure which enables efficient integrity proofs, flexible auditing schemes, efficient queries and exporting capabilities. On top of the Merkle tree structue, an FssAgg (Forward secure sequential Aggregate) MAC (Message Authentication Code) is introduced which strengthens the resistance to truncation-attacks and provides more options for auditing schemes.

A proof-of-concept implementation was created and performance was measured to show that the combination of the Merkle tree log and the FssAgg MAC does not significantly reduce the performance compared to the individual schemes, while offering better security. The logging system design and the proof-of-concept implementation presented in this project will serve as a starting point for PrimeKey when developing a new audit log for EJBCA.

Keywords

Cryptography, Audit Log, Tamper-evident, Merkle tree

Sammanfattning

En granskningslogg är viktig eftersom den ger oss möjligheten att upptäcka misstänkt aktivitet i ett system. Granskningsloggen ger också möjligheten att undersöka och förstå ett säkerhetsintrång efter att det har inträffat. En attackerare som kompromet terar ett system har ofta granskningsloggen som mål, eftersom de ofta vill dölja sina spår.

I denna rapport presenteras en litteraturstudie av nuvarande forskning på säkra loggingsystem samt en design av ett nytt loggingsystem. Det nya loggingsystemet har bättre säkerhetsegentskaper och funktionalitet jämfört med den nuvarande implementationen i EJBCA. Designen bygger på en kombination av två välciterade forskningsartiklar.

Vår design är en granskningslogg baserad på en Merkle träd-struktur som möjliggör effektiva bevis av loggens integritet, flexibel granskning, effektiv sökning och exportfunktionalitet. Förutom Merkle träd-strukturen består den nya loggen även av en FssAgg (Forward secure sequential Aggregate) MAC (Message Authentication Code) som förstärker loggens motstånd mot trunkeringsattacker och möjliggör fler sätt att granska loggen.

En prototypimplementation skapades och prestandamätningar genomfördes som visar att kombinationen av Merkle träd-loggen och FssAgg MAC:en inte försämrar loggens prestanda jämfört med de individuella logglösningarna, trots att starkare säkerhet uppnås. Designen av det nya loggingsystemet samt prototypimplementationen kommer att utgöra en grund för PrimeKeys arbete med att implementera en ny audit log i EJBCA.

Nyckelord

Kryptografi, Granskningslogg, Manipuleringsupptäckbarhet, Merkle träd

Acknowledgments

I would like to thank Professor Panos Papadimitratos and Hongyu Jin Ph.D. at NSS Group at KTH Royal Insitute of Technology for being supportive during this project and helping me achieve the necessary academic quality of the report.

I would also like to thank all the people at Keyfactor (formerly PrimeKey) for providing the idea for this project, for letting me do this project at the company, and for being supportive and helpful all the way. A special thanks to Mike Agrenius Kushner for supervising the project and always being helpful.

A big thank you to my fellow students and great friends Joakim Loxdal, Eric Vickström, Oscar Almqvist and Kalle Meurman for input and support.

Stockholm, December 2022
Måns Andersson

Contents

List of Figures

List of Tables

List of acronyms and abbreviations

API	Application Programming Interface
CA	Certificate Authority
CPU	Central Processing Unit
CRL	Certificate Revocation List
CT	Certificate Transparency
DB	Database
EC	Elliptic Curve
ECDSA	Elliptic Curve Digital Signature Algorithm
EJBCA	Enterprise Java Beans Certificate Authority
FIFO	First In First Out
FssAgg	Forward secure sequential Aggregate
GDPR	General Data Protection Regulation
HMAC	Hash-based Message Authentication Code
HTTP	Hypertext Transfer Protocol
IBE	Identity Based Encryption
IBS	Identity Based Signature
LGPL	Lesser General GNU Public License
MAC	Message Authentication Code
MMT	Maximum Merge Delay
OCSP	Online Certificate Status Protocol
PKI	Public Key Infrastructure
RSA	Rivest–Shamir–Adleman
SCT	Signed Certificate Timestamp
SHA-256	Secure Hashing Algorithm, 256-Bits
TLS	Transport Layer Security
WORM	Write-Once Read-Many

Chapter 1

Introduction

The audit log is important, not only to be able to keep track of what is happening in a system but also to be able to discover suspicious activity and to learn more about a security breach after it has happened. This means that an attacker whose main objective is some other part of the system often has the audit log as their first target [1], to hide their tracks. For that reason, many different secure logging schemes have been proposed over the years, each with different properties, strengths and weaknesses.

In this thesis, we focus on a software called EJBCA (Enterprise Java Beans Certificate Authority) [2] which is an open-source CA (Certificate Authority) software maintained by a company called PrimeKey. It has an audit log that serves multiple purposes. It enables the CA to prove that they have performed a certain action, such as certificate issuance, and when it happened. It should also provide non-repudiation; if the CA has issued a certificate, the audit log should make it impossible for the CA to deny that issuance.

1.1 Background

1.1.1 PKI

Information encryption and authentication is used by organizations all over the world, all the time. Organizations often use a PKI (Public Key Infrastructure) to manage keys, identities and certificates, thereby making encryption manageable. Asymmetric encryption is a common form of encryption which involves a pair of keys, a public and a private key [3]. The public key, as the name suggests, is made public and anyone can use it to encrypt a message. But only the one holding the private key (secret key), which was generated

together with the public key, can decrypt that message. The private key can also be used to sign information, making it possible to validate the identity of the sender of the signed information. In the 1990s, PKIs emerged and were used to create digital certificates which connect an identity with a public key. The certificates can be seen as the passports or driver's licenses of the digital world [3]. The most publicly known PKI is the Web PKI where TLS server certificates helps facilitate trust for a user when visiting web pages. But PKIs can be used for many circumstances. For instance, digital certificates are used by car manufacturers to give vehicles identities [4][5].

The primary role of a CA in a PKI is to handle certificate issuance and revocation. The CA often requires the entity requesting a certificate to prove its identity in some way. In the case of the Web PKI, certificates connect public keys with domain names. In that case, the CA often requires that the entity requesting a certificate for a certain domain prove that they own the domain by adding an entry in the DNS (Domain Name System) for that domain [6]. CAs have their own certificates and are often placed in a hierarchy where an initial trusted CA delegates responsibility to another CA that is at a lower position in the hierarchy. This hierarchy can have multiple levels. When verifying the certificate of a CA, a root CA will eventually be reached. The root CA is at the top of the hierarchy and its certificate is self-signed [3]. The client often has a list of trusted root CAs; in the case of the Web PKI, a list of certificates for root CAs is included with the operating system [7].

It is worth mentioning that background of PKI and EJBCA is helpful for understanding the use case of the audit log. However, the audit log design presented in chapter 6 could be used with any system that requires secure logging, such as access logging for a physical premise, even though it was designed to suit the needs of EJBCA.

1.1.2 Audit Log

The most important security property of an audit log is integrity. The integrity of a log is compromised if log entries are modified, removed or reordered. This implies that a secure audit log should be strictly append-only, i.e. entries can only be added at the end of the log.

A log is tamper-evident [8] if it is possible to cryptographically prove whether the integrity of the log was compromised or not. An even stronger assumption is a tamper-resistant log on tamper-resistant hardware such as a WORM (Write-Once Read-Many) storage [9]. Without special purpose hardware, a tamper-resistant log is impossible. This is due to the fact that

standard storage hardware enables both reading and writing, and it is possible for an attacker to modify the storage, regardless of the logging scheme. In this thesis, we aim at designing a tamper-evident log while tamper-resistant hardware is out of scope.

Integrity-compromise, thus tampering, of the log can only be detected through some verification process performed on the log. Therefore it is not only important that the log is tamper-evident but also that there are flexible and efficient ways to verify the integrity of the log.

1.2 Problem

The current audit log system of EJBCA is based on a table structure with signatures for the log entries and sequence numbers to give an ordering of entries. Auditing and searching in the log is slow and inconvenient. Performing searches in the audit log occupies the database, which makes the CA software unable to perform other tasks, that is not ideal. A security flaw of the current audit log is that it is impossible to detect when entries have been removed from the tail-end of the log. This is known as a truncation-attack [10]. In other words, the log is not tamper-evident.

Our design aims to address the following requirements:

- **Entity authentication** - only trusted entities should be allowed to write new entries to the log.
- **Tamper-evident** - log tampering should be detectable, including modification, deletion or reordering of log entries.
- **Scalability** - the log should be searchable in an efficient way and insertions should be efficient, and remain so for growing log size.
- **Pruning** - it should be possible to prune away and export parts of the log in a way that does not cause issues with verification.

The requirements are expanded upon in chapter 5.

1.2.1 Original Problem and Definition

The purpose of the project is to propose a logging system to satisfy the requirements. The new design should be evaluated based on the requirements and a proof-of-concept implementation should be created.

1.3 Purpose

From the point of view of PrimeKey, the project should serve as an inspiration and a starting point for implementing a new and improved logging system for EJBCA. Some EJBCA customers have complained about the audit log, mostly in regards to slow searching and slow verification. Therefore it is reasonable, from a business standpoint, for PrimeKey to upgrade the audit log.

From the author's point of view, this project offers an opportunity to learn more about security, cryptography and software engineering. Designing a tamper-evident and efficient, scalable log system is interesting both from an engineering and an academic point of view since the problem combines aspects of cryptography, security and system design.

1.4 Goals

The goals that this project tries to achieve are as follows:

1. Provide a literature review presenting the most important research done on the topic of secure logging.
2. Describe how the audit log currently works within EJBCA to provide insight into why a new system is needed and what is important to improve for the new system.
3. Clearly describe the most important requirements for the new log system.
4. Propose a design for a new audit log system based on the discoveries from the previous goals.
5. Evaluate how well the new log system satisfies the requirements.
6. Provide a proof-of-concept implementation to serve as a basis for future development.
7. Measure performance of different log operations on the proof-of-concept implementation to evaluate the viability of the design.

1.5 Delimitations

It was made clear from the start that PrimeKey did not expect a full implementation of a new audit log system within EJBCA. Their hope was to have a report serving as an inspiration for a future implementation. The main focus would be to provide a theoretical background on secure logging and a theoretical design for a new and improved logging system. A proof-of-concept implementation was said to be considered a nice bonus but not a necessity.

- The project does not involve any analysis of legal requirements placed on audit logs based on data protection and privacy laws.
- No special-purpose hardware was considered. The new logging system was assumed to be running on standard computer hardware with read-write storage.
- The topics of data redundancy and backups are not explored in this project. The main focus regarding log design is to have a log that is tamper-evident, i.e. tampering is detectable. An attacker could also be seeking to destroy the log, without covering their tracks, by deleting all entries or a large portion of the entries. Protection against this type of attack could be achieved by having redundancy of the log data.
- Encryption functionality is not considered a requirement for the new log design. The current EJBCA audit log has integrity protection functionality, but not encryption.

1.6 Structure of the Thesis

Chapter 2 presents the literature review of the state-of-the-art research on secure logging. Chapter 3 describes the methodology used in this project. Chapter 4 is an analysis of the current EJBCA audit log system and chapter 5 presents the requirements identified for the new logging system. Chapter 6 describes the design of the new logging system. Chapter 7 describes the proof-of-concept implementation and presents performance metrics. The report ends with chapter 8 which summarizes the project and discusses future work.

Chapter 2

Background

Research on cryptoraphic schemes for securing logs is presented in this chapter. Different schemes have different techniques for securing the integrity of the log and also different data structures for storing the log entries. Some logging schemes in the literature are more powerful than others but in general they all offer different trade-offs in terms of security and performance for log entry insertion and integrity validation.

The potential benefits of blockchain technology for secure logging [11] and for PKIs [12][13] have been explored in the literature. The strength of blockchain technology emerges in distributed settings. Since the EJBCA audit log is not distributed, blockchain technology does not provide benefits and is therefore not explored further in this chapter.

2.1 Schneier-Kelsey Log

In 1999, Schneier and Kelsey presented a logging system [14] that allows logging to occur on an untrusted machine while occasionally being transferred to a trusted machine. For example, the untrusted device could be some sort of smart card that logs events describing its usage. The card itself is not considered to be tamper-resistant to guarantee that an attacker could not alter its contents. The goal of the log design presented by Schneier and Kelsey is that an attacker should not be able to read the content of the log (confidentiality) and if the attacker alters or deletes some part of the log it should be detectable once the untrusted logger interacts with the trusted machine. In other words, it should also be tamper-evident.

