MASARYK
UNIVERSITY

FACULTY OF INFORMATICS

Fortifying Digital Evidence Integrity: A Forensic-Ready Software Approach with Distributed Protection

Master's Thesis

DÁVID MASLO

Brno, Fall 2023

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Fortifying Digital Evidence Integrity: A Forensic-Ready Software Approach with Distributed Protection

Master's Thesis

DÁVID MASLO

Advisor: RNDr. Lukáš Daubner

Department of Computer Systems and Communications

Brno, Fall 2023


FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Dávid Maslo

Advisor: RNDr. Lukáš Daubner

Acknowledgements

First, I would like to thank my advisor, Lukáš Daubner, who was always providing me with new ideas and valuable feedback. I would also like to thank my teammates from the PV204 semestral project, who allowed me to reuse some parts of the implementation for this thesis. Lastly, I would like to thank my family, without whose support I would not have been able to get to this point.

Abstract

This thesis delves into the preservation of potential digital evidence integrity for forensic investigations, examining the efficacy of blockchain and threshold signatures in addressing this challenge. Through the utilization of threshold signatures, the research suggests a collaborative multi-organizational scheme, emphasizing its security and performance attributes. A key focus is on bridging a critical gap by experimentally assessing the exposure window size, the temporal frame during which potential evidence could be vulnerable to manipulation. The evaluation further contrasts the proposed scheme with blockchain approaches, highlighting the reduction in exposure window sizes that was achieved.

Keywords

Threshold Signatures, Blockchain, Forensic-Ready Software, Multi-Party Computation, Cybersecurity

Contents

Introduction **1**

1 Logs as a Forensic Evidence **3**
1.1 Real-world Storage Systems Model 3
1.1.1 Security Properties of the Model 5
1.2 Security Analysis 6
1.3 Context and Asset Identification 6
1.4 Security Objective Determination 8
1.5 Threat Agents 8
1.6 Risk and Impact Analysis 9
1.7 Risk Treatment Decision 12
1.8 Security Requirements 12

2 State of the Art **14**
2.1 Related Work 14
2.2 Blockchain-based Database: EngraveChain 15
2.2.1 Architecture 16
2.2.2 Security Analysis 17
2.2.3 Design Shortcomings 19
2.2.4 Exposure Window 19
2.3 Threshold Signatures 19
2.3.1 Architecture 19
2.3.2 Security Analysis 20
2.3.3 Design Shortcomings 22
2.3.4 Exposure Window 23
2.4 Trusted Notary Service 23

3 Solution Design **25**
3.1 Choice of Technology 25
3.2 Threshold Signatures 26
3.3 Threshold Signatures Protocol 27
3.4 Design 27
3.4.1 Signing 27
3.4.2 Log Integrity Verification 28
3.4.3 Threshold 30
3.4.4 Security Requirements 30

3.5 Security Analysis 31
3.6 Key Generation 33
3.6.1 Key Rotation 33
3.7 Design Shortcomings 34
3.7.1 Scalability 34
3.7.2 Colluding Organizations 35
3.8 Summary 35

4 Implementation **36**
4.1 Log Signing Server 36
4.1.1 Technology 36
4.1.2 Server Design 36
4.1.3 Endpoints 38
4.1.4 Asynchronous Design 44
4.2 Testing 46
4.2.1 Technology 46
4.2.2 Integration Tests 47
4.2.3 Performance Tests 47
4.3 Deployment 48
4.4 Summary 48

5 Evaluation **49**
5.1 Testing Environment 49
5.2 Evaluation Targets 50
5.3 Threshold Setups 50
5.4 Methodology 51
5.5 Results 51
5.5.1 Key Generation 51
5.5.2 Signing 52
5.5.3 Verification 55
5.6 Summary 56

6 Conclusion **58**

A Archive Content **59**
A.1 LS Server 59
A.1.1 Prerequisities 59
A.1.2 Simple Setup 59

A.1.3 Operations 60
A.2 Evaluation 61
A.3 Deployment 61

Bibliography **62**

List of Tables

1.1 Remaining impacts of the centralized model 11
1.2 Targeted security properties 13
2.1 Security properties of EngraveChain 18
2.2 Security properties of threshold signatures 22
3.1 Security properties of proposed design 32

List of Figures

1.1 Centralized log storing model (based on [5]) 4

2.1 EngraveChain high-level design [2] 16
2.2 EngraveChain file recreation [2] 17
2.3 The model with blockchain 18
2.4 A resilient architecture for forensic storage of events in critical infrastructures [3] 20
2.5 The model with threshold signatures 21
2.6 The model with a timestamping service 24

3.1 Distributed threshold signature solution 29
3.2 Verification of signature by forensic analyst 30
3.3 A key generation without trusted dealer 33

4.1 Server setup 38
4.2 Implementation of /key_gen endpoint 40
4.3 Implementation of /sign endpoint 43
4.4 Implementation of /verify endpoint 45

5.1 A key generation duration (lower is better) 52
5.2 Exposure window (lower is better) 53
5.3 Configuration 1: log/second ratio (higher is better) 54
5.4 Configuration 2: log/second ratio (higher is better) 54
5.5 Verification log/second ratio (higher is better) 56

Introduction

In today's rapidly evolving digital landscape, digital forensics is undeniably critical to modern investigations. This specialized field enables investigators to analyze electronic evidence, uncover digital trails, and piece together crucial information, significantly contributing to resolving complex cases.

For digital forensic evidence to be admissible, especially in a court of law, it must maintain its integrity. Traditionally, this assurance is established as a forensic specialist secures the potential evidence in the form of logs and ensures its preservation throughout the investigative process. However, before the logs are secured, there are no strong guarantees that the logs were not tampered with, leaving a significant gap for manipulation. To bridge this gap, systems must be purposefully designed to be forensic-ready [1].

To achieve forensic readiness and mitigate the issue of insufficient log integrity, the researchers conducted studies leveraging blockchain technology and threshold signatures [2][3]. Blockchain approaches use a decentralized ledger for tamper-proof storage, while threshold signatures represent a cryptographic technique that can distribute the responsibility of log signing among multiple parties. While both approaches show promising outcomes, these studies generally lack a thorough examination of the *exposure window size*, a critical time frame during which potential digital evidence may be vulnerable to tampering.

This thesis aims to fill this gap by proposing an approach based on the threshold signatures. It builds its proposal based on existing solutions and, where applicable, compares the security properties and performance to the blockchain-based solutions. Since current studies are primarily theoretical and provide limited implementations, the thesis seeks to create a proof-of-concept that can be used to assess the exposure window size experimentally.

Chapter 1 develops a threat model, specifies threat agents, and analyzes risks to the centralized log storage systems with baseline security measures. Based on this analysis, the security goals of the thesis are specified.

Chapter 2 discusses the current state-of-the-art solutions for storing proper digital forensic evidence. It looks at three possible approaches and analyzes them for their security properties and shortcomings.

Chapter 3 designs a solution built on the idea of two-organization collaboration. It leverages threshold signatures that decrease the impact of the threat agents mentioned in the first chapter.

Chapter 4 presents the proof-of-concept implementation that is later used for the evaluation. It explains the building blocks of the solution, how it was tested, and how it can be deployed.

Chapter 5 evaluates the exposure window size that the proposed approach achieves and whether it is feasible for real-world usage, comparing it with other approaches where applicable.

1 Logs as a Forensic Evidence

Generally, a system generates and retains logs either on local storage or by transmitting them to a centralized server [4]. The term "log" refers to the output generated by a computer system, device, or software in response to various stimuli and serves diverse purposes, including providing informational content, debugging messages for developers, warnings, or documenting errors within the system [4]. They are typically categorized into different levels to help distinguish their severity and are valuable information for forensic analysts securing potential digital evidence. However, using logs as legal evidence poses challenges because, before they are secured for investigation, they remain susceptible to modification by different threat actors.

1.1 Real-world Storage Systems Model

Real-world storage systems are rather complex. They include many log sources such as firewalls, applications, web servers, etc., meaning they typically need dedicated infrastructure to handle them. One approach is having a *collection server* that processes them and then stores them in storage [5]. From there, a forensic analyst can extract the logs (including origin verification) to serve as potential digital evidence, see Figure 1.1.

The model contains four entities:

1. **Log Generators**: aggregation of different entities that produce logs.
2. **Collection Server**: collects and processes logs from *log generators*.
3. **Storage Server**: stores logs.
4. **Forensic Analyst Disk**: a medium to which a forensic analyst extracts potential evidence.

When a *log generator* produces a log, it sends it to the *collection server*, where the logs are processed. When finished, the *collection server*

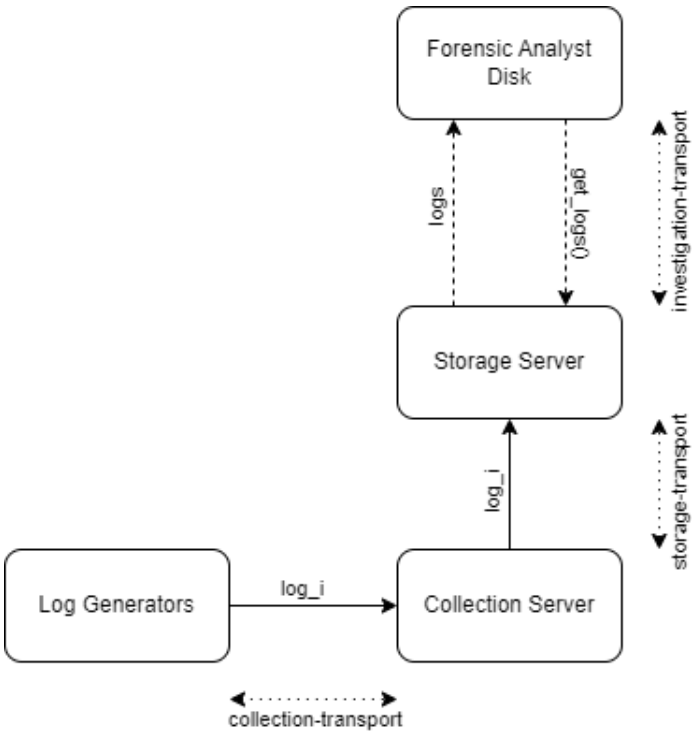


Figure 1.1: Centralized log storing model (based on [5])

forwards it to the *storage server*, where it is stored until the forensic analyst aggregates the logs in the *forensic analyst disk* to obtain potential forensic evidence from them.

1.1.1 Security Properties of the Model

Specific baseline security measures are assumed to mimic existing log-gathering solutions as much as possible:

- The entities use TLS when communicating with each other, meaning the client is authenticated, and the logs are encrypted. The *log generators* and the *forensic analyst disks* are clients only. The *collection server* serves as the server for the *log generators* and is the client in the *collection server* to the *storage server* interactions. The *storage server* is the server both for the *collection server* and the *forensic analyst disk*.

- Strong access control prevents external attackers from compromising the *log generators* and *collection/storage servers* where they could tamper with the logs.

- The medium through which logs are transferred makes it infeasible to prevent logs from reaching their target. This is true when the logs travel from *log generators* to the *collection server*, from the *collection server* to the *storage server*, and from the *storage server* to the *forensic analyst disk*.

- The *log generators* and *collection server* are potentially susceptible to the denial of service attack, making the collection of logs unavailable.

However, even with these security precautions, the system administrator can write or make logs unavailable during almost any stage. This is because standard security measures do not guard against insider threats (there is always at least one administrator who has control over at least part of the system). Moreover, the fact that the system administrator can read the logs is not considered because it is essential that an organization can see its logs.