This is achieved by using a symmetric encryption key which evolves through a hash function with every new log entry. The trusted machine and the

untrusted logger agree on the initial key. With every new log entry, the logger removes the previous key from memory after it has created the new key with the hash function. This key is used both for encryption of the log content and for the MAC (Message Authentication Code) for each log entry. The result is that an attacker can not read an old log entry since it is impossible to recreate any of the old keys. This is thanks to the trap-door characteristic of the hash function.

For the same reason it is also impossible for the attacker to alter an old log entry without being detectable by the trusted machine. We assume a cryptographically secure hash function, such as SHA-256. For the MAC, an HMAC (Hash-based Message Authentication Code) could be used which retains the security of the underlying hash function [15].

Each log entry also contains a hash chain field. This contains a hash of the concatenation of the previous hash chain entry and the new log entry data. The result is that if an attacker deletes an old entry or changes the ordering of entries, it will be detectable by the trusted machine. The MAC of an entry is created using the hash chain field as input. This is done since the hash chain field encapsulates information about the entry's position in the log and the log entry data.

One major flaw of the design is the vulnerability to so-called truncation-attacks [10]. If an attacker deletes a continuous set of tail-end log entries, it would actually not be detectable by a verifier. Since only entries from the tail-end of the log are removed, the hash chain would still be valid. However, if the attacker truncates a log entry that has been seen by the trusted machine or a verifier at an earlier point in time, the attack would be detectable. After a truncation-attack has been performed and a new log entry is added, the attack will also be detectable since the wrong symmetric key will be used at the wrong position in the log. But if an attacker has been able to compromise the logger and remove events from the end of the log, it is also likely that the attacker could prevent new events from being added.

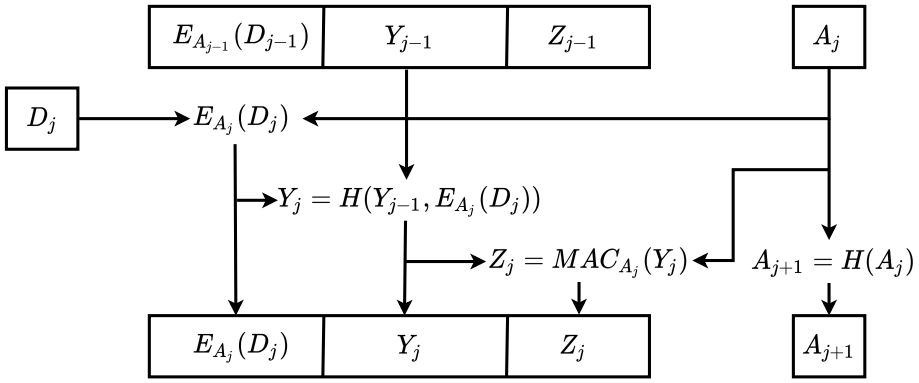

Figure 2.1: A simplified view of the Schneier-Kelsey log scheme

Figure 2.1 demonstrates how the logging scheme works. A_j is the symmetric key used to protect the j:th entry and D_j is the content of the j:th log entry. D_j is encrypted using an encryption function E with the symmetric key A_j. Y is the hash chain field which makes deletion and reordering detectable. Z protects the integrity of the entry with a MAC function, also using A_j as the key.

Due to the usage of symmetric encryption keys, it becomes problematic to delegate verification and read permissions. A verifier (which is different from the trusted machine) could verify that the hash chain is correct, but that does not guarantee that the log has not been tampered with. An attacker could have, for example, edited a log entry and then recomputed the hash chain. This means that a verifier would always need to interact with the trusted machine to verify entries. If the trusted machine wants to give another machine the rights to truly verify the log on its own, it would have to share the initial encryption key. That means that the newly trusted entity could use this power to alter the log in an undetectable way.

2.2 Logcrypt

In 2005 Holt et al. expanded on the work of Schneier-Kelsey and proposed a logging scheme called Logcrypt [16]. They start off by introducing a simplified version of the logging system presented by Schneier and Kelsey. It is pointed out that it is not necessary to encrypt the log entry data. For certain applications, it is important that the log is tamper-evident but the actual contents of the log entries are not considered to be confidential. As a natural next step, the authors introduce the idea of using asymmetric encryption for the log.

Through the use of asymmetric encryption, one of the potential flaws of the Schneier-Kelsey log [14] is fixed. By using a signature instead of a MAC, it makes it possible to delegate the task of auditing/verifying the log to any entity without compromising the security of the log. This is done by using meta entries in the log that records the next n public keys for the n upcoming log entries. After a private key has been used to sign a specific log entry that private key is permanently deleted, which means that an attacker cannot alter the entry. This system can be used with any signature scheme. The fact that the public keys need to be recorded in the log is a downside since it consumes space.

As a remedy to the issue of public keys taking up space in the log, the authors propose using Identity Based Encryption (IBE) [17]. IBE makes it possible to derive public keys based on arbitrary bit strings. So for the logs you could simply use the string "5" to generate the public key for the fifth entry of the log, thus eliminating the need for storing all public keys in the log. IBE schemes require a master public and private key that are used in the process of generating the identity-based public keys and their corresponding private keys. The idea is to generate such a master key pair for the coming n entries, store the public master key in the log, generate the private key to be used for each of the n entries and finally remove the master private key. This means that meta entries are still required but are a lot smaller since only one key needs to be stored per n entries.

The authors recognize the problem of truncation-attacks (tail-end deletion of log entries) and propose a solution that they call metronome entries. It means that a special metronome type entry is added to the log at a certain time interval. If an attacker truncates the log and removes one of the existing metronome entries that would be detectable by a verifier. If the attacker truncates the log but not far enough that a metronome entry gets removed, they have two options. They could allow the next metronome entry to get appended to the log. In that case, it will be detectable by a verifier since that entry will not have used the expected private key for its signature. The private key that was meant to be used for that log entry has already been used and removed from the system. The attacker could also prevent the metronome entry from being added to the log, but that would also be detectable by a verifier.

There are a few downsides to this approach. During the time between metronome entries, a truncation-attack would not be detectable. For that reason, you would want to keep the time interval between metronome entries as short as possible. On the other hand, all metronome entries consume both space and computation power. For this reason you would want to keep the

interval between metronome entries as long as possible.

Logcrypt expands on the Schneier-Kelsey log by creating a log that is publicly verifiable and has a protection against truncation-attacks. Both of these improvements add an overhead to the log which consumes extra space.

2.3 A Log with an Aggregate Signature

In 2009 Ma and Tsudik presented a log design which achieves forward-secure stream integrity [10], meaning that any tampering (deletion, alteration, insertion, reordering) of pre-compromise log entries will be detectable. The difference compared to previous logging systems is that their design also protects against truncation-attacks, without the need for metronome entries.

This is possible through the use of a technique called FssAgg authentication, proposed by the same authors in 2007 [18]. The idea of the scheme is that a single signature or MAC verifies the entire log. The scheme consists of the following four algorithms:

- **FssAgg.Kg** Generates the initial encryption key(s) required for the scheme.
- **FssAgg.Asig** Creates an aggregate signature and updates the current key. Takes in a new log entry to be signed, the aggregate signature up until this point and the current key. Returns the new aggregate signature and the updated private key. The update of the key and the update of the aggregate signature is performed in the same algorithm to provide stronger security guarantees.
- **FssAgg.Aver** Takes in an aggregate signature, the secret key, and an ordered collection of log entries. Returns a boolean indicating whether the signature correctly matches the log entries and the public key.

2.3.1 Scheme and Algorithms

Ma and Tsudik present two versions of the FssAgg scheme, one privately verifiable and one publicly verifiable. The main benefit of the publicly verifiable scheme is that it gives the ability to delegate verification of the log to non-trusted entities. The privately verifiable scheme requires that the verifier is trusted (or semi-trusted if multiple signatures are used). The benefit of using the privately verifiable scheme is that it is much more computationally efficient. Moreover, the public key used to verify the publicly verifiable

FssAgg signature has a size that grows linearly with the number of key evolvements. For the system design (which is presented in chapter 6) it is more reasonable to use the privately verifiable scheme, which will be described in the coming paragraphs.

The scheme assumes the following cryptographic primitives:

- F - a collision-resistant hashing algorithm for k-bit strings, $F : \{0,1\}^k \rightarrow \{0,1\}^k$
- H - a collision-resistant hashing algorithm for arbitrary length strings, $H : \{0,1\}^* \rightarrow \{0,1\}^k$
- h - a secure MAC function $h : \{0,1\}^k \times \{0,1\}^* \rightarrow \{0,1\}^t$ that, on input of a k-bit key x and an arbitrary length message M outputs a t-bit MAC

Let $\sigma_{0,i}$ be the aggregate MAC verifying messages $0, ..., i$. The algorithms are described as follows:

- **FssAgg.Kg**: Generate an initial k-bit secret symmetric key sk_0 (the FssAgg scheme is independent of the symmetric encryption algorithm used)
- **FssAgg.Asig**: A new message M_i is received. The logger has an aggregate MAC $\sigma_{0,i-1}$. First an individual MAC for M_i is created: $\sigma_i = h(sk_i, M_i)$. Next, σ_i is folded into the aggregate MAC through the following operation: $\sigma_{0,i} = H(\sigma_i || \sigma_{0,i-1})$. The individual MAC σ_i is removed and the previous aggregate MAC $\sigma_{0,i-1}$ is replaced by the new one, $\sigma_{0,i}$. Next the key update algorithm, FssAgg.Upd is executed.
- **FssAgg.Upd**: The key is updated through the hash function, $sk_{i+1} = F(sk_i)$. The previous key, sk_i, is deleted from memory which is crucial for the security of the logging scheme.
- **FssAgg.Aver**: To verify an aggregate MAC $\sigma_{0,i}$, you need the initial key sk_0 and the messages $M_0, ..., M_i$. Since the verifier has the initial key sk_0 it can recreate all keys $sk_0, ..., sk_i$ (the hash function used to update the key is known). After that, the verifier tries to recreate the aggregate MAC through the same steps used in FssAgg.Asig. The result is $\sigma^c_{0,i}$. If $\sigma^c_{0,i} = \sigma_{0,i}$ it means that the MAC is valid. If they do not match, it signals that some form of tampering has occurred.

2.3.2 Security and Performance

The most important benefit of applying the FssAgg scheme to a log is making truncation-attacks detectable. In contrast to previously described schemes using a hash chain, the single aggregate MAC makes all forms of tampering detectable, including truncation.

Another benefit of the scheme is that very little resources are required by the logger. The logger only needs to store one aggregate signature and one secret key, a very small overhead compared to other logging systems. Updating the aggregate MAC is also fast, only requiring a constant number of operations. Verifying the log to detect tampering has $O(n)$ time complexity. In order to verify the integrity of the log, a verifier needs to verify the entire log.

The authors suggest combining the FssAgg scheme with MACs or signatures for individual log entries as well. This makes it possible to verify the integrity of a single entry while avoiding the time-consuming process of verifying the FssAgg MAC. It also makes it possible to detect where the integrity compromise has occurred. A failed FssAgg MAC verification only shows that the log integrity has been compromised somewhere.

The FssAgg approach achieves forward-security. Once an entry has been added and the FssAgg MAC has been updated, any tampering with that entry will invalidate the FssAgg MAC, thus making the tampering detectable. No guarantees can be made about log entries added after the logger has been compromised, since the attacker could modify the entries before they reach the log.

2.4 Merkle Tree Log

In 2009 Crosby and Wallach approached the problem of secure logging from a different angle compared to previous research [8]. They argue that you can only discover suspicious activity by actually doing audits on the log, which is of course true. Therefore they take the approach of optimizing the efficiency of verification. This is achieved by making it possible for the logger to create time and space-efficient proofs of correct behavior.

This is possible thanks to a data structure called a Merkle tree [19]. A Merkle tree is a binary tree where the leaves are hashes of some data blocks (in this case, log entries). The parent of two nodes is the hash of the concatenation of the two children's hashes. The result is that a change in a leaf node propagates all the way up to the root of the tree. However, some parts of the tree are left unchanged.

There are three types of entities considered to be relevant in this logging system. The logger itself that is untrusted, clients and auditors. Clients append events to the log. Auditors challenge the logger by asking for proofs, trying to make sure that the logger stays honest.

The logging system consists of the five following algorithms:

- H.ADD(X) $\leftarrow C_j$. A client may send and event X to the logger and the logger returns C_j to the client. C_j is called a commitment and is the root hash of the Merkle tree after adding X, signed by the logger. The index j means that X was added as the j:th commitment which means that X was the j:th entry of the log.

- H.INCR.GEN(Ci, C_j) $\leftarrow P$. The logger creates an incremental proof P which is meant to prove that the events $X_1, X_2, ...X_i$ that are fixed by C_i are consistent with (equal to) $X'_1, X'_2, ...X'_i$ which are fixed by C_j. (C_j is a later commitment than C_i, $j > i$.)

- H.MEMBERSHIP.GEN(i, C_j) $\leftarrow$ (P, X_i). The logger creates a proof of membership for the event i from commitment C_j, ($j > i$). Also returns the event itself, X_i.

- P.INCR.VF(C_i, C_j) $\leftarrow \{\top, \bot\}$. The logger checks whether the incremental proof P claiming to show consistency between C_i and C_j is valid or not.

- P.MEMBERSHIP.VF(i, C_j , X'_i) $\leftarrow \{\top, \bot\}$. The logger checks whether the membership proof P proves that X'_i is fixed as the i:th entry in the commitment C_j.

The first three algorithms are run by the logger on its history tree (Merkle tree) when requested to do so by auditors or clients. The last two algorithms are run by clients and auditors to validate proofs that they have received from the logger.

The purpose of an incremental proof is to show that the new commitment, C_j, is consistent with C_i, a commitment seen previously. In a traditional logging scheme with a hash chain (like Schneier-Kelsey) incremental proofs may require a lot of storage space. The equivalent of C_i and C_j in a hash chain-based log would be the hash chain field and the signature/MAC for the i:th and j:th log entries, (i < j). An incremental proof between these for a hash chain log would require every log entry between them to be included in

the proof, otherwise the hash chain cannot be verified. If $i << j$, this could be very costly.

With the Merkle tree-based log, proofs do not need to include every event between the commitments to show that they make consistent claims about the past. The idea is that you need to show that you can reach the root hash of C_j from the root hash of C_i without changing the hashes of the events that should be equal for both commitments, events $1, 2, ..., i$. The proof is a tree that can be pruned to avoid including all the intermediate hashes. The fact that these proofs are secure is based on the assumption that a collision-resistant hash function (like SHA-256) is used.

The second type of proof is called a membership proof. The incremental proofs are great to show that the logger is making consistent claims about the past, but without membership proofs you couldn't actually be sure that the logger is adding new events the way that it is supposed to. Membership proofs are not only used to verify that new events have been added correctly but they can also be used to confirm that old events are still stored correctly by the logger. The membership proof generator algorithm takes in an event index i and a commitment C_j, $(i \leq j)$. A membership proof is returned together with the actual event X_i. If C_j is trusted to be a valid commitment (thanks to earlier incremental proofs), the membership proof (if valid) proves that X_i is the i:th element of the log fixed by commitment C_j. The membership proof needs to include a path to the leaf node for X_i to prove its location but can prune away many of the other nodes in the tree.

2.4.1 Log Structure

So how is the Merkle tree actually used to create the log? A tree of depth d is initialized and the resulting log can store 2^d entries. The log entries are the leaves of the tree and new entries are added from left to right. If the log gets full it can have its capacity increased in a straightforward way. When the log becomes full the height can be increased to $d + 1$ by making the previous root node the left child of the new root node, effectively doubling the capacity of the log. The authors call their construct a history tree to differentiate from the more general Merkle tree data structure that can be used in many different ways.

Thanks to the nature of the Merkle tree, the root node encapsulates information about every log entry added so far. A parent in a Merkle tree is the hash value created by concatenating the hashes of the two children and running that through the hash function again. Since a tree is filled from left

to right there will be some nodes that are empty. When a concatenation is performed with one of these, an empty string is used instead.

2.4.2 Proof Construction

In figure 2.2 we see a version 2 (three log entries added so far) history tree. Internal nodes are represented as I_i^r where i is the index and r is the row number. The row number is counted down-up so the leaf nodes are at row 0. An interior node I_i^r has the left child I_i^{r-1} and right child $I_{i+2^{r-1}}^{r-1}$.

Figure 2.3 shows a new version, version 6, of the history tree. Comparing figure 2.2 and 2.3 we see that both internal and leaf nodes they are suffixed by one or two ticks (’). This notation is used to show that we do not know at this point if these two history trees make consistent claims about past events. For example, we do not know if $X_2' = X_2''$, but if the log is behaving correctly they should be equal. In figure 2.2 and 2.3 the circled nodes show which nodes are needed to be able to compute the root hash.

Figure 2.4 shows how an incremental proof between version 2 and version 6 can be created. Only the hashes that are circled in the figure are included in the proof. The intuition is that based on the pruned tree in the figure you can compute both the root hash of the version 2 history tree and the version 6 history tree. To recompute the version 2 tree, you can take the pruned tree from figure 2.4 and ignore X_3 and the entire right sub-tree. Then you can recompute the version 6 root hash by actually using all the hashes included in the proof. If both of these hashes match the commitments you have seen for version 2 and version 6 the proof is valid and it is confirmed that $X_0' = X_0''$, $X_1' = X_1''$ and $X_2' = X_2''$. This would show that the logger is behaving correctly.

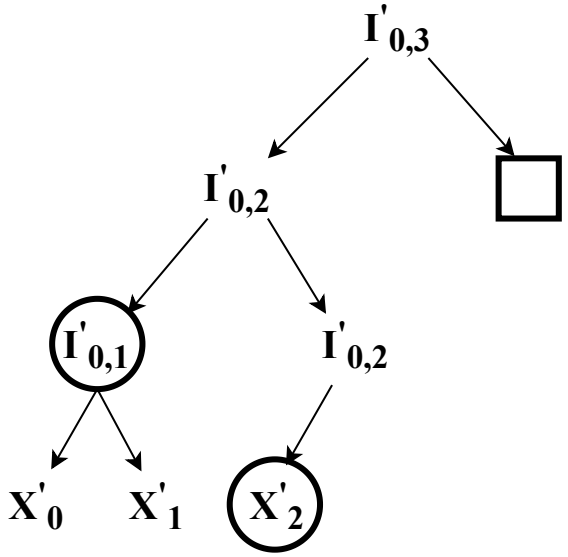

Figure 2.2: A version 2 history (3 added events)

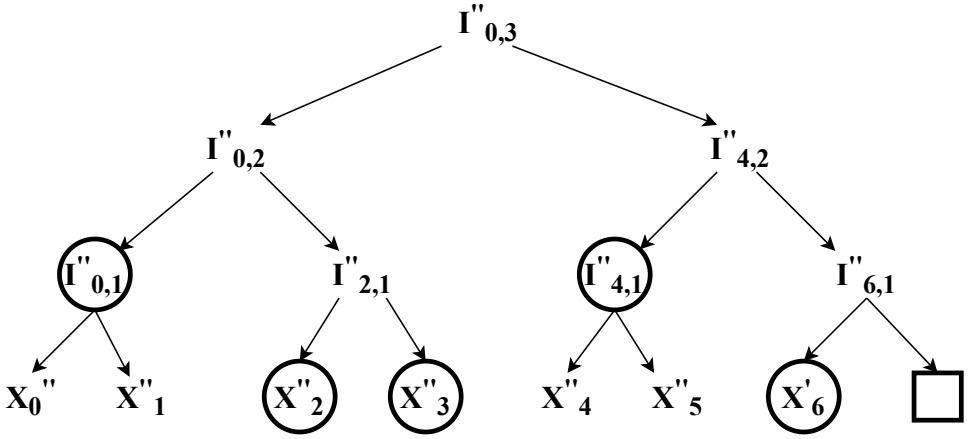

Figure 2.3: A version 6 history (7 added events)

What is worth noting is that the proof didn't need to include all the entries between version 2 and version 6. X_1, X_2, X_4 and X_5 are omitted from the proof. The idea is that parts of the tree that are the same for both commitments can be pruned and replaced by stubs (nodes further up the tree). The example in figure 2.4 is small but one can imagine an example when two commitments C_i and C_j are separated by millions of events. In that case, the benefits in terms of both time and space efficiency become very clear.

The size of the incremental proof is logarithmic to the number of log entries, $log_2(n)$. Membership proofs follow the same concept and also require

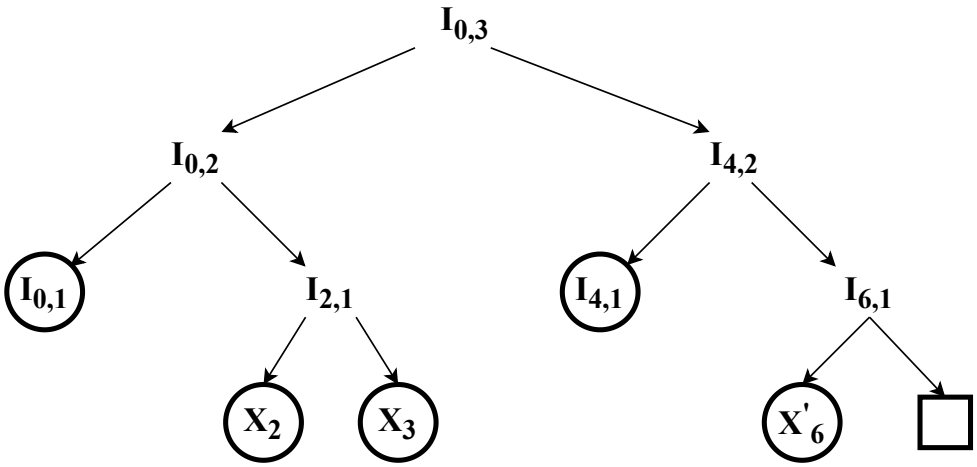

Figure 2.4: A pruned tree that can be used as an incremental proof between the two versions

$log_2(n)$ nodes in the proof. The time required to construct the proofs is also $log_2(n)$.

2.4.3 Auditing, Verification and Gossiping

There are many potential auditing schemes to use together with the history tree logger, which is one of the major strengths of the design. Clients who add a new entry to the log could request a membership proof in conjunction with the adding of the entry. This would make it possible for the clients to detect logger misbehavior very quickly.

It would be possible for an attacker to modify old entries without recomputing the hashes, making the Merkle tree still appear correct. There is no way around the fact that you need to look at a specific entry to be certain that it has not been tampered with, no matter what logging scheme you use. (If you do not have special purpose write-only storage hardware.) To mitigate this it would be wise for auditors to randomly request membership proofs for older events. Another approach is to combine membership proofs with search queries. For example, if a user makes a query (more on queries in section 2.4.5) for all entries added during a certain week, they could thereafter request membership proofs for them to verify that the entries in the query result are valid.

Clients and auditors could also share the information received from the logger with each other through something called a gossip protocol. The purpose of this is to stop an attacker that is controlling the logger from

performing a split-world attack. A split-world attack is when the logger makes inconsistent claims about its state to different clients/auditors. The purpose of this type of attack could, for example, be to fool a specific client without alerting auditors.

The topic of gossip protocols is explored further in [20]. They look for efficient gossip protocols in the Web PKI Certificate Transparency context. Their initial idea is for web clients and web servers to share the most recent commitment that they know of with each other. When an entity sees a commitment that is newer than the newest one they have seen so far, they request an incremental proof and if it is valid they update their stored commitment with the newest one.

If an inconsistency is discovered between commitments it is possible that some sort of split-world attack could be happening and warning messages can be sent to auditors and other entities interested in potential logger misbehavior.

To reduce the need for requesting incremental proofs from the logger, it is suggested gossiping not only the most recent commitment but the two most recent commitments and an incremental proof that proves their consistency. If a client gossips C_a, C_b and $P_{a,b}$ (where $P_{i,j}$ is an incremental proof connecting C_i and C_j), it is possible that the receiver of the message can update their most recent commitment stored without having to contact the logger. This will happen if the receiver stores C_a and the message with C_a and C_b includes a valid incremental proof. In that case the receiver can update their most recent known trusted commitment to C_b without contacting the logger.