1.2 Security Analysis

For the security analysis, the thesis uses the *Domain Model for Information Systems Security Risk Management*, or *ISSRM*, described in [6] to assess risks systematically. The process consists of six steps regarding risk analysis and risk mitigation:

1. Context and asset identification.
2. Security objective determination.
3. Risk analysis and assessment.
4. Risk treatment.
5. Security requirements definition.
6. Security selection and implementation[1].

The results, namely of risk analysis, will serve as a basis for analyzing the security properties of existing solutions that try to minimize the window of exposure in chapter 2.

1.3 Context and Asset Identification

Logs are a **business asset** that "describes the information, processes, capabilities, and skills essential to the business and its core mission" [6], and a security need on the asset is described through **security criterion** in the form of **confidentiality**, **integrity**, and **availability** of logs.

The only business asset that is considered is the logs. There are plenty of reasons why organizations might consider logging to be useful. For example, [7] splits logs into four categories:

1. **Business relevant logging:** covers features used and business metrics being tracked.
2. **Operations-based logging:** errors, critical system conditions, configuration changes, etc.

1. Since this chapter only presents a centralized model of how the logs are typically stored, it omits the last step.

3. **Security (forensics) related logging:** concerned with authentication, authorization, and forensics support. "In addition to these three cases, security tools (e.g., intrusion detection or prevention systems or antivirus tools) will log all kinds of other security-related issues, such as attempted attacks or the detection of a virus on a system." [7]

4. **Regulatory and standards mandates:** compliance and regulatory demands are another group of use cases that demand to log.

When introducing changes to improve forensic-ready logging, it is necessary to do so in a way that the current processes are not disrupted. This means that organizations can still analyze their logs, track metrics, detect intrusion, etc., without unacceptable degradation of service.

The business asset (logs) is supported by **system assets** (defined as assets that support business assets [6]). These are four entities from section 1.1 plus transport mediums:

- The **log generators** where the logs are generated.
- The **collection server**, where logs are collected and processed.
- The **storage server**, where the logs are stored and from where they can be retrieved to be used in an investigation.
- The **forensic analyst disk** where the forensic analyst stores logs during an investigation.
- The **collection-transport** transports logs from *log generators* to the *collection server*.
- The **storage-transport** transports logs from the *collection server* to the *storage server*.
- The **investigation-transport** transports logs from the *storage server* to the *forensic analyst disk*.

1.4 Security Objective Determination

First, the **exposure window** is defined as a critical time frame during which potential digital evidence may be vulnerable to tampering (be, it on the *log generator*, *collection server*, *storage server*, *forensic analyst disk*, or during transitions between these entities).

The exposure window begins with *log generators* generating the logs and ends once the forensic analyst presents them as evidence after an investigation. It is the time during the threat agents can compromise confidentiality, integrity, and availability of the logs. In standard log storage systems, the window is quite large. The time from the log generation to their usage as potential evidence can easily take years.

Since the thesis is trying to minimize the exposure window, the security goal is to have log confidentiality, integrity, and availability as early as possible. This should be valid from the standpoint of an **external attacker**, **infiltrated attacker**, **system administrator**, and **forensic analyst** threat agents. Even though these are ideal goals, confidentiality will not be reached from the *infiltrated attacker's* point of view, and there will be an availability impact from the *system administrator* on the *collection server*.

1.5 Threat Agents

The thesis considers four threat agents:

1. An **external attacker**: any entity with malicious intentions outside the organization that has not yet infiltrated any business assets from section 1.1.

2. An **infiltrated attacker**: any entity that compromised some (potentially all) business assets.

3. A **system administrator**: any entity with administrator rights to access the logs or a superior who can influence such people.

4. A **forensic analyst**: any entity tasked with collecting potential evidence. He poses a threat both to the logs in the *log storage* and their copy on the *forensic analyst disk*. In the *log storage*, he can directly manipulate the logs. In the *forensic analyst disk*, he

can manipulate the copy to plant evidence; it is then unclear who manipulated the data, whether the forensic analyst or an organization.

1.6 Risk and Impact Analysis

The baseline security measures discussed in subsection 1.1.1 mitigate most of the security risks caused by an *external attacker*. However, there are plenty more to consider. This part splits the model from section 1.1 to parts. It states the impact that each threat agent can have on the logs (if any) in terms of a CIA triad and describes the risks that lead to this impact.

Log generators and Collection Server

1. An *external attacker* can impact:
   - *Availability:* influence the log generation process by leveraging denial of service attacks and thus prevent the creation of potential digital evidence (super producer threat [8]).
2. An *infiltrated attacker* can impact:
   - *Confidentiality:* read the content of the logs.
   - *Integrity:* tamper with the logs and thus tamper with the potential digital evidence (in other words, she can plant invalid evidence).
   - *Availability:* compromise the log generation process on *log generators* or prevent forwarding the logs to the *storage server* on the *collection server*. She thus prevents log creation and prevents the creation of potential digital evidence.
3. A *system administrator* can impact:
   - *Integrity:* tamper with the logs and thus tamper with the potential digital evidence (in other words, he can plant invalid evidence).

- *Availability:* compromise the log generation process or storage forwarding to prevent log creation and thus prevent the creation of potential digital evidence.

Collection-transport, storage-transport, investigation-transport

1. An *infiltrated attacker* can impact:
   - *Confidentiality:* read the content of the logs.
   - *Integrity:* tamper with the logs and thus tamper with the potential digital evidence (in other words, she can plant invalid evidence).
2. A *system administrator* can impact:
   - *Integrity:* tamper with the logs and thus tamper with the potential digital evidence (in other words, he can plant invalid evidence).

Storage Server

1. An *infiltrated attacker* can impact:
   - *Confidentiality:* read the content of the logs.
   - *Integrity:* tamper with the logs and thus tamper with the potential digital evidence (in other words, she can plant invalid evidence).
   - *Availability:* remove logs from the *storage server* (and thus remove potential digital evidence).
2. A *system administrator* can impact:
   - *Integrity:* tamper with the logs and thus tamper with the potential digital evidence (in other words, he can plant invalid evidence).
   - *Availability:* remove logs from the *storage server* (and thus remove potential digital evidence).
3. A *forensic analyst* can impact:

Table 1.1: Remaining impacts of the centralized model

	Ext. A.	**Inf. A.**	**Sys. A.**	**For. A.**
Log generators	-/-/A	C/I/A	∅/I/A	∅
Collection-transport	-/-/∅	C/I/∅	∅/I/∅	∅
Collection Server	-/-/A	C/I/A	∅/I/A	∅
Storage-transport	-/-/∅	C/I/∅	∅/I/∅	∅
Storage Server	-/-/-	C/I/A	∅/I/A	∅/I/A
Investigation-transport	-/-/∅	C/I/∅	∅/I/∅	∅
Forensic Analyst Disk	∅	∅	∅	∅/I/A

. *C* - confidentiality violation (a threat agent can read logs), *I* - integrity violation (a threat agent can modify logs), *A* - availability violation (a threat agent can make logs unavailable), ∅ - does not apply.
. *Ext. A.* - external attacker, *Inf. A.* - infiltrated attacker, *Sys. A.* - system administrator, *For. A.* - forensic analyst

- *Integrity:* tamper with the logs when securing the potential evidence.
- *Availability:* remove logs when securing the potential evidence.

Forensic Analyst Disk

1. A *forensic analyst* can impact the following properties of the log copy:
   - *Integrity:* modify or plant logs and falsify potential evidence.
   - *Availability:* delete logs and thus remove part of the evidence.

The risk and impact analysis summary can be seen in the Table 1.1. It is essential to point out that this section treats the *forensic analyst disk* as a special case compared to the other parts. It assumes that the *forensic analyst* is the only entity with access to it, so other threat agents do not apply in this case. Moreover, it also assumes that he has access only to the *storage server* (when securing evidence) and no other.

1.7 Risk Treatment Decision

Four risk treatment concepts are presented in the [6]: *risk avoidance*, *risk reduction*, *risk transfer*, and *risk retention*. *Risk avoidance* represents a decision to withdraw from risk by modifying or removing system functionality. *Risk reduction* means taking actions to lessen the probability of the risk. *Risk transfer* defines how risk parties can share the burden of loss from risk. *Risk retention* implies acceptance of the risk.

The time from the log generation until it is securely present at the *collection server* is labeled as the **remaining exposure window** (risk retention). The thesis proposes a distributed method for the remaining assets with the aim of incorporating a risk reduction strategy to enhance security guarantees regarding the log's integrity and availability.

1.8 Security Requirements

The model from the section 1.1 already reduces a lot of impacts from the point of an *external attacker* by incorporating baseline security measures. However, the distribution of the *collection server* with the *storage server* between multiple organizations can further decrease the impacts from other threat agents. The improvements that it achieves are colored green in the Table 1.2. These represent the security properties of the proposed solution presented in the chapter 3. To achieve those goals, the following security requirements have to be placed on the solution:

1. The *collection server* has to be distributed between at least two entities unlikely to cooperate, and each part must have backups so the *external attacker* and the *infiltrated attacker* cannot bring the system offline (although the *system administrator* still can).

2. For the *collection server* to accept a log, at least two entities for which it is infeasible to cooperate must reach a consensus about log authenticity and prove that they did.

3. All business asserts after the *collection server* must treat a log without proof of consensus as invalid.

Table 1.2: Targeted security properties

	Ext. A.	**Inf. A.**	**Sys. A.**	**For. A.**
Log Generators	-/-/A	C/I/A	∅/I/A	∅
Collection-transport	-/-/∅	C/I/∅	∅/I/∅	∅
Collection Server	-/-/-	C/-/-	∅/-/A	∅
Storage-transport	-/-/∅	C/-/∅	∅/-/∅	∅
Storage Server	-/-/-	C/-/-	∅/-/-	∅/-/-
Investigation-transport	-/-/∅	C/-/∅	∅/-/∅	∅
Forensic Analyst Disk	∅	∅	∅	∅/-/-

. *C* - confidentiality violation (a threat agent can read logs), *I* - integrity violation (a threat agent can modify logs), *A* - availability violation (a threat agent can make logs unavailable), ∅ - does not apply.
. *Ext. A.* - external attacker, *Inf. A.* - infiltrated attacker, *Sys. A.* - system administrator, *For. A.* - forensic analyst

4. The *storage server* must be distributed among at least two entities unlikely to cooperate and must store logs together with the proof of consensus.

5. The third party must be able to verify that the data on the *forensic analyst disk* are complete and equal to the data present on the *storage server*. What is more, the validity of the proof must be verifiable.

6. All communication must be encrypted.

Requirement 1 eliminates the impact of the availability on the *collection server* from the point of the *external* and *infiltrated attackers*. Requirements 2 and 3 eliminate the integrity impact of the logs in the whole system from the point of reaching consensus on the *collection server* until they are transported to the *forensic analyst disk* from all threat agents. Requirement 4 eliminates the impact of the availability of the logs in the *storage server*. Requirement 5 eliminates the integrity and availability impacts on the *forensic analyst disk*. Requirement 6 eliminates the impact of the *external attacker* on confidentiality.

2 State of the Art

The problem with the standard database storage of logs is that no matter what security measures are applied to guard against the *external attacker*, intentional/unintentional changes by the systems administrators are hard to guard against. To tackle this issue in recent years, researchers have tried to leverage **blockchain technology** in similar scenarios. Blockchain technology allows for much stronger integrity and availability [2]. Besides that, there was a proposal to solve integrity problems using **multi-party computation**, using **threshold signatures** in particular [3]. However, both approaches yielded mixed results when applied to this use case.