This sort of protocol is relevant for the CT [21] (Certificate Transparency) context because the commitments are not updated very often. A new leaf node is not created for every new log entry, instead there is a guarantee that an SCT (signed certificate timestamp) will be included in the CT log within one MMD (maximum merge delay), which is 24 hours. If new commitments of the log were more frequent, the benefits of gossiping incremental proofs would decrease. The reason why CT logs do not create new leaves for every new log entry is to increase the throughput performance of the log.

2.4.4 Performance

The authors of the Merkle tree log paper [8] also created a prototype implementation for their log using C++ and Python 2.5.2. In their benchmarks, they discovered that their implementation could insert 1 750 entries per second in the log; the measurements were made using a total of 4 million inserts. The system running the benchmarks was equipped with an Intel Core 2 Duo

2.4GHz CPU and 4GB of RAM, but the implementation is single-threaded, so the second CPU core was not utilized. The bottleneck was identified to be the signing of commitments. 1024-bit DSA signatures were used. It is claimed that if commitment generation, commitment signing and audit requests are handled by separate CPU cores, the log can insert 38 000 events per second.

Since the generation and signing of commitments are the most time inefficient steps of the entry insertion it is suggested that the logger could avoid creating a new commitment for every entry. The logger would then delay returning the commitment for a new insertion and return a later commitment to optimize throughput. The client could then receive a commitment C_j when it would have received C_i $(i < j)$ if a commitment was created for every entry. The client could still request a membership proof the same way as before, using C_j instead of C_i. This approach improves the performance of the log, but if the client is dependent on receiving a membership before continuing some process, it will harm the performance of the client.

2.4.5 Merkle Aggregation

The paper [8] also shows how it is possible to allow for efficient searching in the history tree through a technique they call Merkle aggregation. The tree structure allows for $log_2(n)$ time access to an arbitrary entry. The idea of Merkle aggregation is to mark nodes with certain properties and propagate that property upwards through the internal nodes in the tree.

For example, certain entries could be flagged as "important". In that case, the parent of that node will also be flagged as important and the process will continue all the way up to the root of the tree, marking all internal nodes that are ascendants of the important leaf node as important as well. When making a query for important entries, one would start at the root of the tree and follow all paths that are marked as important downwards in the tree to find the important leaf nodes. If important nodes are rare, this will allow for very efficient searching compared to a table-based log structure. In a table, you always need to look at every entry to make sure that you have found all important entries. But with Merkle aggregation, the search could be very quick if, for example, only one node in the entire tree is marked as important. This means that the best case search has a time complexity of $O(log_2(n))$. In the worst case (where every log entry matches the query), every node in the tree would need to be traversed in the search process. There is a total of $2 \cdot n$ nodes in the tree, which means that the worst case search is $O(n)$, the same as in a table-based log.

The logger can return not only the leaf nodes matching the query but also the tree containing the paths leading to those nodes. The parts of the full history tree that does not contain matching events would be pruned away and left with stubs. This makes it possible for the client to verify that it has actually received all the events matching the query. This is done by checking that the stubs are unflagged and seeing that the root node corresponds to a trusted commitment.

The example with the important flag is a simple boolean property. However, the Merkle aggregation concept can be used for other types of attributes as well. The authors define a query attribute as a three-tuple containing the type of the attribute, the function used to aggregate the attributes of the children, and a deterministic function that defines how to retrieve the attribute from the entry data. The tuple is denoted by $(\tau, \oplus, \Gamma)$ In the example with the important flag we have τ = Boolean, $\oplus$ = OR (a parent is flagged as important if one or both of the children are) and $\Gamma(x) = x.isFlagged()$. Another example would be entries containing transaction values and using MAX as an aggregation function. This would allow for queries to find all entries with a transaction that exceeds a certain monetary value.

The system is very flexible. To allow for arbitrary keyword searching, one way would be to use bloom filters. The bloom filter would be aggregated with the OR function. This comes with the trade-off where using larger bloom filters requires more space overhead but using smaller bloom filters results in more false positives and slower queries.

Nodes in the history tree have attribute annotation data connected to them to make Merkle aggregation possible. To make attributes tamper-evident just like the log contents, a second hash value is introduced for each node. This hash value is created by hashing the concatenation of the node's hash and its attribute annotation data. This way, attributes gets fixed and propagated up the tree the same way that the log data does.

The introduction of Merkle aggregation requires some small changes to the logging algorithms described previously. It also introduces overhead in terms of space and the amount of overhead depends on the number of searchable attributes. It does, however, enable efficient searching compared to traditional table-based logs.

2.4.6 Safe Deletion, Pruning and Exporting

Queries can also be used to perform deletions in a way that is safe. A proof can be generated to show that only log entries matching the deletion query were removed. Stubs are left where entries have been deleted. In the new pruned history tree, it will not be possible to generate incremental or membership proofs involving commitments that were deleted. I.e., if entry X_i was deleted, you can no longer generate proofs involving C_i. But proofs not involving deleted entries will stay valid.

Safe deletion can be combined with a preceding query to enable exporting of a subset of log entries. Events older than a certain date could be exported and removed in order to save space on the machine where the logger is running. Thanks to the safe deletion functionality, you could also prune events on criteria other than age without causing issues with auditing. For example, you might want to remove all entries made by a certain client that is no longer relevant. With a hash chain-based solution, this would not be possible since it would cause problems when trying to verify the hash chain.

Since the result of a query can include a pruned history tree leading to the results of the query, the query result is a history tree with the same capabilities as the full history tree. This means that an exported section of the tree that is removed from the logger can later be verified in the same way as the original log.

2.4.7 Summary

In the logging systems described in older research papers, the goal is to achieve forward security. This means that pre-compromise records can not be modified in an undetectable manner. The history tree-based approach tries to reach something stronger by making it difficult for the logger to tamper even with post-compromise records. In a scenario where the attacker has full control over the logging machine but the logger is constantly being challenged to prove its honest behavior the attacker who has compromised the logger will have a hard time performing undetected tampering. The best bet for the attacker is to fork the log by tampering with the log while trying to show a facade of correct behavior to clients and auditors demanding proofs. This becomes increasingly difficult with time and with a large number of honest clients and auditors.

The history tree with the Merkle aggregation technique provides great benefits compared to the table + hash chain-based designs presented in previous sections. The history tree enables efficient auditing, efficient querying and flexible capabilities for pruning and exporting entries. The trade-

off is that adding new entries to the logger is slower. Inserting a new entry to the history tree requires $O(log_2(n))$ time compared to $O(1)$ time for a table-based log. The tree-based structure also requires some additional storage overhead which is the tree consisting of $2 \cdot n$ nodes.

2.5 Certificate Transparency

RFC 9162 [21] published in December of 2021 describes version 2.0 of the Certificate Transparency system. The CT system is heavily influenced by the Merkle tree log[8] which was described in section 2.4. The purpose of the CT system is to make it possible to notice if a CA has "gone rogue" and started to issue faulty certificates like in the case of the DigiNotar hack in 2011 [22]. CT is a logging system where all TLS server certificates used in the Web PKI are logged. When a CA requests to add a certificate to a CT log, an SCT (Signed Certificate Timestamp) is returned from the CT log to the CA. The SCT is added to the certificate as a token. Both Safari and Chrome browsers require a minimum of 2 SCTs to consider a certificate valid [23]. The CT log promises to add the certificate to the log within one MMT (Maximum Merge Delay), which is normally 24 hours. Many certificates are bundled together in leaves of the Merkle tree-based CT log. The original Merkle tree log presented in [8] assumes that each new log entry corresponds to a new leaf in the Merkle tree. The reason CT bundles entries together is to increase throughput.

CT generally serves a different purpose than Certificate Revocation Lists (CRL). The purpose of CRLs is to check if a certificate has been revoked. A certificate would be revoked if the entity to whom the certificate is issued to misbehaves in some way. E.g. if the entity loses control of their private key they would ask for the certificate to be revoked to avoid any abuse. CT on the other hand is used to detect misbehavior of the CAs themselves. There are monitors that are always checking the logs to discover malfeasance. For example, this makes it possible for a company like Google to quickly notice if a CA has issued a certificate for one of Google's domains when it shouldn't have done so. This allows breaches like the one in the DigiNotar hack to be discovered quickly.

The Online Certificate Status Protocol (OCSP) is in most cases used to check the revocation status of certificates, just like CRLs. However, in some circumstances the previously mentioned SCTs are retrieved by using OCSP instead of having the SCTs embedded in the certificate itself. This can be useful if the CA wants to be able to deliver the certificate before receiving the SCT from the CT log. [24]

2.6 Summary

Table 2.1 shows a comparison of the different logging schemes presented in this chapter. Other than the Merkle tree log, the other papers do not explicitly mention membership or incremental proofs. If one would like to prove the position and integrity of log entry j in a hash chain-based log, one would need to verify the hash chain from the start of the log until entry i, which is why the time complexity is $O(i)$.

For the equivalent of an incremental proof proving the consistency of C_i and C_j for a hash chain-based log, you would need to verify the hash chain between entries i and j, resulting in a time complexity of $O(j - i)$. For the FssAgg log it would be possible for an auditor to perform a verification equivalent to an incremental proof if they store an FssAgg MAC/signature previously received from the logger and use it to compute a later FssAgg MAC/signature. This also results in a time complexity of $O(j - i)$.

The Merkle tree-based log is not inherently resistant to truncation-attacks. The security of the log depends on there being honest clients and auditors continuously requesting proofs. If auditors and clients are compromised or inactive in their requesting and verification of proofs, it would be possible for an attacker that has compromised the logger to truncate the log for commitments that clients or auditors are not verifying in an honest manner.

Chapter 3

Methodology

3.1 Identifying the Problem and Requirements

An important task was to figure out and specify the requirements of the new logging system. Discussions with PrimeKey were an important part of figuring out what EJBCA customers are not happy with regarding the audit log. Reading the EJBCA documentation [25] and exploring the existing EJBCA audit log were also important steps in understanding the current implementation. The result of this process is a summary of the current audit log solution in chapter 4 and a specification of requirements in chapter 5.

3.2 Literature Review

To be able to propose a new system design, it was important to learn about the current state-of-the-art research on secure logging. During the time of the literature review, the requirements for the new log system were taken into consideration in order to identify the research papers with solutions matching the requirements. The state-of-the-art research is summarized in chapter 2 where the papers that are the most relevant for the solution are described in detail.

3.3 System Design

A system design was proposed based on the information retrieved in the literature review to satisfy the requirements as well as possible. Not only the functionality of the log is described but also the behavior of the clients

and auditors interacting with the log. A scenario based on a real use case for EJBCA (passport certificates) is used as an inspiration for the design since it is a scenario with a high emphasis on security and multiple entities interacting with the log. The log design is evaluated based on the requirements. The design is described in chapter 6.

3.4 Implementation

A proof-of-concept implementation was made to serve as an example for how the log, clients and auditors could be implemented. The implementation also made it possible to run performance metrics to see how the log performs compared to the individual logging schemes that this design is based on. The implementation and the performance metrics are presented in chapter 7.

Chapter 4

Existing Solution

EJBCA is an open-source PKI software that is available under an LGPL (Lesser GNU General Public License) license [2]. The enterprise edition of EJBCA is available under a paid subscription model. EJBCA allows for certificate lifecycle management and a flexible way of setting up a PKI for the user's needs. An example of this is a PKI set up to allow for encrypted emails within an organization. But the PKI set up with EJBCA could also be used on a larger scale, such as national passport systems where physical passports have corresponding digital certificates.

The main purpose of the audit log in EJBCA is to preserve information about important events that have happened in the PKI. There is also a system log that logs all possible kinds of events with less emphasis on security. Examples of important events are "Certificate issued", "Certificate Profile edited" and "Administrator accessed resource" [26].

Another property of the audit log is to provide non-repudiation. If a CA has issued a certificate, undeniable proof of that action should be available in the audit log. For a malicious actor (insider or outsider), the audit log is a valuable target. An attacker could want to hide the traces of some action that has been performed by a CA.

4.1 Integrity Protection

EJBCA Enterprise Edition offers integrity protection to increase the security of the log. The CA creates a signature for a log entry using its private key before adding the entry to the log. One of the fields in the log entry is a sequence number which is a number that a CA node increments by one for every log entry it adds to the audit log [27]. The combination of the

signature and the sequence number creates a total ordering of log entries. This means that if a log entry is modified, deleted or entries are reordered, the tampering will be detectable by an auditor that tries to verify the log. However, there is a caveat. If a malicious actor deletes entries from the end of the log, the tampering will not be detectable since the action will not cause any visible problems with the sequence numbers. In other words, the current implementation is vulnerable to truncation-attacks.

If the malicious actor deletes log entries that has already been seen by a verifier the tampering will be detectable. The CA nodes keep their most recently written sequence number in memory, so if a CA node adds new entries after a malicious truncation-attack has been performed, the tampering will become detectable. If a CA node goes offline, it will read its most recently written sequence number from the audit log table in the database. This combination means that an undetectable truncation-attack is a possibility.

When audit log entries are fetched from the database, their individual signatures are automatically checked, but the sequence numbers are not. This means that modifications of log entries are detectable when database queries of audit log entries are made. To check sequence numbers which allow for detection of reordering and deletion of log entries, a special tool included with EJBCA EE needs to be used.

This tool has an "all-or-nothing" approach which means that it checks that sequence numbers are correct for every entry in the audit log table when it is run. This is something that is causing issues for EJBCA customers with very large audit logs since verification of all sequence numbers takes a long time.

4.2 Clustering

If the PKI is used at a small scale, there will probably be only one CA node handling the issuance of certificates and other actions. In some cases, high levels of throughput for CA actions are necessary. In that case, multiple CA nodes are run in a cluster where each CA node has its own database instance. These database instances are continuously synchronized to provide a consistent view for all CA nodes. The public/private key pair used for signing audit log entries (when integrity protection is enabled) can be either unique for each CA node or the same keypair can be used by all of them. Unique key pairs offers greater security but requires configuring the nodes so that they can verify signatures that they did not create themselves.

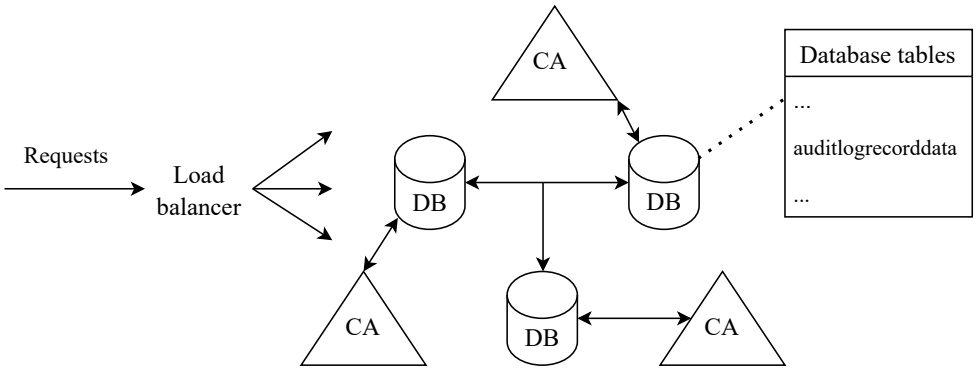

Figure 4.1: An overview of how the current EJBCA system works when run in a cluster

In figure 4.1 we see EJBCA run in a cluster with three CA nodes. CA nodes share the load of incoming requests. Each node has its own database that they communicate with. The databases continuously synchronize to be consistent with each other.

4.3 Implementation and Performance

EJBCA is implemented in Java and supports integration with multiple SQL databases such as MariaDB and MySQL [28]. When the CA performs an action that is required to be logged as an event in the audit log, atomic transactions are used. The result is an "all-or-nothing" effect where the action is only performed if the entry can be added to the audit log and the entry is only added to the audit log if the action is performed.

In the current implementation, the audit log is a table in the database used by the CA. The columns of the "auditrecorddata" table are:

1. **pk**: Primary Key, uniquely identifies the row in the table
2. **additional details**: Event specific message with additional information
3. **auth-token**: Identifies the administrator or internal module that caused the event
4. **customId**: Identifier used in log messages, commonly the certificate authority an event was related to
5. **eventStatus**: Either success or failure describing if the event that generated the log entry was executed successfully

6. **eventType**: The type of audit log event that occurred
7. **module**: The internal EJBCA module the event was generated from. This is useful for categorizing events. Examples of modules are: ADMINWEB (shows that the event was generated by an action in the admin web GUI) and CERTIFICATE (shows that the event was generated by the certificate issuance and handling module).
8. **nodeId**: The ID of the CA node which generated the event
9. **rowProtection**: The signature of the log event (only used when integrity protection is activated)
10. **rowVersion**: A column showing how many times a row has been edited. This column is present in all tables but is not relevant for the audit log table since rows should never be modified.
11. **searchDetail1**: Detail used in log messages, commonly the serial number of the certificate
12. **searchDetail2**: Detail used in log messages, commonly the username an event was related to
13. **sequenceNumber**: The incremental number for the event (separate for each EJBCA node)
14. **service**: The service the event originates from, can be either EJBCA or CORE. CORE means that the event was generated from software in the core project which is shared among multiple PrimeKey products.
15. **timeStamp**: The time in milliseconds since epoch that the event occured

When making queries in the database, a CA node becomes occupied with the task of performing the query. This is not great since it decreases the throughput capabilities of the CA. For very large audit logs, query requests can also time out, simply because there are too many entries to search through.

4.4 Areas of Improvement

The existing EJBCA implementation has some flaws that customers have complained about. The process of verifying the integrity of the log is slow, and the only available verification method requires verification of the entire

log. Seaching in the log is also inefficient and can cause the CA node to time out and prevent it from performing other action.

Customers have also asked for exportation capabilities where a part of the log is exported to save space, while preserving the ability to verify the integrity of both the original log and the exported part of the log. A new log design should aim to satisfy these wishes from customers and improve security, especially regarding the vulnerability to truncation attacks.

Chapter 5

Analysis and Requirements

It was decided early on that the new log design would have the logger as a separate machine in the system, decoupled from the CA nodes and their database. Effectively this means that the "auditrecorddata" table (described in section 4.3) is removed from the database and the logger is a separate entity in the system. This makes reasoning about security easier. It also provides the benefit that queries to the log could happen without disrupting the operation of one of the CA nodes. In practice, the audit log may be on the same machine as one of the CA nodes, but it would be a separate process.

5.1 Threat Analysis

The ultimate goal of the audit log design is to make it append-only and that any kind of tampering with old events is detectable. Spoofing in order to add entries to the log without permission is a potential threat, but it is probably not a big one, and it is not difficult to solve.

The main threat that is considered is an attacker compromising the logger. If an attacker has gained access to the logger, their primary goal is probably to hide some malicious activity. For example, they might have issued a certificate but they want to hide any tracks that could trace back to the performed action. Note that this attacker could be an insider. Nobody should be allowed to delete or edit log entries, no matter what privileges they have.

An audit log is never better than its latest audit. Tampering cannot be detected unless some auditor actually examines the log. This means that the more often audits happen, and the more extensive they are, the better. A possible scenario is that the attacker doesn't care about being discovered and just deletes the entire log or a large part of it. There are two potential remedies

to this. One is special-purpose append-only hardware, but that is outside the scope of this work. The other is some form of redundancy of the data. The simple option would be backups of the log done at regular intervals, stored on other machines (the more the better). The focus of this project is to achieve the tamper-evident property, to make tampering detectable. The topic of log data redundancy is not explored further.

5.2 Security Requirements

5.2.1 Entity Authentication

The logger needs to check where incoming entries are coming from. Only known CA nodes should be allowed to add new entries to the log.

5.2.2 Tamper-evident

It should be detectable if an attacker were to remove, alter or reorder existing log entries. Both integrity and authenticity are important for the audit log. This is achieved through the use of signatures and sequence numbers in the existing audit log implementation.

The log system should be **Truncation-attack resistant**. The current EJBCA audit log implementation and the Schneier-Kelsey logs [14] are both vulnerable to truncation-attacks, that is when an attacker deletes a continuous subset of tail-end log messages in an undetectable way. For a log to be considered tamper-evident it should be resistant to truncation-attacks.

5.3 Functional Requirements

5.3.1 Writable from Multiple Sources Concurrently

This is a requirement from PrimeKey and it is more dependent on the implementation itself rather than the design of the log. To solve this, the logger would need to be able to handle requests to add new entries from multiple sources concurrently. The procedure of actually updating the data structure containing the log data would not need to support parallelism for this requirement to be considered fulfilled.

5.3.2 Scalability

The new log design should make it possible to perform search queries in a more efficient way than previously. While still providing good throughput performance in terms of entry insertions. In the current table-based log, searching is always $O(n)$.

5.3.3 Pruning

It should be possible to prune the log, for example by deleting old events and exporting them. In the current solution, log entries that have been exported cannot be verified later. The new solution should have this capability.

Chapter 6

Our Log System Design

The new log solution is based on the Merkle tree log presented in section 2.4 with an addition of an FssAgg MAC as they are presented in section 2.3 to further improve security. The Merkle tree-based log has many benefits compared to the current EJBCA audit log. It gives PrimeKey customers a lot more flexibility in choosing their own way of performing audits, both internal and external. It also enables more efficient searches thanks to the Merkle aggregation technique described in section 2.4.5.

When designing the logging system, inspiration was taken from a real-world use case where EJBCA is used by a company to provide a passport service on behalf of a government agency [29]. Based on this setup, there are three different types of entities interacting with the log.

The *clients* are the CA nodes that are adding entries to the log for the auditable actions it performs, such as issuing a certificate.

The second entity type is *frequent auditors* which has active communication with the logger to request proofs at a high frequency to force the logger to prove its correct behavior. This type of auditor is likely run by the company internally to detect intrusions or insider misbehavior.

Finally, there are *infrequent auditors* which is an entity outside the company which does not have an active connection to the logger but instead performs audits more rarely, maybe at random times. In the passport scenario, this would be the government agency that wants to see the company is behaving correctly and also wants proof that the log has not been tampered with.

The expected use case is that the EJBCA customer is running the CA, the logger itself and also has at least one frequent auditor set up. Potentially there are also infrequent auditors, run by other third-party entities. Most of the reasoning about security is based on this use case. However, the log design

is flexible and can be used in other scenarios. For example, there could be use cases where the logger itself is run by some other entity than the EJBCA customer.

6.1 Logger Design

The logger itself works a lot like the Merkle tree-based log described in [8] and 2.4.

Some of the functional requirements are satisfied by the design of the Merkle tree log:

- **Scalability**. Thanks to the Merkle aggregation technique, searching can be done more efficiently than in a table based log, with a better time complexity in the best case, as shown in table 2.1. Log entry insertion is $O(log_2(n))$.
- **Pruning**. The Merkle tree structure makes pruning easy and flexible. In the current solution, the only possible pruning would be to remove a certain number of the oldest events. Anything more advanced would cause verification problems. With the Merkle tree, you can prune and export based on any query and create a proof that only entries matching that query were removed. The exported part of the log is also verifiable in the same way as the main log.

Only verified CA nodes should be allowed to add entries to the log. The logger needs to verify the identity of the client who sends the entry. This can easily be achieved due to the fact that the CA nodes sign each individual log entry. This is how it works in the current implementation and that signature will remain. This means that the logger needs to check that the log entry received has a valid signature from the CA node that sent it. This satisfies the entity authentication requirement.

In terms of being tamper-evident, the Merkle tree log shows the strongest properties out of all the log designs examined in this project. As opposed to the forward security property of other log designs, the Merkle tree log makes it difficult for an attacker that has compromised the logger to perform any tampering undetected. This is based on the assumption that there are honest clients and auditors requesting proofs. The addition of the FssAgg MAC in this design makes undetected tampering even more difficult for an attacker to achieve. The FssAgg MAC can make tampering detectable even if someone

who controls the clients, frequent auditors and logger tries to truncate the log. More on threats in section 6.5.

In the passport scenario described previously, there are frequent auditors, CA clients and the logger itself, all run by the same organization. There is an infrequent auditor who wants to check the audit log for malfeasance on more rare occasions.