2.1 Related Work

Since the boom of *blockchain technology*, researchers have examined scenarios that could leverage its integrity and availability properties with regard to forensic evidence. One prominent example is ProvChain, which designs blockchain-based data provenance architecture [9]. Its goal is to store provenance data for object operations in the cloud (cloud services meaning storage services such as Google Drive[1] or Dropbox[2]) and allow an independent provenance auditor to verify provenance data. Guo et al. leveraged blockchain to design an antitampering scheme to record evidence transfer between different departments of China's judicial system [10]. LeChain proposed a scheme to control the entire chain of evidence, from an initial collection of evidence from victims and witnesses through evidence gathering by police investigators up until juror votes and its verification by a judge [11]. The most promising study in this regard seems to be [2], which proposes the use of blockchain in a multi-corporation setting to achieve log integrity and availability (see section 2.2). Unfortunately, none of these considered the exposure window.

Multi-party computation got much less attention both in general and in digital forensics. Despite wide potential usage, such as privacy-preserving auctions, private machine learning, or trusted timestamp-

1. https://www.google.com/drive/
2. https://www.dropbox.com/

ing, nowadays, it is mainly used for threshold signatures [12], especially in multi-signature crypto wallets [13]. Threshold signatures are getting more mature, which is confirmed by NIST's efforts to standardize the schemes [14]. Regarding digital forensics, one prominent example is [3], which leverages current SIEM solutions with the Shoups [15] threshold signature algorithm (based on RSA) to provide forensic resilience (see section 2.3). Unfortunately, the Shoups algorithm suffers from issues such as needing a trusted dealer, which was already fixed in the [16]. Despite RSA, ECDSA signatures have their schemes, such as [17]. Another alternative is the so-called BLS threshold signatures, which provide non-interactive signatures, leading to a more efficient signing [18].

Another potential approach to tamper-proof logs would be to leverage *trusted timestamping services*. They can be used to prove that specific data, such as a document, existed at some point in the past. There are different approaches; some use distributed timestamping, usually through blockchains such as [19], https://guardtime.com, or https://opentimestamps.org/. Others use trusted third parties, such as [20].

The rest of this chapter takes one representative out of these three groups and looks at their security properties in more detail.

2.2 Blockchain-based Database: EngraveChain

The main motive behind putting logs on a blockchain is that it provides the most desirable property, **integrity** [21]. On top of that, its design also provides **availability** since it is distributed in nature [21]. As such, unless fifty-one percent of nodes collude, it provides reasonable guarantees that no one tampered with the evidence, and it cannot be destroyed.

To leverage these properties for log storage, [2] proposes a solution for multiple organizations collaborating to achieve data integrity and availability using Hyperledger Fabric permissioned blockchain. They present an example of multiple banks working together to achieve those goals. Since banks may not necessarily trust each other, each organization encrypts its logs to ensure a certain level of security.

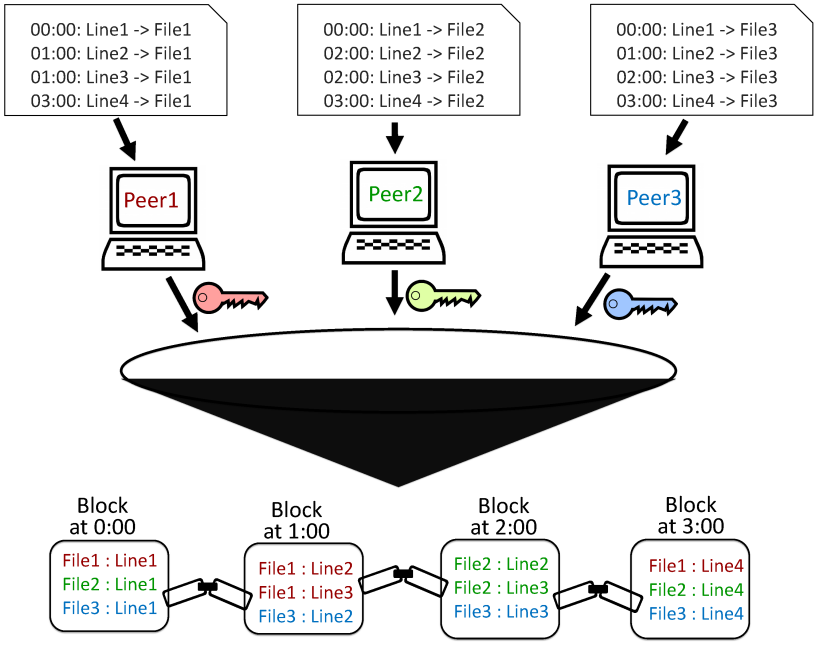


Figure 2.1: EngraveChain high-level design [2]

2.2.1 Architecture

Each organization has an administrator and a certification authority, which generates credentials for its associated peer nodes, allowing them to access the peer-to-peer network. Each peer then stores a copy of a ledger and works with peers from other organizations using the PBFT (Practical Byzantine Fault Tolerance) consensus protocol to store logs [2].

The high-level idea is simple and is visualized in the Figure 2.1. When a log-generating device creates a log, it writes it into the file on a new line. The corresponding organization peer reads this line with the file name from which the line originates and encrypts the line with a private key known only to that organization. Then, by working together with peers of other organizations, it adds *FileN:LineN* into the blockchain.

When the need to recreate the *fileN* later arises (e.g., due to an investigation), the peer search blockchain, for all mentions of the fileN,

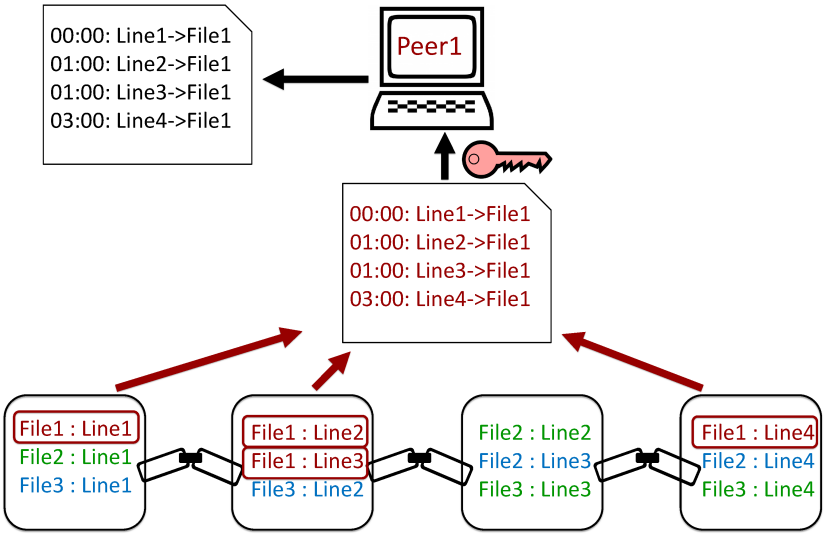


Figure 2.2: EngraveChain file recreation [2]

decrypts each line with the organization's private key and orders them by a timestamp. See Figure 2.2.

2.2.2 Security Analysis

For a security analysis, the model from the section 1.1 was taken and replaced the *collection server* and *storage server* with a blockchain, see Figure 2.3. The results are in the Table 2.1 and are compared against the Table 1.2 (impacts in red means that the design proposed by this thesis does better in this regard). The same security properties are assumed for this model as for the original model in the subsection 1.1.1.

The impacts are almost the same as in the solution that this thesis proposes, except there remains an availability impact on *blockchain* from an *infiltrated attacker*. This is because the *infiltrated attacker* can destroy potential evidence by deleting the decryption key. The fix is not to encrypt the logs, but as the number of participating organizations increases, this becomes infeasible because they typically do not trust each other with logs.

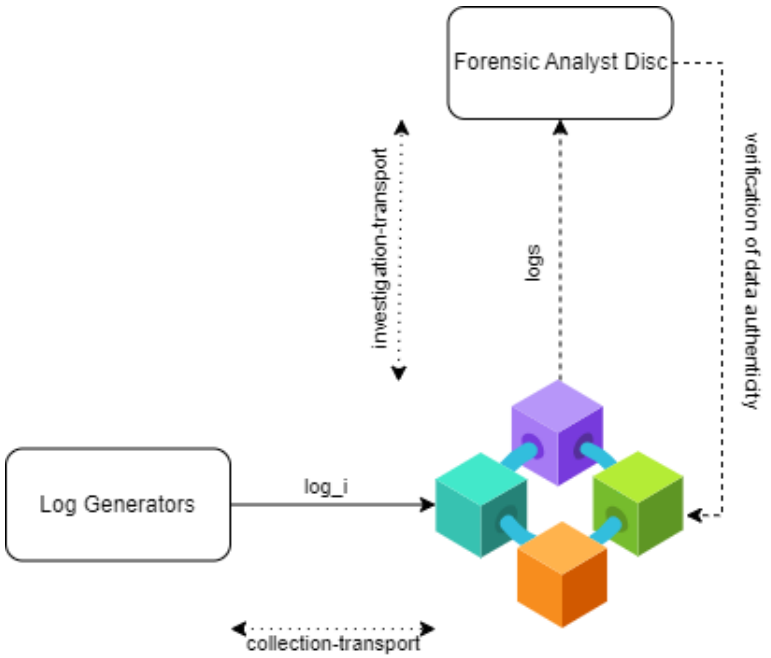


Figure 2.3: The model with blockchain

Table 2.1: Security properties of EngraveChain

	Ext. A.	**Inf. A.**	**Sys. A.**	**For. A.**
Log Generators	-/-/A	C/I/A	∅/I/A	∅
Collection-transport	-/-/∅	C/I/∅	∅/I/∅	∅
Blockchain	-/-/-	C/-/A	∅/-/A	∅/-/-
Investigation-transport	-/-/∅	C/-/∅	∅/-/∅	∅
Forensic Analyst Disk	∅	∅	∅	∅/-/-

. *C* - confidentiality violation (a threat agent can read logs), *I* - integrity violation (a threat agent can modify logs), *A* - availability violation (a threat agent can make logs unavailable), ∅ - does not apply.

. *Ext. A.* - external attacker, *Inf. A.* - infiltrated attacker, *Sys. A.* - system administrator, *For. A.* - forensic analyst

2.2.3 Design Shortcomings

Apart from simply storing logs for possible future use in the investigation, there are usually mechanisms in the company that detect intrusions in the first place. It is, however, much more complicated with the blockchain structure, and as such, "it is unlikely to get blockchains to replace a mature real-time log storage and analytics engine like ElasticSearch in the near future" [22]. It might thus be more feasible to store only hashes on the blockchain and store the logs off-chain, which is a more typical approach. It is important to point out that even in this case, the logs themselves need to be distributed so each organization has logs of each other. Otherwise, an organization can delete its logs (it will be detectable, though, because hashes are in the blockchain).

2.2.4 Exposure Window

The EngraveChain paper did not reproducibly demonstrate its exposure window. They proposed a theoretical concept but did not support it with an implementation that would demonstrate its performance. They only analyzed the overall performance of HyperLedger. Other papers, such as [22], concluded that HyperLedger Fabric is still far behind conventional solutions, although they did not do any reproducible performance analysis themselves either.

2.3 Threshold Signatures

The [3] approached the problem of potential digital evidence storage using *threshold signatures* instead. The distributed computation is preserved by distributing signature computation on different nodes. The authors proposed a solution based on Shoup's RSA threshold cryptography [15] for resilient forensic storage.