In this scenario, if the organization makes a mistake that would cause them to lose their government assignment, they might be tempted to roll back the log and hide that the event has occurred. Since the organization controls the logger, the frequent auditors and the clients, the organization can tamper with the log. They could delete entries from tail-end of the log and recompute a previous commitment. As long as this commitment does not pre-date the commitment most previously seen by the government auditor, the tampering will not be detectable.

Therefore FssAgg MAC [18] is introduced to further increase the tamper-evident strength of the log.

If the FssAgg MAC is used to verify all log entries, many of the benefits of the efficient proofs enabled by the Merkle tree structure are lost. Verifying an FssAgg MAC protecting n log entries requires $O(n)$ time. Therefore the FssAgg MAC is used to protect only the tail-end of the log. This gives the best of both worlds: the efficient proofs of the Merkle tree log and protection from truncation-attacks from the FssAgg MAC.

Figure 6.1 gives an overview of the full system. The behavior of the different entities shown in the figure are described in more detail in the upcoming sections.

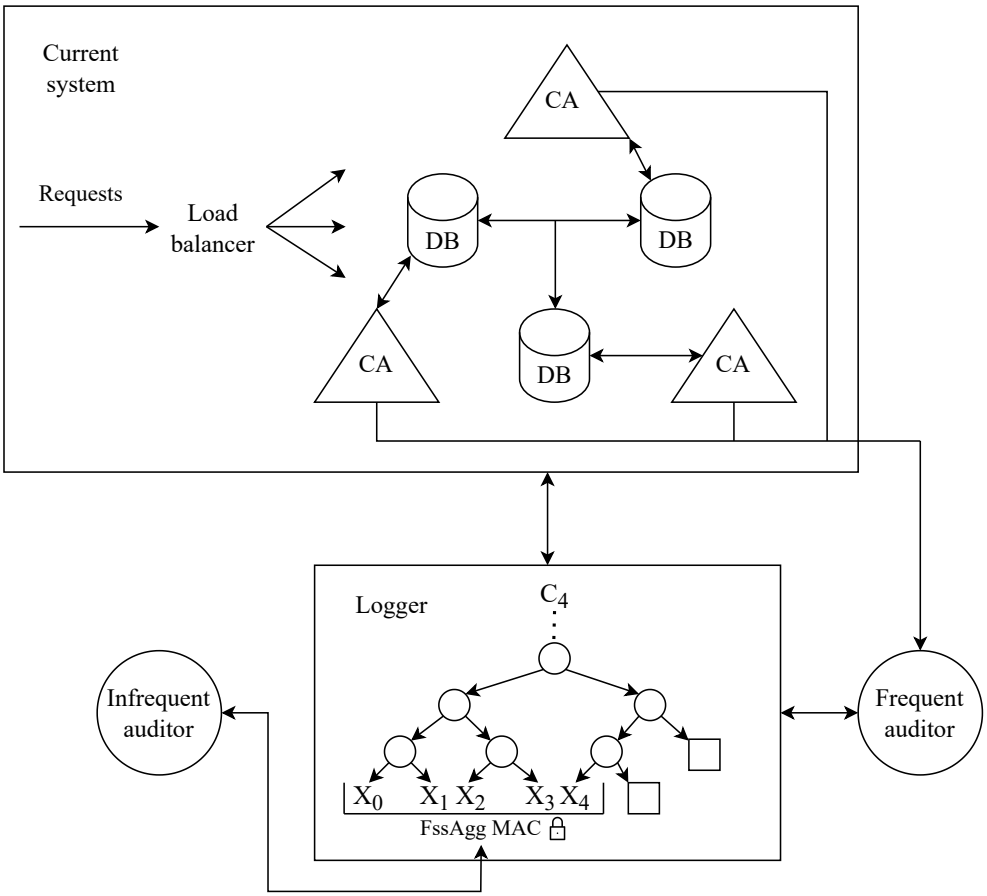

Figure 6.1: An overview of how the new logging system works incorporated with EJBCA. In the example the log is at commitment 4, since five entries have been added.

6.2 Infrequent Auditor Design

In the scenario with the infrequent government auditor, the benefit from the FssAgg MAC comes from protecting the tail-end of the log during the time in-between the audits. When there is an FssAgg MAC authenticating the entries at the end of the log it means that even when the frequent auditors, clients and logger are under the control of the same entity, they cannot truncate the log without invalidating the FssAgg MAC.

The trusted infrequent auditor needs to agree with the logger on an initial secret key. If there are multiple third-party auditors they may have separate FssAgg MACs or share the same one. Having separate MACs is more secure

but requires more overhead.

The fact that the FssAgg MAC protects from truncation assumes that the logger was behaving correctly when the event that the attacker wants to hide occurred. For example, if an attacker compromises the logger, they could save a snapshot of the FssAgg MAC. This would make it possible to roll back the log to a previous state in an undetectable way.

The following scheme is used to initialize an FssAgg MAC:

1. The logger generates a symmetric key and sends it to the trusted infrequent auditor(s).

2. The logger creates a meta entry stating that a new FssAgg MAC has been generated, generates the initial FssAgg MAC based on the meta entry and the initial key. The key is then evolved through the hash function and the initial secret key is deleted from the logger.

When a new entry arrives, the logger runs the FssAgg.Asig algorithm as described in 2.3. The first time the infrequent auditor wants to verify the log, say at commitment C_i ($i + 1$ entries have been added) it will run the FssAgg.Aver algorithm to check that the FssAgg MAC is correct with respect to entries $0, ..., i$ and the initial secret key sk_0. The infrequent auditor will remember the FssAgg MAC, $\sigma_{0,i}$ and the latest signed Merkle root hash C_i, which is also sent from the logger.

When it is time for the next audit, the infrequent auditor will make a request to the logger and receive $\sigma_{0,j}$ and C_j, $j > i$. The logger will provide an incremental proof connecting C_i and C_j. This will allow the auditor to verify that the current version of the log is consistent with the version of the log that was seen during the last audit. The auditor will also verify $\sigma_{0,j}$. When recreating $\sigma^c_{0,j}$, the auditor will start from $\sigma_{0,i}$ which was stored from the last audit.

This approach of reusing previously verified FssAgg MAC:s at the side of infrequent auditor saves time and is reasonable since the purpose of the FssAgg MAC is to show that there has been no truncation of entries from the end of the log. If the incremental proof and the FssAgg MAC are both valid, this shows that the log is consistent with what was seen previously and no truncation has occurred. The time complexity of the audit is $O(log_2(n)+(j-i))$. The process repeats for all upcoming audits from the third-party infrequent auditor.

The fact that it is possible to verify a new FssAgg MAC based on a previously computed and verified FssAgg MAC is not something that is mentioned in the original FssAgg paper [10]. The authors likely assumed that

there wouldn't be any reason to use this approach since it does not verify the integrity of the entire log. But, when incremental and membership proofs are available it is more efficient to reuse previous computations, since the purpose of the FssAgg MAC is to protect the latest entries in the log.

6.3 CA Node Design

The logger will be separated from the rest of the CA system, which is different from the old solution. Since the audit log now runs separately, it becomes difficult to guarantee atomicity of actions. In the old solution, there is an "all-or-nothing" approach. For example, when a new certificate is issued it is guaranteed the corresponding event is added to the audit log. This is achieved by using atomic database transactions. If there is a problem that stops the event from being written to the audit log, the entire transaction is aborted and the certificate wouldn't get issued. This is easy with the old solution since it all happens within the same database.

With the new design, a different approach is suggested (the log algorithms are described in section 2.4):

1. The CA node gets a request to issue a certificate, creates the audit log entry X and sends it to the logger by using the function H.ADD(X).

2. The CA node receives a commitment, C_k, from the logger. The CA node remembers this commitment. If the CA doesn't receive such a commitment within a specified time limit, the entire process is aborted.

3. The CA node requests a membership proof for event k on commitment C_k from the logger, H.MEMBERSHIP.GEN(k, C_k) $\leftarrow$ (P, X_k). The entire process is aborted if this request times out or if the proof is invalid. The CA node confirms both the proof and that the log entry data received together with the proof is identical to what the client originally sent.

4. The CA nodes requests an incremental proof from a previously remembered commitment C_j to C_k, H.INCR.GEN(C_j, C_k). The entire process is aborted if this proof is invalid or the request times out. If the proof is valid, C_k replaces C_j as the last known commitment for this CA node.

5. The CA node has confirmed correct behavior from the logger and proceeds to release the certificate.

6. The CA node gossips the new commitment C_k to known frequent auditors.

The process above shows that the CA nodes serve both as clients and auditors of the logger. When the CA nodes adds a new event to the logger, it confirms that the entry has been added properly by requesting and verifying a membership proof. But the CA node also performs additional auditing by requesting an incremental proof from the commitment it received the last time it added an entry to the log. This way, the CA node confirms that the current state of the logger is consistent with the commitment it received the last time the CA client added an entry to the log.

For the event of issuing a new certificate, it is critical that the event is added to the audit log. Otherwise, there is a risk of unknown rogue certificates out in the wild wreaking havoc. For other events, it is not as critical that the event has to be added to the audit log to allow the action to be performed. Therefore it may not be worth it to wait for confirmation from the logger before performing the action. In some cases, it may even be crucial that you can perform the action even if the logger is unavailable for some reason. An example of this is certificate revocations.

6.4 Frequent Auditor Design

The CA clients already serve the dual role of being both clients and auditors, but dedicated frequent auditors (at least one) are needed to prevent split-world attacks and to detect tampering with old events.

It is possible for an attacker controlling the log to change the data of a log entry without making changes to the hashes in the tree. This means that it would seem that the log is behaving correctly when only considering incremental proofs. The only way to detect such tampering is to actually look at the log entry itself. Tampering will be detectable in two ways, the signature for the individual entry created by the CA will be invalid, and the hash of the entry itself, which is present in the Merkle tree, will not be correct.

To be able to detect this kind of tampering the frequent auditor randomly selects an entry, using a uniform random distribution, and requests a membership proof for this entry. If the most recent known commit is C_m a random event from 0 to m will be chosen. If the membership proof is valid the entry has not been tampered with. The more often these random membership proofs are requested, the more likely it becomes to detect tampering. But to avoid overloading the logger, they should be requested at a set time interval.

The best interval could be discovered through experimentation; it can differ between setups.

The second task of the frequent auditor is to request incremental proofs based on gossip from CA clients. The CA clients send new commitments that they receive from the logger to the frequent auditors. If the commitment received by the frequent auditor is newer than the most recent commitment it has seen, the frequent auditor requests an incremental proof between its most recent known commitment and the newly received commitment. The purpose of this scheme is to prevent a split-world attack where a malicious logger shows inconsistent information to different clients and auditors.

All the auditing that has been discussed so far is about detecting the tampering on the log and misbehavior by the logger. It is also likely that the infrequent auditor will be examining the log entry data itself, not only verifying that it has not been tampered with. The exact rules of what events are allowed or considered suspicious differ widely from application to application. The main focus of this project is to create a log that is tamper-evident, therefore the topic of log entry inspection will not be explored further.

6.5 Secruity Analysis

The security of the log is based on the combination of having honest clients and frequent auditors challenging the logger for proofs and the FssAgg which provides stable forward-security.

6.5.1 Adversarial Model

From the perspective of the organization that is running the CA, there are internal and external attackers that could compromise one or more parts of the audit log system. The internal and the external attacker would share the same goal; to hide traces of some events.

The third-party infrequent auditor sees the entire organization as untrusted. If the audit log system is compromised it does not matter to the infrequent auditor if it is an internal or an external attacker performing the attack. The FssAgg MAC makes sure that the log is forward-secure from the perspective of the infrequent auditor. All entries created before the log is compromised is protected by the FssAgg MAC, even the entries that are new since the last audit.

6.5.2 Logger

If the logger itself is compromised, there is not much that the attacker can do if there are honest frequent auditors and honest clients demanding proofs from the logger.

The likely scenario would be that an attacker creates a certificate via one of the clients i.e. the CA nodes (perhaps with the help of an insider) and tries to hide information about its existence from other auditors and clients. It would still be necessary to show the correct membership proof to the CA node which created the certificate, otherwise the certificate cannot be created in the first place.

This is an example of a split-world attack where one CA node gets information about the certificate, but that information is hidden from other entities. Due to the fact that CA clients gossip their recently seen commitments to known frequent auditors, this kind of attack will be detected, assuming there is a frequent auditor running and ready to receive gossip. In other words, if the compromised logger sends commitments that are not consistent with each other to different clients (CA nodes), this will get detected if there is at least one honest frequent auditor that the CA nodes can gossip to.

However, if the logger and the frequent auditors are compromised, an undetected split-world attack would be possible. In this case one CA node can issue a certificate while the information of this does not reach any other entity. Gossiping is required to prevent split-world attacks.

6.5.3 CA Node

If a CA node gets compromised in such a way that the attacker is actually able to alter the execution flow of the program the security of the CA breaks down. For example the attacker could bypass the requirement that the CA node receives a valid membership proof and a valid incremental proof for a certificate creation event from the logger before actually releasing the certificate.

One way to prevent this would be to force the commitment from the logger to be included as a part of the certificate. But that would mean that the audit log would have to be accessible by not just CA nodes but also users of the certificate in order to verify the commitment. The audit log is not meant to be a public log in that way. Instead something like the public Certificate Transparency logs could be used to achieve that goal.

Even though this type of attack would compromise the security of the CA as a whole, the audit log itself would still be tamper-evident and append-only.

6.5.4 Logger, Frequent Auditors and Clients

This is the special scenario described earlier where an organization could try to hide misbehavior by changing the behavior of everything they control internally. Without the FssAgg MAC this type of compromise would allow for undetectable truncation of the log. But due to the presence of the FssAgg MAC, truncation is detectable by an outside infrequent auditor even if the logger, frequent auditors and clients are all dishonest.

If old FssAgg MAC data is properly removed after each evolvement, after-the-fact undetectable tampering is impossible. If the organization plans to perform some illegal action beforehand, they could alter the logger to save previous states of the FssAgg MAC allowing them to roll back the log to a previous state in an undetectable manner.

Chapter 7

Implementation

A proof-of-concept prototype of the design presented in chapter 6 was created to show how the logging system can work in practice and to give PrimeKey a basis for implementing the proposed logging system in EJBCA. The implementation was created using Node.js [30] and is available on Gitthub[1].

A fork of the npm package merkletreejs[2] [31] is used to provide the Merkle tree functionality for the logger. The fork was created for this project since the default merkletreejs implementation for adding a leaf to the tree unnecessarily recomputed all the hashes in the tree, even though only $O(log_2(n))$ hash computations are necessary. The FssAgg MAC was implemented using hash and HMAC functions from the standard Node.js libraries crypto [32] and crypto-js [33]. Signed log commitments are also implemented. A commitment consists of the Merkle tree root hash, the current index when the commitment was created (showing how many entries had been added at that point) and a signature. The signing of commitments is only enabled if you provide a private key object to the constructor when creating a log object.

The repository contains not only a proof-of-concept implementation of the log but also an implementation of an HTTP server which can be used to initialize and interact with the log. The server exposes the following API:

- `/addEntry`
- `/getProofByIndex`
- `/getProofByEntry`
- `/addEntryAndGetProof`

[1] https://github.com/mans-andersson/Merkle-FssAgg-log
[2] https://github.com/mans-andersson/merkletreejs

- `/getEntry`
- `/getEntries`
- `/getCommitment`
- `/initializeFssAggMAC`
- `/getFssAggMAC`

In the system design presented in chapter 6 above, the most commonly used endpoint by clients would be `/addEntry` and `/addEntryAndGetProof`. A third-party infrequent auditor would use `/initializeFssAggMAC` to start a new FssAgg MAC and would use `/getFssAggMAC` when it is time for an audit. The use of a HTTP server is one way to fulfill the functional requirement that the log should be writable from multiple sources concurrently. In this case, the HTTP server would have a queue that accepts requests from different CA nodes concurrently. Then the elements are taken from the queue in a FIFO manner and added to the log.

7.1 Performance

Some performance metrics were created to show the performance of different actions in the logging system and to see how the combination of the Merkle tree log and the FssAgg log compare to their standalone schemes. The tests were run on a system equipped with a 3.1 GHz Dual-Core Intel Core i5 CPU and 8 GB RAM; the implementation is single-threaded.

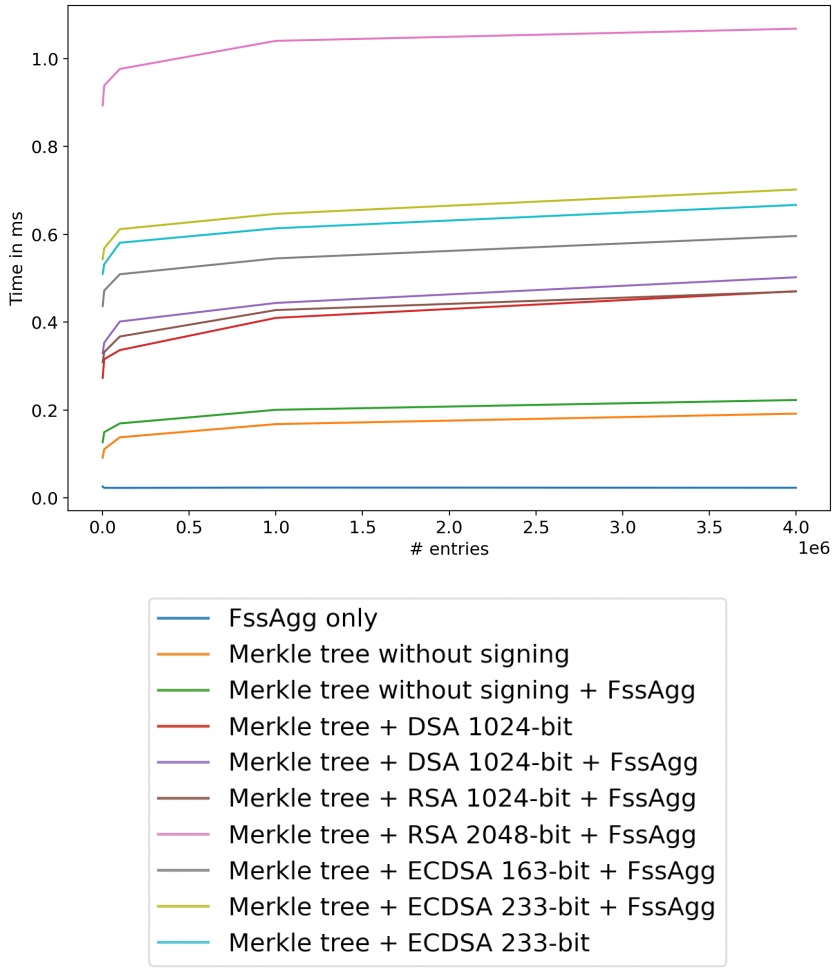

Figure 7.1: Plot of the average time required to add new entries to the log for different logger setups and signature algorithms

Figure 7.1 shows a comparison of the average time required to add a new log entry for different logger setups and log sizes. Combining the Merkle tree based logging with the FssAgg scheme only slightly decreases the throughput performance of the logger, while providing significant security benefits. This result corresponds to the theory since updating the FssAgg MAC requires a small constant number of operations, no matter the size of the log. The time required to update the Merkle tree is $O(log_2(n))$, as confirmed by the measurements.

The complete log design, that is presented in chapter 6, includes the

Merkle tree structure, FssAgg and commitment signing. The figure shows a comparison of different signature algorithms. Commitment signing is the most time-demanding operation performed by the logger. The same discovery was made in the original Merkle tree log paper [8].

According to the most recent NIST (National Institute of Standards and Technology) recommendations [34], a minimum key size of 2048-bit and 224-bit are considered safe until 2030 for RSA (Rivest–Shamir–Adleman) and EC (Elliptic Curve), respectively. Based on the metrics, ECDSA (Elliptic Curve Digital Signature Algorithm) 233-bit is the best choice since it offers good security and good throughput performance.

The purpose of commitment signing is to make it possible for clients and auditors to verify that a commitment received was in fact created by the logger. Therefore, it is possible for the logger to change signature scheme during its lifetime. As long as clients and auditors are aware of the change and when it takes place, it is not an issue. For that reason, it would be unnecessary to use an excessively large key size, since log throughput performance would be negatively impacted.

When adding 4 million log entries to the log with ECDSA 233-bit signatures the throughput is 1420 entries per second when FssAgg MAC is enabled compared to 1500 entries per second when FssAgg MAC is disabled. Throughput decreases by 5.3% when FssAgg is enabled.

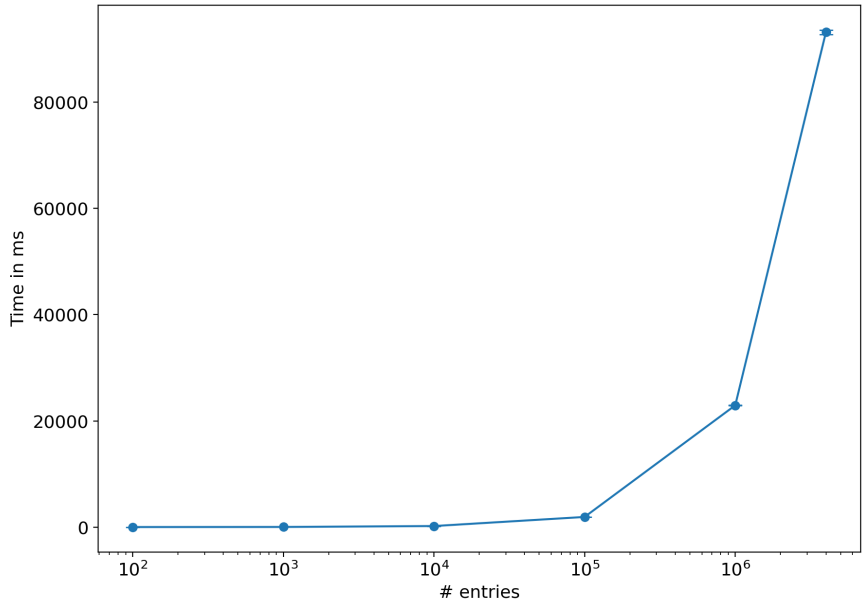

Figure 7.2: The time required to verify an FssAgg MAC depending on the number of entries it protects. Plotted with confidence intervals on a log scale x-axis.

Figure 7.2 shows the time required to verify an FssAgg MAC plotted on a log scale x-axis. As expected, time increases linearly with the number of entries protected by the FssAgg MAC. This can be bad for big logs but, as previously mentioned, the main purpose of the FssAgg MAC is to provide stronger protection against truncation-attacks. Therefore the auditor verifying the FssAgg MAC does not need to verify it for all entries every time, only for the entries not seen since the last audit.

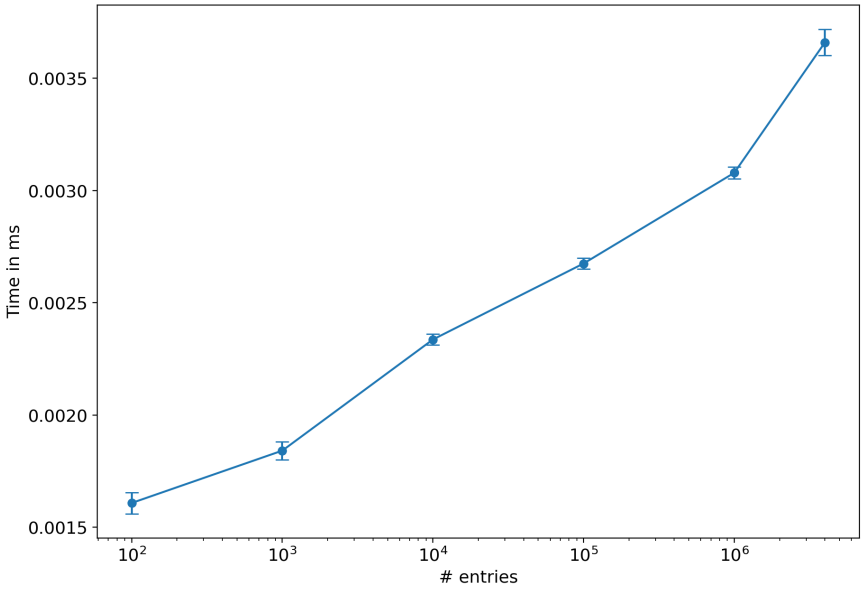

Figure 7.3: The time required produce a Merkle membership proof for one entry for different sized logs. Plotted with confidence intervals on a log scale x-axis.