2.3.1 Architecture

The architecture of the solution is visualized in the Figure 2.4. When some system creates a security event m[3], it propagates it to all n nodes

3. A log in the context of this thesis.

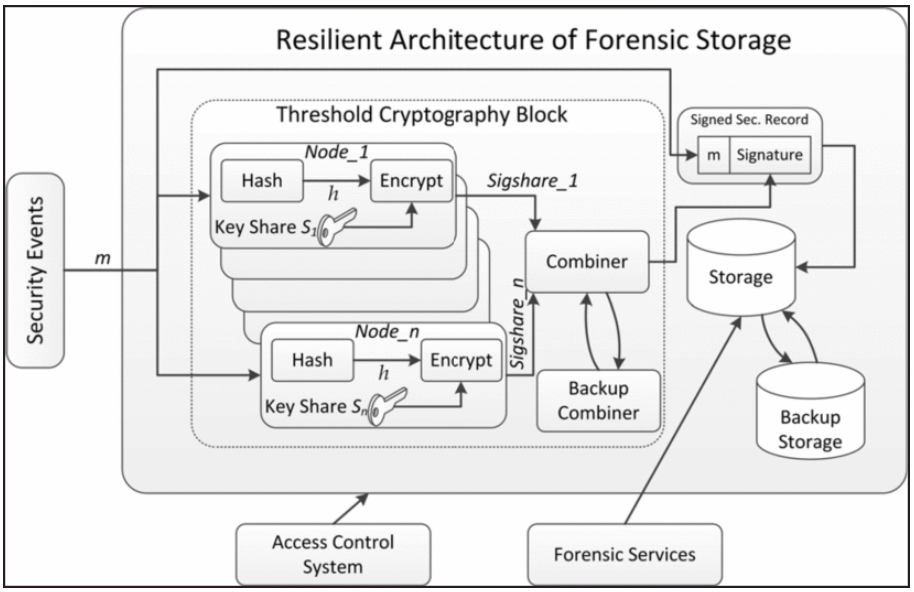


Figure 2.4: A resilient architecture for forensic storage of events in critical infrastructures [3]

for signature. Each node tries to create a partial signature and send it to the *combiner*. Shoup's RSA threshold protocol requires at least *k+1* ($k \leq n$)[4] correct partial signatures to create a complete signature. When the *combiner* collects *k+1* valid signatures, it makes a complete signature. This signature and security event m (log) are then stored in the *storage*. The *backup combiner* and *backup storage* are added for resilience.

2.3.2 Security Analysis

This section incorporated this solution into the model from the section 1.1 to do a security analysis; see Figure 2.5. First, the *log generator* generates the log_i and sends it to the nodes and the *combiner*. The nodes try to compute a partial signature and send it to the *combiner*.

4. When talking about (k, n) scheme, some authors refer to threshold k as the number of parties required for a signature, others as a threshold where an attacker compromising k or fewer nodes learns nothing about the key. Shoup's paper uses the former, but the protocol used in the implementation uses the latter. This part is adjusted to use the latter convention.

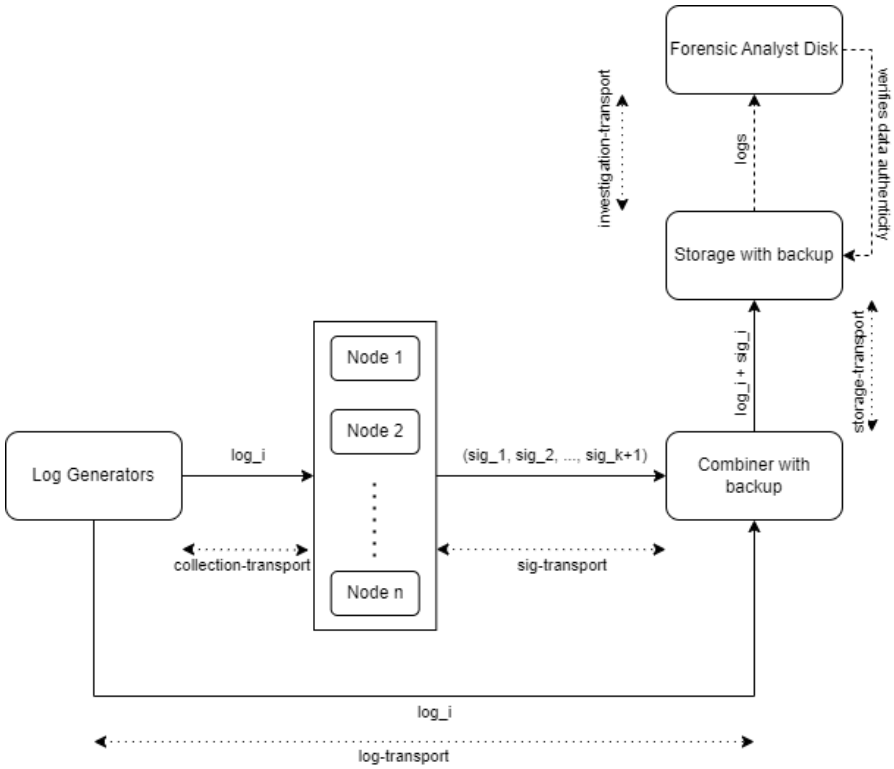


Figure 2.5: The model with threshold signatures

Given that the (k, n) scheme is used, where n is the number of parties, and k is a threshold, at least $k + 1$ partial signatures are required to produce a signature. The *combiner* produces a signature and appends it together with the log_i into the storage. The *forensic analyst* can then verify the log signatures on his disk using a public key.

For the rest of this analysis, it is assumed that the *infiltrated attacker* cannot compromise enough nodes to forge signatures and that a single company operates all the nodes. Such design would have security properties as specified in Table 2.2 (impacts in red means that the design proposed by this thesis does better in this regard). The same security properties are assumed for this model as for the original model in the subsection 1.1.1.

The integrity and availability impact remains from the *system administrator* because the assumption was that the single organization operates all the nodes. Regarding the *infiltrated attacker*, only the avail-

Table 2.2: Security properties of threshold signatures

	Ext. A.	**Inf. A.**	**Sys. A.**	**For. A.**
Log Generators	-/-/A	C/I/A	∅/I/A	∅
Collection-transport	-/-/∅	C/I/∅	∅/I/∅	∅
Log-transport	-/-/∅	C/I/∅	∅/I/∅	∅
Nodes	-/-/-	C/-/-	∅/I/A	∅
Sig-transport	-/-/∅	C/-/∅	∅/I/∅	∅
Combiner	-/-/-	C/-/A	∅/I/A	∅
Storage-transport	-/-/∅	C/-/∅	∅/I/∅	∅/-/-
Storage	-/-/-	C/-/A	∅/I/A	∅
Investigation-transport	-/-/∅	∅/-/∅	C/I/∅	∅
Forensic Analyst Disk	∅	∅	∅	∅/-/-

. *C* - confidentiality violation (a threat agent can read logs), *I* - integrity violation (a threat agent can modify logs), *A* - availability violation (a threat agent can make logs unavailable), ∅ - does not apply.

. *Ext. A.* - external attacker, *Inf. A.* - infiltrated attacker, *Sys. A.* - system administrator, *For. A.* - forensic analyst

ability impact remains on the *combiner* and the storage because she could potentially start deleting signatures in both places[5].

This approach nicely demonstrates why integrity and availability impacts remain when the servers are not distributed between multiple organizations.

2.3.3 Design Shortcomings

Apart from the fact that the nodes are not distributed among multiple organizations, the other design shortcoming of this scheme is the used protocol. Shoup's scheme requires a trusted dealer to generate a complete signature key, split it into n shares and each share is distributed into one of the n nodes [15]. This is problematic because it means there is an entity at some point in time that possesses the whole key. However, this can be fixed with the use of a better protocol.

5. This does not happen on nodes because the assumption states that she cannot compromise enough of them.

2.3.4 Exposure Window

While the paper does not directly consider an exposure window, it tested the performance of its solution. On the I3-2330M processor with 2GB of RAM, they can sign 1000 logs in approximately 2.5 seconds. Same as with the subsection 2.2.4, the implementation is not provided; thus, it cannot be verified.

2.4 Trusted Notary Service

Another approach considered is to leverage already existing proof-of-existence services called **Trusted Notary Services**. "A trusted notary service stores the required evidence. If a dispute about the origin of the provenance data later arises, the notary may be queried to either support or contradict a particular claim" [23]. Examples of such services are opentimestamps.org or poex.io. With these services, you can upload an arbitrary file and obtain a complete timestamp that can later be used to prove that a particular file existed at some point. However, reusing existing services without major redesign is not feasible.

Consider a scenario in the Figure 2.6 where there is a *log generator* producing logs. Then, there is a *log storage* where the logs are placed. On top of that, there is a *trusted notary service*, to which the *log generator* sends the hash of a log h_i and receives a signed timestamp *st* in return. The *forensic analyst* can then verify that the logs existed at some point and were not tampered with using the timestamp and the public key of the *trusted notary service*

Since trusted notary services are typically designed for the general public, anyone can use them to obtain signatures. Previous solutions could verify data origin, but this one cannot unless data origin is part of the signed timestamp. That means an attacker accessing the *log storage* can add arbitrary signed data. Moreover, since the *log storage* is not distributed in this concrete example, he can delete logs. Modification is more tricky because it is still a timestamping service, and the attacker cannot change past timestamps.

Regarding the exposure window, no relevant study was found. However, I performed an experiment at https://opentimestamps.org/, and uploading a document into the blockchain can easily take a

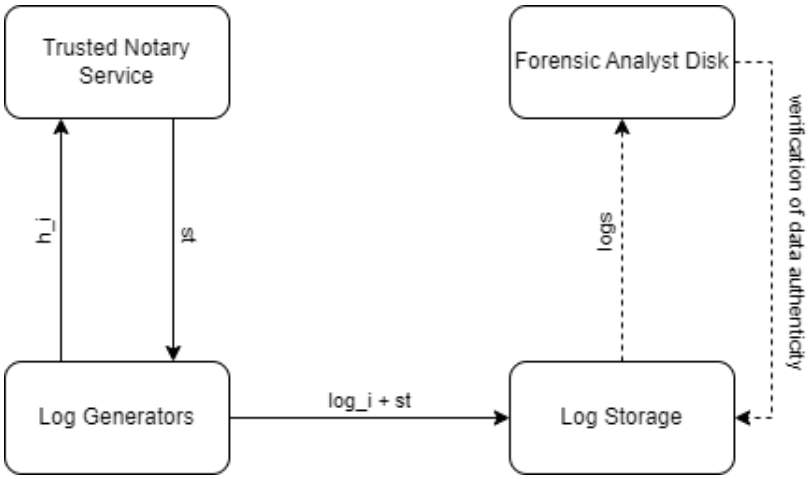


Figure 2.6: The model with a timestamping service

whole day. One could use timestamping services that do not leverage blockchain, but that would mean a trust to a third party is required.

So, in summary, the outcome here is that usage of existing *trusted notary services* is infeasible, so no further security analysis was done.

3 Solution Design

The chapter 1 concluded that the *exposure window* of logs is quite large when stored in the database with baseline security measures. The chapter 2 analyzed state-of-the-art approaches that could improve it, namely blockchain, threshold signatures, and trusted notary service. This chapter explains why threshold signatures were chosen and how they are used.

3.1 Choice of Technology

Starting with the *trusted notary services*, the chapter 2 concluded that it is infeasible to reuse existing solutions and that custom ones must be designed. The other problem with this approach is that the log storage is not distributed. If the legal disputes were against it, they could delete the logs and thus delete the potential evidence. The other issue is that the true security properties would depend on the concrete implementation of the trusted notary service. Simply trusting a third party and its private key might not be enough. The solutions that leverage public blockchain provide better security properties in this regard, but uploading data to public blockchain takes time, which makes the exposure window rather large. Trusted timestamping service could also be implemented using private blockchains or threshold signatures, which makes all three solutions presented in the chapter 2 a trusted timestamping service in a sense.

Leaving the idea of using existing timestamping services, the threshold signatures and private blockchains remain. These seem to be different sides of the same coin because both approaches require a consensus between participating parties; they just use slightly different approaches to get there.

Starting with the blockchain, and [2] in particular, it seems solid from a security point of view. Simply dropping encryption would result in having all security properties as described in Table 1.2. Unfortunately, all the logs would still be in the blockchain structure, so the current real-time log storage and analytics engines would have trouble interpreting the results. This is, however, a problem mostly specific to EngraveChain because, typically, the blockchain is only used to

store hashes of the logs and not the logs themselves. However, even if this is the case, the log storage has to be distributed between multiple organizations because otherwise, an organization can delete the logs, and the availability impact remains. Therefore, it would likely lead to a rather complicated design, so trying a different approach might be worth it.

And this is where MPC, and threshold signatures in particular, come into the picture. Taking [3], splitting the nodes between multiple organizations, and replacing Shoup's protocol with one that does not require a trusted dealer achieves all the security goals from Table 1.2 while keeping the design relatively simple. It seems to be the most promising approach, although what exposure window it achieves remains to be seen.

3.2 Threshold Signatures

It was previously already mentioned that the threshold signatures are of the form (k, n), where n is the number of parties and k is the number of nodes an attacker can compromise and still learn nothing about the key. However, both k and n can be adjusted according to one needs. This is quite a useful feature of threshold signatures because changing these values can achieve interesting designs [24]. Examples:

- **2-of-2** $(1, 2)$ - Two people's savings account. Both people need to sign to withdraw funds.
- **7-of-12** $(6, 12)$ - Self-governing developer teams. The majority of the team needs to agree to fire a team member.
- **3-of-5** $(2, 5)$ - Low-trust donation address. Five trusted people from a project each hold a private key. Three people are required to spend the money, but anybody can donate to the project's address.

The number of configurations and use cases is limitless. For this thesis, the chosen scheme is *3-of-4* or $(2, 4)$, which balances availability and the scheme's robustness.

3.3 Threshold Signatures Protocol

The implementation for this thesis uses the GG20 threshold signature protocol as described in [25], with ZenGo-X implementation[1].

While the protocol is representative of a threshold signature scheme implemented in Rust, it is not a good choice for real-world usage. The implementation contains vulnerabilities that allow an attacker to extract the private key [26], which will not be fixed because ZenGo-X no longer supports it. Moreover, its authors already deprecated the protocol, and a slightly modified scheme should be used instead [25]. Fortunately, this is not a problem for this proof-of-concept implementation.

The downside of the scheme for this study is that it is quite slow, meaning an evaluation of the exposure window size will be affected. The [27] showed that the scheme is one of the slowest regarding key generation and signatures. I will, however, show in chapter 5 that even this scheme achieves reasonably small exposure windows.

3.4 Design

An important observation from examining the state-of-the-art solution in chapter 2 is that to eliminate the impact on availability, the solution needs to distribute logs between multiple organizations. However, as was demonstrated in the section 2.2, these organizations cannot encrypt the logs, so trust between them is required. It also concluded that the blockchain structure is inconvenient from the log analysis point of view, so threshold signatures might be a better approach. Following this reasoning, this chapter proposes a slightly modified approach compared to the already presented.

3.4.1 Signing

The signing is demonstrated in the Figure 3.1. This setting has two organizations: *ORG1* and *ORG2*. Each of the organizations has its pair of *LS servers* (LS stands for Log Signatures), *LS1*/*LS2* and *LS3*/*LS4*, respectively. What is more, each organization has its own database,

1. Available at https://github.com/ZenGo-X/multi-party-ecdsa.

Database1 and *Database2*, where they store both their own and other organization' logs. Lastly, there are *log generators*, which aggregate all different entities producing logs that *ORG1* or *ORG2* own. As was described in section 1.1, these represent firewalls, servers, OSes, etc.

For each generated log, the following holds:

1. One of the *log generators* randomly chooses a subset of three *LS servers*.

2. To the chosen subset, it sends the log it generated (log_i) and the organization's id (org_{id}), to which the *log generator* belongs. The tuple is signed (s_{orgn}) so the *LS server* can verify the origin of the log_i and that the org_{id} corresponds to the given signature. If the signature is incorrect or the org_{id} does not correspond to the organization's signature, the *LS server* refuses to participate in a signature, hence preventing its creation.

3. If the signing succeeds, both organizations store (s_i, log_i, org_{id}) in their database, where s_i is the signature of the $log_i + org_{id}$.

4. If the signing fails, the corresponding *log generator* chooses a different subset and tries again.

With regards to the model in the section 1.1, the *collection server* is replaced with *LS servers*, and the *log storage* is distributed between multiple organizations as *Database1* and *Database2*. The *forensic analyst disk* is excluded from the figure for simplicity; however, he is part of it regardless, see subsection 3.4.2.

3.4.2 Log Integrity Verification

The verification of log authenticity by *forensic analyst* is visualized in Figure 3.2. First, a *forensic analyst* chooses one of the organizations and downloads the log and organization ID (log_i and org_{id}) with its signature (s_i) to his disk (*forensic analyst disk*). Then, he chooses an arbitrary TS server (possibly from a different organization) and sends the $log_i + org_{id}$ with s_i to it. The TS server returns $true$ if the signature is valid or $false$ otherwise.

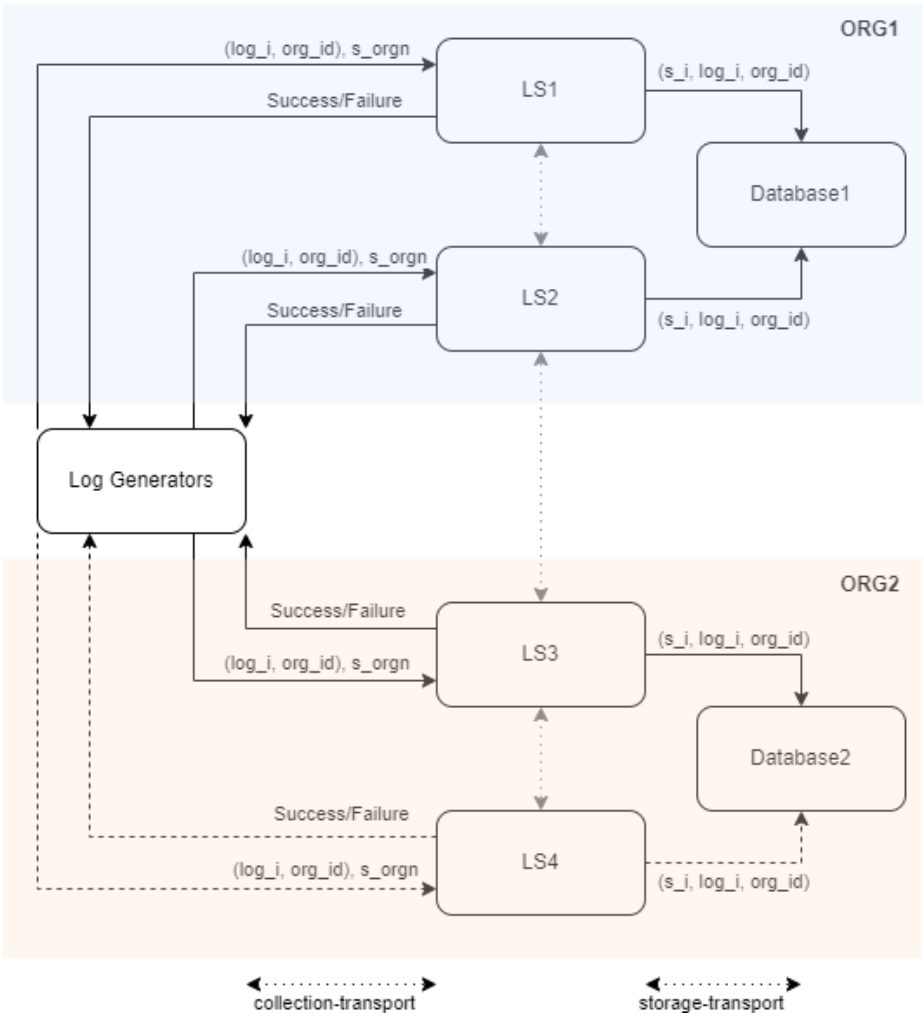


Figure 3.1: Distributed threshold signature solution

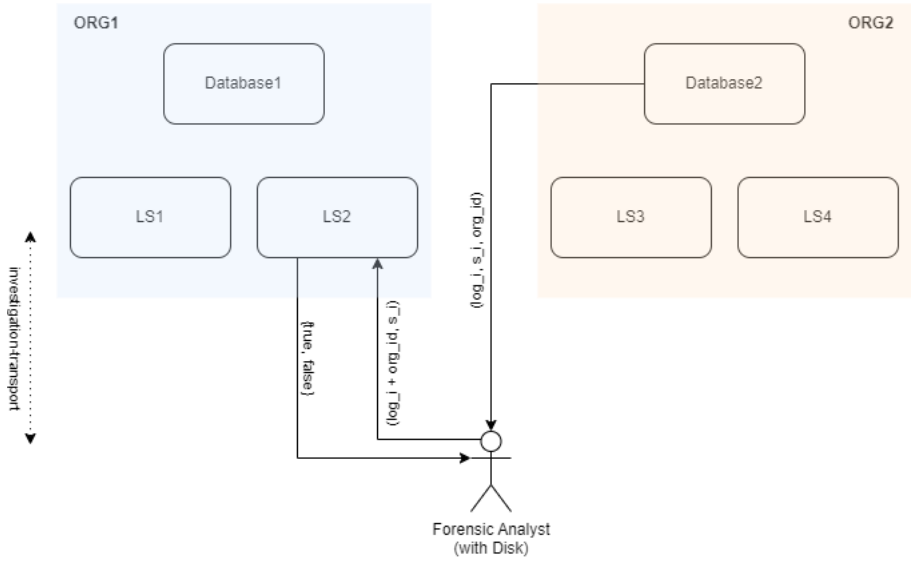


Figure 3.2: Verification of signature by forensic analyst

3.4.3 Threshold

The selected threshold is $(2, 4)$, or in other words, 3-of-4. The goal is to ensure that both organizations participate in a signature while providing some robustness.

If robustness is not as important of a goal, another possible configuration is $(1, 2)$. It would also require both organizations to participate. However, if one *LS server* goes down, the system cannot generate signatures promptly because there is no backup.

One could also argue that in highly critical use cases, the $(2, 4)$ scheme might not be enough because an administrator from a second organization already having access to LS3 and LS4 only needs to compromise one other server (and vice versa). The $(3, 6)$ scheme may be used if this is the concern.

The performance implications of these three different schemes are analyzed in chapter 5.