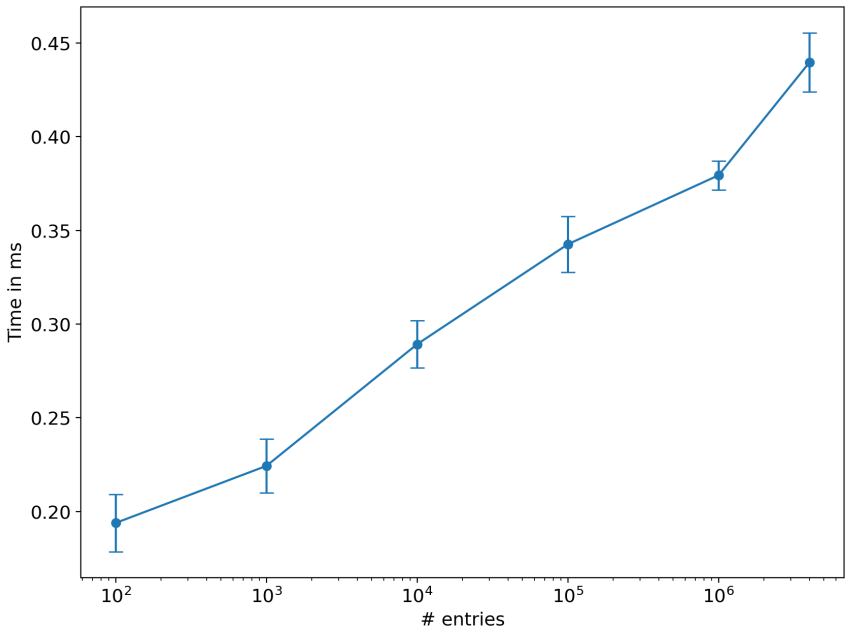

Figure 7.4: The time required to verify a Merkle membership proof for one entry for different sized logs. Plotted with confidence intervals on a log scale x-axis.

Figures 7.3 and 7.4 show the time required to generate and verify Merkle membership proofs. As expected, the time required grows logarithmically with the number of log entries.

Verifying the Merkle tree proofs is a lot faster than verifying FssAgg MACs. But they grant different security properties, and having both gives the user of the log flexibility in how they want to perform audits and verification.

Chapter 8

Discussion, conclusions and future work

This project has resulted in a log design and a proof-of-concept implementation based on the current state-of-the-art secure logging research and an analysis of the current EJBCA audit log. The goal from the start was to provide a basis of knowledge to start from when implementing the next-generation audit log for EJBCA. The solution and the evaluation show that it is possible to create a new logging system that is more secure and functionally superior to the current logging system in EJBCA. The design is based on an analysis of the current EJBCA solution and the research literature.

Merkle tree-based logging is in use today for tamper-evident logging. Most notably for the web PKI:s Certificate Transparency system. The Merkle tree log design shows that a log based on a Merkle tree provides both strong security properties thanks to efficient proofs and efficient searching courtesy of the Merkle aggregation technique.

Adding the FssAgg MAC to the logging design provides additional security to the log without sacrificing log throughput, as was shown from the performance metrics from the proof-of-concept implementation.

The choice of using the Merkle tree design for the log provides many benefits but has the downside of making the most used log operation, adding entries, slower compared to table-based solutions. Adding an entry is an $O(log_2(n))$ operation in the Merkle tree, compared to $O(1)$ in a table. Performance and throughput has not been the main focus of this project. It is possible that the log system design proposed is not efficient enough for EJBCA customers that have PKIs under very heavy load.

In general, it is always a trade-off that a system with increased security

will have worse performance. One option would be to provide a setting in EJBCA to use either the less secure table based-log or the new log design, with superior security properties. Another option for future work would be to explore the possibility of having a sharded log where there would be multiple Merkle tree-based logs at the same time. The logs could share the log entries between them, thus enabling a higher throughput for adding new entries. The downside of such an approach would be that querying and verification would become more complex. It could also make the log less secure.

Encryption of log entry data was not in the scope of this project since confidentiality is not a requirement for the EJBCA audit log. However, it is possible that some EJBCA customer could want encryption capabilities for audit log contents for some special scenario. Furthermore, if the log solution presented in this project would be used for other applications, encryption could be useful. Since the new log has verification capabilities for third-party actors, for example government agencies, there are possible scenarios where the audit log could be susceptible to GDPR regulationns. In that case, confidentiality could be an important aspect. Extending the log solution presented in this project with encryption capabilities is possible, e.g. the log entry could be encrypted before being sent to the logger. The topic of having confidentiality functionality as a part of the log itself is something that could be explored in future work.

PrimeKey will pursue the goal of creating a new logging system with this project as a starting point. The work presented in this project is mostly theoretical and focused on security. It is likely that when creating the new EJBCA audit log implementation, trade-offs will need to be made regarding security, performance and the complexity of the design. Using a Merkle tree for the log has large benefits in terms of efficient and flexible auditing. The vulnerability to truncation-attacks is the major security flaw of the current audit log implementation and the FssAgg MAC is a good way to make truncation-attacks detectable.

The proposed solution for the new audit log design is more complex than what is currently used in EJBCA. If the new design is implemented it could result in a slight increase in power consumption when new auditable events occur, compared to the previous design. PKI is a small part of the cybersecurity space, which in turn is just one part of the bigger information technology sector. Therefore it is extremely unlikely that the implementationn of the log design proposed in this report would have any significant impact on the ecological environment.

From a societal viewpoint, the log design can increase the trust in CAs and

therefore PKIs, which is a positive for society since the purpose of a PKI is to facilitate security, integrity and trust.

I have achieved all the goals set out at the start of this project and in the process I have learned a lot about cryptoraphy, Merkle trees, secure logging and computer security in general.

References

[1] Mihir Bellare and Bennet Yee. *Forward integrity for secure audit logs*. Tech. rep. Citeseer, 1997.

[2] *EJBCA - The Open Source CA*. en-US. URL: https://www.ejbca.org/ (visited on 05/16/2022).

[3] *What is PKI? A Public Key Infrastructure Definitive Guide*. en-US. URL: https://www.keyfactor.com/resources/what-is-pki/ (visited on 05/17/2022).

[4] Mohammad Khodaei and Panos Papadimitratos. "Scalable Resilient Vehicle-Centric Certificate Revocation List Distribution in Vehicular Communication Systems". In: *IEEE Transactions on Mobile Computing (IEEE TMC)* 20.7 (July 2021), pp. 2473–2489.

[5] Mohammad Khodaei, Hongyu Jin, and Panos Papadimitratos. "SECMACE: Scalable and Robust Identity and Credential Management Infrastructure in Vehicular Communication Systems". In: *IEEE Transactions on Intelligent Transportation Systems (IEEE T-ITS)* 19.5 (May 2018), pp. 1430–1444.

[6] *GoDaddy - Verify domain ownership (DNS or HTML) for my SSL certificate*. en. URL: https://ca.godaddy.com/help/verify-domain-ownership-dns-or-html-for-my-ssl-certificate-7452 (visited on 05/17/2022).

[7] *Who your browser trusts, and how to control it.* URL: https://expeditedsecurity.com/blog/control-the-ssl-cas-your-browser-trusts/ (visited on 05/22/2022).

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

[9] Radu Sion. "Strong worm". In: *The 28th International Conference on Distributed Computing Systems*. IEEE. 2008, pp. 69–76.

[10] Di Ma and Gene Tsudik. “A new approach to secure logging”. In: *ACM Transactions on Storage (TOS)* 5.1 (2009), pp. 1–21.

[11] Benedikt Putz, Florian Menges, and Günther Pernul. “A secure and auditable logging infrastructure based on a permissioned blockchain”. In: *Computers & Security* 87 (2019), p. 101602.

[12] Alfonso de la Rocha and Panos Papadimitratos. “Blockchain-based Public Key Infrastructure for Inter-Domain Secure Routing”. In: *IFIP WG 11.4 Workshop on Open Problems in Network Security (IFIP iNetSec)*. Rome, Italy, May 2017.

[13] Bastian Fredriksson. *A distributed public key infrastructure for the web backed by a blockchain*. 2017.

[14] Bruce Schneier and John Kelsey. “Secure audit logs to support computer forensics”. In: *ACM Transactions on Information and System Security (TISSEC)* 2.2 (1999), pp. 159–176.

[15] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. “Keying hash functions for message authentication”. In: *Annual international cryptology conference*. Springer. 1996, pp. 1–15.

[16] Jason E Holt and Kent E Seamons. “Logcrypt: forward security and public verification for secure audit logs”. In: *Cryptology ePrint Archive* (2005).

[17] Adi Shamir. “Identity-based cryptosystems and signature schemes”. In: *Workshop on the theory and application of cryptographic techniques*. Springer. 1984, pp. 47–53.

[18] Di Ma and Gene Tsudik. “Forward-secure sequential aggregate authentication”. In: *IEEE Symposium on Security and Privacy (SP’07)*. IEEE. 2007, pp. 86–91.

[19] Ralph C Merkle. “A digital signature based on a conventional encryption function”. In: *Conference on the theory and application of cryptographic techniques*. Springer. 1987, pp. 369–378.

[20] Laurent Chuat et al. “Efficient gossip protocols for verifying the consistency of certificate logs”. In: *IEEE Conference on Communications and Network Security (CNS)*. IEEE. 2015, pp. 415–423.

[21] Ben Laurie et al. *Certificate Transparency Version 2.0*. Request for Comments RFC 9162. Num Pages: 53. Internet Engineering Task Force, Dec. 2021. DOI: 10.17487/RFC9162. URL: https://datatracker.ietf.org/doc/rfc9162 (visited on 05/19/2022).

[22] Josephine Wolff. “How a 2011 Hack You’ve Never Heard of Changed the Internet’s Infrastructure”. In: *Slate* (Dec. 2016). ISSN: 1091-2339. URL: https://slate.com/technology/2016/12/how-the-2011-hack-of-diginotar-changed-the-internets-infrastructure.html (visited on 05/19/2022).

[23] *How CT Works : Certificate Transparency*. URL: https://certificate.transparency.dev/howctworks/ (visited on 05/20/2022).

[24] *How to Enable Certificate Transparency (CT) | DigiCert.com*. URL: https://www.digicert.com/faq/certificate-transparency/enabling-ct.htm (visited on 11/29/2022).

[25] *EJBCA Documentation*. URL: https://doc.primekey.com/ejbca (visited on 05/18/2022).

[26] *EJBCA Audit Logging*. URL: https://doc.primekey.com/ejbca6152/ejbca-operations/ejbca-concept-guide/logging/ejbca-audit-logging (visited on 05/19/2022).

[27] *Integrity Protected Security Audit Log*. URL: https://doc.primekey.com/ejbca6152/ejbca-operations/ejbca-concept-guide/logging/view-log-options/integrity-protected-security-audit-log (visited on 05/19/2022).

[28] *Creating the Database - EJBCA - Documentation Space*. URL: https://download.primekey.se/docs/EJBCA-Enterprise/latest/Creating_the_Database.html (visited on 06/08/2022).

[29] *Issuing eID Certificates and Signing ePassports*. URL: https://doc.primekey.com/ejbca/solution-areas/issuing-eid-certificates-and-signing-epassports (visited on 05/19/2022).

[30] Node.js. *Node.js*. en. URL: https://nodejs.org/en/ (visited on 05/20/2022).

[31] Miguel Mota. *MerkleTree.js*. original-date: 2017-07-22T07:25:26Z. May 2022. URL: https://github.com/miguelmota/merkletreejs (visited on 05/20/2022).

[32] *Crypto | Node.js v18.2.0 Documentation*. URL: https://nodejs.org/api/crypto.html (visited on 05/20/2022).

[33] *crypto-js*. en. URL: https://www.npmjs.com/package/crypto-js (visited on 05/20/2022).

[34] Elaine Barker. *Recommendation for Key Management Part 1: General*. en. Tech. rep. NIST SP 800-57pt1r4. National Institute of Standards and Technology, Jan. 2016, NIST SP 800–57pt1r4. DOI: 10.6028/NIST.SP.800-57pt1r4. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r4.pdf (visited on 07/04/2022).

TRITA-EECS-EX-2022:926