3.4.4 Security Requirements

Lastly, it should be verified that all the security requirements specified in the chapter 1 hold:

1. The *collection server* (in the form of *LS servers*) is distributed among two organizations that are unlikely to cooperate. Each organization has a backup *LS server*, so it is hard to bring it offline.

2. Both organizations need to participate to produce a signature; see subsection 3.4.3.

3. Both *databases* and *forensic analyst disk* treat the log without signature as invalid.

4. The *log storage* is distributed between two organizations and stores the proof of consensus (which is the signature).

5. The third party can check if the logs are complete and equal to what is in the databases by checking against two copies owned by different organizations. What is more, the validity of the signature is verifiable; see subsection 3.4.2.

6. The communication between entities is encrypted.

3.5 Security Analysis

This design offers all the security properties the thesis set to achieve; see Table 3.1. Once the log reaches *LS servers*, there are strong security guarantees that neither of the threat agents defined in section 1.5 can tamper with the logs nor make them unavailable.

When the log is generated on one of the *log generators*, the *infiltrated attacker* and the *system administrator* can either tamper with the logs generated or with the log generation process itself. Both threat actors can also modify all the logs during the transmission to *LS servers*.

However, given that no threat agent can compromise enough *LS servers* to forge a signature, there are no integrity and availability problems after in the *storage transport* and *databases*. On the *LS servers*, no threat agent can modify logs on enough *LS servers* to create a signature of a forged log. Moreover, no high-ranking individual in an organization can order her *system administrator*s to remove the logs because she would need to do so in both organizations. However, a

Table 3.1: Security properties of proposed design

	External Attacker	**Infiltrated Attacker**	**System Administrator**	**Forensic Analyst**
Log Generators	-/-/A/	C/I/A	∅/I/A	∅
Collection-transport	-/-/∅	C/I/∅	∅/I/∅	∅
LS servers	-/-/-/	C/-/-	∅/-/A	∅
Storage-transport	-/-/∅/	C/-/∅	∅/-/∅	∅
Databases	-/-/-/	C/-/-	∅/-/-	∅/-/-
Investigatio-transport	-/-/∅/	C/-/∅	∅/-/∅	∅
Forensic Analyst Disk	∅	∅	∅	∅/-/-

. *C* - confidentiality violation (a threat agent can read logs), *I* - integrity violation (a threat agent can modify logs), *A* - availability violation (a threat agent can make logs unavailable), ∅ - does not apply.
. *Ext. A.* - external attacker, *Inf. A.* - infiltrated attacker, *Sys. A.* - system administrator, *For. A.* - forensic analyst

system administrator can prevent signatures by bringing both *LS servers* owned by his company down, hence the remaining availability impact.

Regarding the *investigation-transport* and *forensic analyst disk*, the *forensic analyst* cannot tamper with the logs because he cannot forge signatures, and the log integrity can be checked against two sources, *Database1* and *Database2*.

DDoS attacks are possible on *log generators* themselves. On *LS servers*, it is considered infeasible to DDoS enough of them (if the number of servers is unsatisfactory, the number can always be increased). On *databases*, an attacker can possibly prevent one of them from getting logs, but the signing will still work, and other organizations will have the evidence (desynchronization between two databases will occur; if that is the concern, backup databases in each organization can be added). Attacking the *forensic analyst disk* is infeasible because this device is meant to be mostly offline and only allows the *forensic analyst* to verify log integrity as part of the investigation procedure.

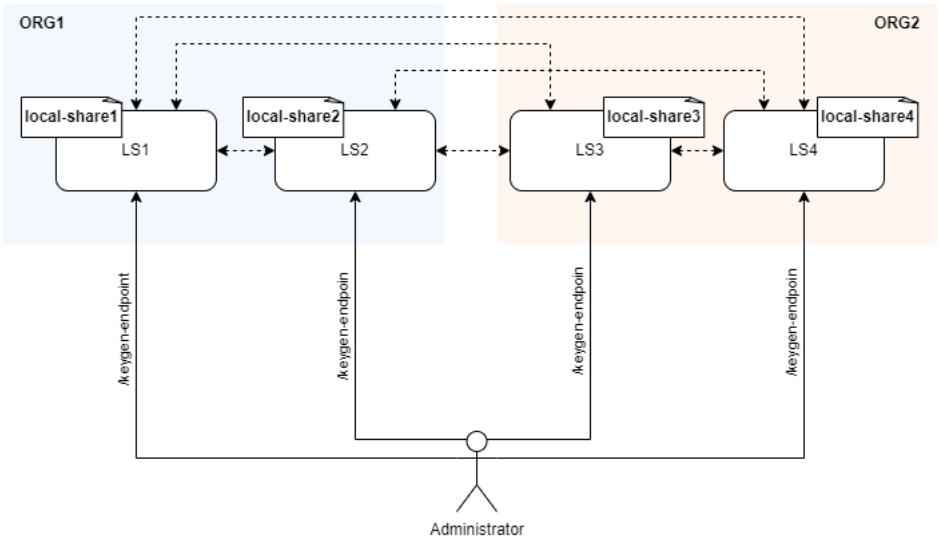


Figure 3.3: A key generation without trusted dealer

3.6 Key Generation

Previously, the thesis pointed out that [3] used required a trusted dealer and set out to fix the issue with a better protocol choice.

The utilized GG20 does not require a trusted dealer. Instead, the parties have to participate with each other to create their local share; see Figure 3.3.

In this case, each server exposes a public endpoint for key generation. Then, an Administrator can trigger this endpoint on all servers. By communicating with each other, each of them can generate their local share ($local\text{-}share\{i\}$). This way, at no point in time would there be a party that would possess the whole key.

3.6.1 Key Rotation

Regarding keys, it is essential to consider how often to rotate keys. Using a single key for prolonged periods is risky; if the key is compromised, the whole period during which the key was used could be tampered with. It is, therefore, necessary to periodically update the key.

An important aspect that has to be kept in mind is that it is crucial to keep the public part of every rotated key so the log integrity can still be verified in the future. The public part should then be stored in another database, and both organizations should save their public parts so there is no single point of failure. Moreover, since these are public parts, they can be stored anywhere publicly.

The period itself is application-specific. In some scenarios, rotating the keys as often as every 10 minutes might be appropriate; in others, a 24-hour period would suffice.

3.7 Design Shortcomings

For the design shortcomings, there are two possible problems that conflict with each other. First, the number of organizations might be insufficient to provide reasonable security guarantees. The other is that it cannot easily scale. Fortunately, with slight modifications, they can be fixed.

3.7.1 Scalability

A most important shortcoming is that it does not scale well when applied to more than two organizations. It is hard to find such a threshold that would require all organizations to participate while also being robust.

For example, each organization could have two *LS servers* in a three-organization setting. But for all parties to participate, the $(4, 6)$ scheme has to be used. Unfortunately, that would mean that two organizations would need to use all their servers, and if one is down, the signatures cannot be created.

To adjust for that, a slightly different approach would be required. In the three-organization setting, each organization could have only a single *LS server* (and possible backup that would share the same key share for robustness). With this setup, $(1, 3)$ setup could be used, where two organizations are required to produce a signature, but not all three have to participate. This can scale well to an arbitrary number of organizations; the only drawback could be decreasing performance.

3.7.2 Colluding Organizations

One could argue that having two organizations is not secure enough because they could collude. In that case, one organization could be replaced by some trusted entity, making this collusion more unlikely.

This could even scale; suppose there is a state that offers its organizations the possibility to opt-in to share their logs with them, and in return, their logs could be used as proof in legal settings. Such a government entity would have plenty of *LS servers*. If an organization opt-in, two of those servers would be reserved, the organization would have the other two, and they will use the scheme described in this chapter. The process would be the same with other organizations opting in, except this central government entity would reserve two other servers and use different keys (if the servers are performant enough, using different keys on the same servers might be sufficient).

3.8 Summary

This chapter proposed a solution for distributed log storage that could pave the way for using logs as potential digital evidence. Guided by an analysis from chapter 2, it designed a system that eliminates problems with direct log storage on blockchain as was discussed in section 2.2 while also eliminating the problem with the need for a trusted dealer, from which the [3] suffered. Then, with the security analysis, the thesis concluded that it achieved all the goals specified in the section 1.8. Lastly, the thesis discussed the potential scaling problems and insufficient security guarantees that the solution may suffer from and suggested possible fixes for future work.

4 Implementation

This chapter explains a proof-of-concept implementation that is provided with the thesis. It starts by explaining the core part of it, the log signing server, and provides a reader with an overall idea of its inner workings. Then, it looks at how the solution was tested. Lastly, it briefly looks at the deployment options.

4.1 Log Signing Server

The chapter 3 introduced *log signing servers*, or *LS servers* for short. These represent the core part of the solution since they are a building block for threshold signing. This section explains their implementation and later uses it to study the exposure window[1].

4.1.1 Technology

As the primary language of the implementation, the **Rust** was chosen. There are two main reasons for the choice despite it still being a relatively recent and evolving language. The first is that it provides a speed comparable to C/C++ while solving most memory management problems that stem from using these languages [28] [29]. As such, it is becoming a popular choice for security-critical projects. Moreover, a Rust can safely handle the concurrency [30].

From a design standpoint, the server exposes all of its functionality to the user via REST API because it is a de facto standard on the web, meaning there are typically good frameworks to work with. There are plenty of good options to choose from in Rust; the Rocket framework (https://rocket.rs/) was chosen for this implementation.

4.1.2 Server Design

The core of the design is the *LS server*, meaning a compiled version of the code is a single server of the threshold signature scheme, not the

1. Apart from the *LS servers*, it also introduced databases where the logs are stored. Since they do not provide any special functionality other than the storage itself, they do not bring much value for evaluation later. As such, they are not implemented.

whole scheme itself. The server exposes API endpoints for a user as well as endpoints for inter-server communication. It is parametrized via command line arguments, determining its identification in the scheme, the scheme it is part of, and network parameters:

```
$ .\log-signing-mpc.exe server-id HTTP-port TLS-
   port threshold number-of-parties
```

- **Log-signing-mpc.exe**: The server. For clarity, the chapter further refers to it as `lsm.exe`.
- **Server-id**: A unique server id.
- **HTTP-port**: A public-facing HTTP port on which the devices can make requests.
- **TLS-port**: An internal TLS port that the servers use internally to communicate with each other.
- **Threshold**: A threshold value in the signature scheme.
- **Number-of-parties**: A number of parties in the signature scheme.

To illustrate it in the example, the 3-of-4 scheme requires four servers. To obtain those servers, one has to run four separate servers as follows:

```
LS1: .\lsm.exe 1 8001 3001 2 4
LS2: .\lsm.exe 2 8002 3002 2 4
LS3: .\lsm.exe 3 8003 3003 2 4
LS4: .\lsm.exe 4 8004 3004 2 4
```

At this point, the servers are still unaware of each other presence. They only expose endpoints through which they can be controlled and paths through which they can communicate, see Figure 4.1.

For the setup to work, the *server-id* must be unique in all instances. The typical rules apply for the ports: you cannot open the same port on a single device more than once. The dashed lines represent communication paths taking place via ports 3001, 3002, 3003, and 3004. The *threshold* with the *number-of-parties* must be the same in all instances and can be **arbitrary**, as long as the scheme is valid. This means that the implementation supports schemes beyond 3-of-4.

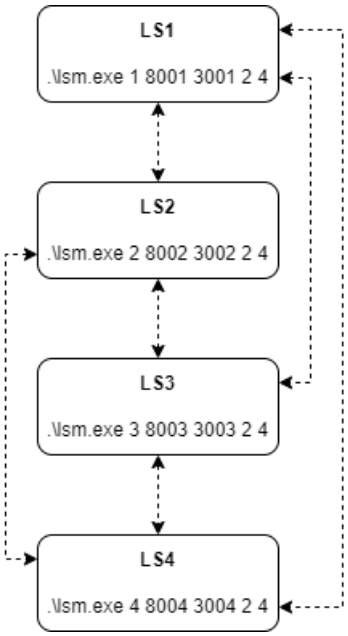


Figure 4.1: Server setup

Regarding the endpoints, while inter-server communication is happening over HTTPS, communication with the outside world is done through HTTP. This is, of course, not ideal and was done for simplification purposes only. In the real world, all communication would be done over HTTPS.

4.1.3 Endpoints

The most important part of the *LS server* is the REST API that it exposes because it provides *key generation*, *signing*, and *verification* functionality. Other than that, it also exposes API that the servers themselves use for communication with each other. The explanation of each endpoint follows.

Key Generation

The key generation process takes place without a trusted dealer. Users can generate keys by triggering the */key_gen* endpoint on all participating *LS servers*. Triggering it starts an internal communication between

the servers and generates a partial key stored in the file *local-shareX.json*. The process cannot be repeated twice unless someone removes the key manually from all the instances (meaning keys cannot be easily overwritten).

For the key generation endpoint, the scheme looks as follows[2]:

```
$ curl.exe -X POST localhost:{HTTP-port}/key_gen
  /<room_id> -d "OTHER-SERVER-URLS-WITH-THEIR-
  TLS-PORTS}"
```

- **localhost:HTTP-port**: A machine address to which the request is sent. A *localhost* can be replaced with any URL, and *HTTP-port* represents the public-facing port as specified in the subsection 4.1.2.
- **/key_gen/<room_id>**: The *key_gen* is the endpoint's name, and the <*room_id*> is the room's id. Rooms are explained later in subsection 4.1.4. For now, let's assume it's fixed to be one on all of the servers.
- **OTHER-SERVER-URLS-WITH-THEIR-TLS-PORTS**: This represents a data part of the request and contains comma-separated addresses of other participating servers.

In the 3-of-4 case, the usage is demonstrated in Figure 4.2. The administrator sends parallel requests to all four servers on the */key_gen* endpoint. With the requests, the administrator also passes the list of addresses of all other participating servers (this is important because, as the previous section mentioned, the servers are unaware of each other's presence). The key generation is considered successful if all four servers return status 200. If at least one of them returns status 403, the key generation is considered unsuccessful.

Signing

The signing is triggered similarly to key generation. The *log generator* chooses a subset of servers and triggers a */sign* endpoint on them.

2. For this and all of the following endpoints, the examples will use the *curl* (https://curl.se/) tool to demonstrate their usage. What is more, all of the requests will be *POST* requests, so it will not be further clarified.

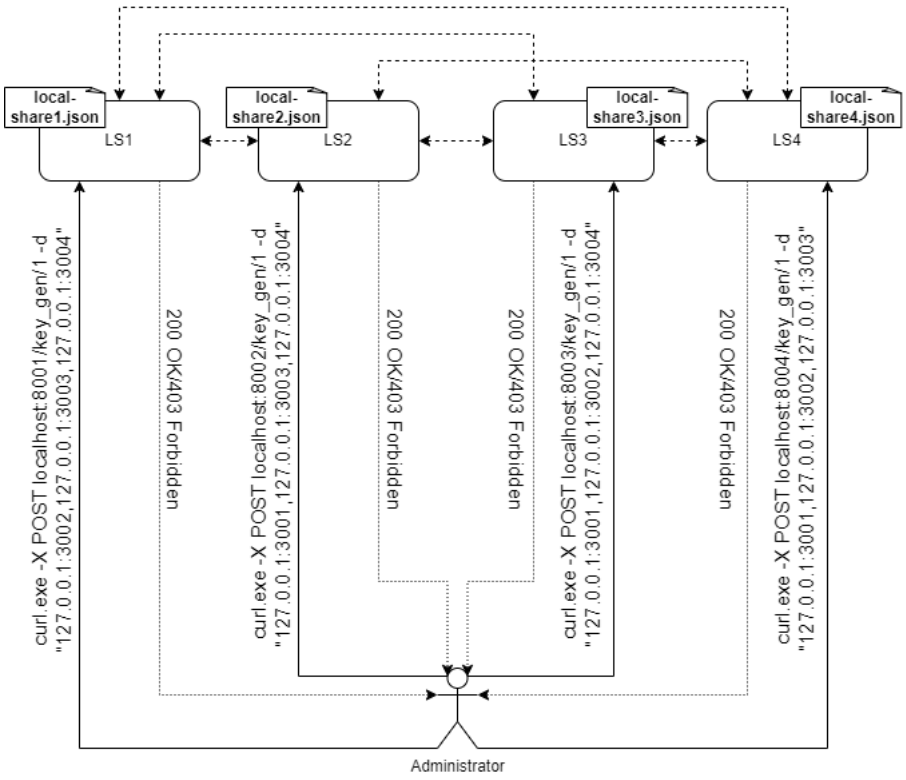


Figure 4.2: Implementation of /key_gen endpoint

Upon the trigger, the servers start participating together to produce a signature. In the request, the log-generating device has to specify the addresses of other parties as well as the data with the timestamp. Data and the timestamp must match on all servers for a signature to be successful.

The scheme looks as follows:

```
$ curl.exe -X POST localhost:{HTTP-port}/sign/<
   room_id> -H "Content-Type: application/json"
   '{JSON-DATA}
```

- **localhost:HTTP-port**: Same as the key generation part.
- **/sign/<room_id>**: The *sign* is the endpoint's name, and the <*room_id*> is the room's ID.
- **JSON-DATA**: The JSON data is sent together with the request. Compared to the key generation, this endpoint requires its data to be in the JSON format because they are more complex.

The JSON structure is of the following form:

```
{
  "participants":
  [
    {
      "server_id": {SERVER-ID},
      "url": {SERVER-URL}
    },
  ...
  ],
  "data_to_sign": "{SHA256-HASH-OF-DATA}",
  "timestamp": "{TIMESTAMP}"
}
```

It consists of participant objects that specify the server ID and server URL. On top of that, a *log generator* has to specify the SHA256 of the log together with the UNIX timestamp. An example sent to LS1 servers looks as follows:

```
{
  "participants": [
    {
      "server_id": 2,
      "url": "127.0.0.1:3002"
    },
    {
      "server_id": 3,
      "url": "127.0.0.1:3003"
    }
  ],
  "data_to_sign": "7b736f6d652c6172...",
  "timestamp": "1701430841"
}
```

The timestamp serves as proof of when the log was signed. In the future, a *forensic analyst* can verify the time when the log was signed, meaning the implementation serves as a timestamping service. It verifies that it holds for the current timestamp that $ct + 5min \geq timestamp \geq ct - 5min$, where ct is the current time. If it does not hold, the *LS servers* refuse to sign.

The flow of a signature in the 3-of-4 case can then be seen on Figure 4.3. A *log generator* device chooses a subset of 3 devices and sends a request to them. To each of them, it sends the IDs and addresses of two other participating parties, the SHA256 hash of the log, and the UNIX timestamp. A chosen subset of servers participate together, and if the signature is successful, all 3 of them return status 200 with the signature. Otherwise, they return status 400, and a device has to either retry or choose a different subset of devices.

Verification

Verification is the most straightforward of the three since a request to a single server is required. The endpoint has the following schema:

```
$ curl.exe -X POST localhost:{HTTP-port}/verify
   -d "{SIGNATURE-AS-HEX-STRING},{SHA256-HASH-OF-
   DATA},{TIMESTAMP}"
```

- **localhost:HTTP-port**: Same as the key generation part.

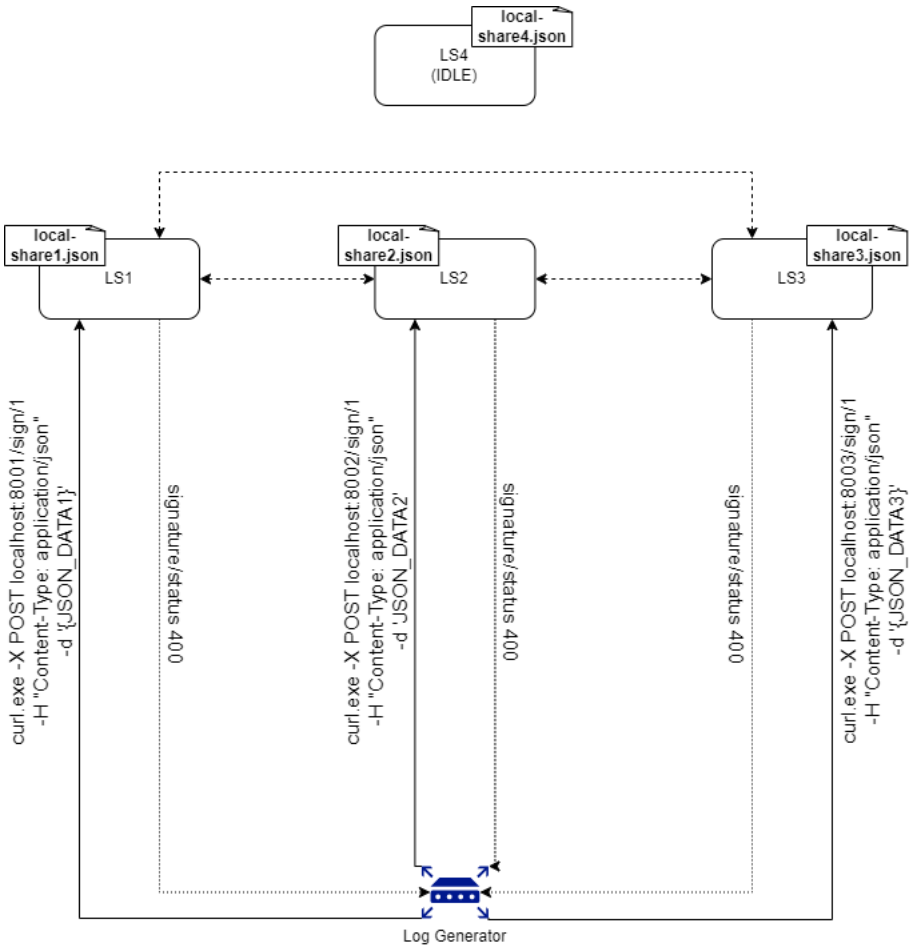


Figure 4.3: Implementation of /sign endpoint

- **/verify**: The name of the endpoint
- **SIGNATURE-AS-HEX-STRING**: Signature obtained and converted to hex format.
- **SHA256-HASH-OF-DATA**: The SHA256 of the log signed.
- **TIMESTAMP**: The UNIX timestamp.

To verify a signature, a *forensic analyst* chooses an arbitrary node and sends a signature for verification, see Figure 4.4. He sends the signature in as a hex string, the hash of the log, and the timestamp to one of the *LS servers* and receives status 200 in case the signature is valid and 400 otherwise. To increase the speed, the *forensic analyst* can verify using all the servers in parallel.

Inter-server Communication

The last endpoint accessible is the */receive_broadcast/<room_id>* endpoint. It is not meant as an endpoint to be public but rather as an endpoint through which the servers exchange messages.

In this proof-of-concept implementation, the simplified approach is taken. When one node needs to send a message to the other node, it sends it to all the participating servers. It specifies the sender's ID, the receiver's ID, and the message it wants to send. The receiver then filters out the messages not meant for them. In the real world, rather than broadcast, a more targeted approach would be better. However, even with this implementation, other servers should learn nothing about the private shares of other servers.

The purpose of *<room_id>* is explained in the subsection 4.1.4.

4.1.4 Asynchronous Design

The crucial observation about the logs is that they are independent, meaning they do not need to be processed in the sequence. Parallel processing of the logs could thus improve the performance.

The gg20 protocol does multiple rounds of communication between servers both when the keys are generated and when the message is signed. Consequently, there are multiple exchanges of information between the servers, and there is a need to distinguish between

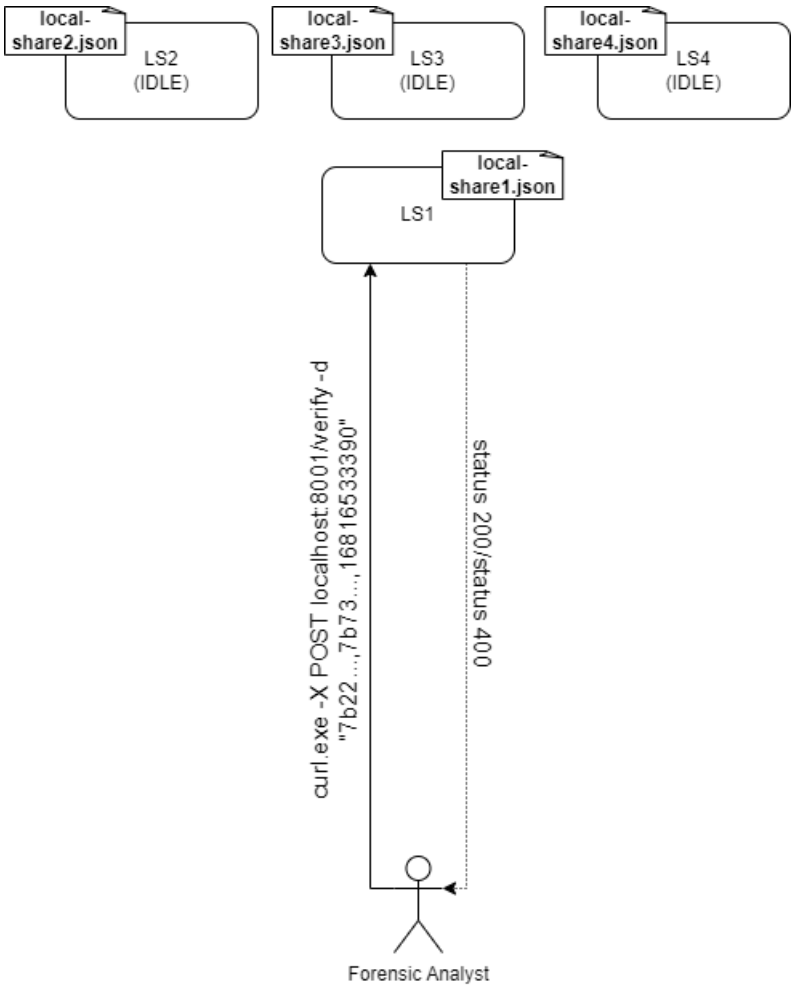


Figure 4.4: Implementation of /verify endpoint

them (communication regarding the signature of log 1 must be distinguished from communication regarding the signature of log 2 and so on).

The solution leverages the concept of *room*s to allow parallel signatures. Each separate operation takes place in a different room, and the room control is provided to the user via the API. To choose a room for an operation, a user specifies the room ID as the last part of the *keygen* and the *sign* endpoints. For example, to sign two logs in parallel, a user can trigger two endpoints, */sign/1* and */sign/2*, where the room numbers are arbitrary (but cannot be the same).

For the concept to work, the servers must also distinguish rooms when they receive data from other servers, not the users. The *room_id* in the */receive_broadcast/<room_id>* endpoint mentioned in the previous section is meant for that. When the servers sign the logs in room *n*, all communication happens in that room, so they should send and receive messages regarding that signature to and from */receive_broadcast/n*.

4.2 Testing

The implementation contains three types of tests: *unit tests*, *integration tests*, and *performance tests*. Unit tests are used to test small components, integration tests verify the correctness of public API functionality, and performance tests focus on the performance evaluation of provided operations.

4.2.1 Technology

The **Rust** is used for the unit tests since the implementation is written in that language. Moreover, Rust provides solid tools for writing automated tests as part of the language.

However, **Python** is used for the integration and performance tests. It is because while Rust's performance and safe memory/thread management properties are great, writing the code is not as efficient. Python was chosen because it provides mature libraries for asynchronous testing while being quite efficient to write in. The latest

version of Python was used, which, at the time of this writing, was 3.12.0[3]. The testing framework used is called **Pytest**[4]

While this thesis's implementation is considered proof-of-concept, implementing an application for threshold signatures is rather complex. It mainly stems from the fact that the threshold signature application must be distributed, and network communication with proper synchronization must be established between participating nodes. The problem is that even minor changes to the core of the implementation can introduce unexpected errors that are hard to track. As such, a pipeline with automated tests is highly beneficial. Since the project was already hosted on GitHub, **GitHub actions**[5] were used for pipelines.

4.2.2 Integration Tests

The tests are written for two schemes, $(1,3)$ and $(2,4)$, to provide reasonable guarantees that the implementation works for an arbitrary (t,n) scheme. That means that each test is written in two forms, one for the $(1,3)$ scheme and the other for the $(2,4)$ scheme.

The *key generation* tests verify that keys can be successfully generated and that further requests for key generation are rejected once they are generated. The *signing* tests verify that the signature can be created in all possible combinations of subsets for the given threshold (e.g., in $(2,4)$, it would be $(1,2,3)$, $(2,3,4)$, $(1,2,4)$, and $(1,3,4)$). Other than that, they verify whether the parallel signatures are working by signing five signatures in parallel. The *verification tests* verify that the endpoint returns a 200 OK error code if the signature is valid and 400 Bad Request otherwise.

4.2.3 Performance Tests

The standard *Time* module[6] was used to evaluate performance since its precision was satisfactory for the tests conducted. Each operation tested, namely key generation, signing, and verification, has a corresponding Python script. Each of these scripts can be parametrized

3. Available at https://www.python.org/downloads/.
4. Available at https://docs.pytest.org/en/7.4.x/.
5. At the time of this writing, it provides 1000 minutes of computation time per month for free.
6. https://docs.python.org/3/library/time.html

via command-line arguments to specify the number of trials of the test plus other parameters specific to a concrete script. The results are then presented to the user at the standard output, including all the intermediary results.

4.3 Deployment

As was already mentioned, the application that implements threshold signatures is rather complex, so it can be pretty complicated to set it up. The application needs to be compiled with a proper version of the compiler, it has to be run with correct parameters on all servers, and it even requires the creation of certificates for secure communication.

Docker is the technology of choice here because it allows hiding all the setup complexities into a single command and can be directly reused in GitHub action pipelines. Additionally, it will enable platform independence (although the plain solution itself should run both on Linux and Windows).

Docker Compose is used to simplify the setup even further by allowing the creation of whole schemes in a single command, including networking.

4.4 Summary

In summary, the implementation, crafted as a proof-of-concept for evaluation purposes, adheres to best development practices. It provides a solid REST API design featuring support for asynchronous operations, has reasonable test coverage of provided functionality, and is equipped with well-established CI/CD pipelines. As such, it hopes to provide robust building blocks for further experimentation with threshold signatures and distributed protection of potential forensic evidence. For more information on how to use the implementation, refer to the Appendix A.

5 Evaluation

This chapter aims to evaluate the performance with the exposure window in particular and complement the security evaluation discussed in the section 3.4. For the evaluation, the proof-of-concept implementation as described in the chapter 4 is used. It was already mentioned in the section 3.3 that the protocol chosen for PoC implementation is relatively slow. The results thus present an upper bound on what is possible. However, the chapter argues that the proposed design is suitable for real-world usage even with the relatively slow implementation.

5.1 Testing Environment

The virtual machine is used for the evaluation because it creates a stable testing environment. What is more, it allows the assignment of computational resources to the system so that different configurations can be tried. The official Windows 11 VirtualBox image was used for the evaluation[1].

The host system that runs virtualization has an *11th Gen Intel(R) Core(TM) i9-11950H @ 2.60GHz 2.61 GHz processor* with *32GB of memory*. The processor has 8 cores and 16 available threads. The following two configurations are tested:

- *Configuration 1*: 2 cores with 2GB of memory
- *Configuration 2*: 8 cores with 8GB of memory

It is important to note that the implementation was tested on a local machine, meaning the network communications delays are not considered. While the ECDSA operations are likely to be more of a bottleneck than the network delays, the results might be slightly distorted.

1.
URL: https://developer.microsoft.com/cs-cz/windows/downloads/virtual-machines/
Name: WinDev2311Eval.VirtualBox.zip
Hash: E8D1FC6C26F56272B5D3DCDB864E8BD9C6DC2BC3647723FAB64BE65179297A49

5.2 Evaluation Targets

Three performance targets were evaluated:

1. Average time of key generation.
2. Average time of signature.
3. Average time of verification.

The **key generation** is not the most crucial performance-wise if the key is rotated relatively infrequently (e.g., once a day or even once an hour). However, for more frequent rotations, it might be worth looking at the average time of a key generation to get an idea of how small a key rotation period can be.

The **signing** time is the most crucial part of the three because it directly corresponds to the exposure window size. The time to complete a signature is the time when the log is not signed and thus potentially susceptible to tampering. Therefore, the low average signing time is crucial for the feasibility of the proposed solution.

Lastly, there is the average time of the **verification**. While not really crucial performance-wise, it is nonetheless worthwhile to evaluate. The average time provides a look into whether verifying the signatures on a log-by-log basis is feasible or whether signatures of larger batches of logs are required.

5.3 Threshold Setups

The tested setups are the following:

- **3-of-4**: The central part that was demonstrated in the chapter 3. This represents the setup where either two organizations are participating or an organization is participating together with a trusted entity.
- **4-of-6**: The same setup as in the previous case (two organizations or an organization and a trusted entity), but each organization has three servers instead of two to improve robustness.
- **5-of-8**: The setup where there are eight organizations, each with a single node, and five of them have to participate to produce a

signature. This is another possible approach; it is included here mainly to observe if increasing the threshold and number of parties adversely affects the performance.

5.4 Methodology

For each of the configurations (section 5.1) and each of the threshold setups (section 5.3), meaning six different configurations, the following methodology was used:

1. Generate keys ten times and compute the average of the results.

2. Compute the *time duration* of the single signature ten times in a row and average the results.

3. Compute 5, 10, 20, and 30 signatures in parallel ten times in a row, average the results, and provide a *log/second ratio*.

4. Verify 1000 signatures ten times in a row, average the results, and provide a *log/second ratio*.

The first run of the tests was run without data collection to warm up the machine.

5.5 Results

5.5.1 Key Generation

The most crucial aspect of the key generation is its duration; see Figure 5.1. In a 3-of-4 scheme, the key generation takes six to seven seconds, depending on the configuration used, meaning key rotation cannot be much lower than that. Fortunately, it is good enough for real-world usage; keys will likely be rotated less frequently. As expected, the key generation time increases as the number of servers increases, as can be seen for 4-of-6 and 5-of-8 setups. For comparison, vanilla ECDSA takes about 0.85 seconds.

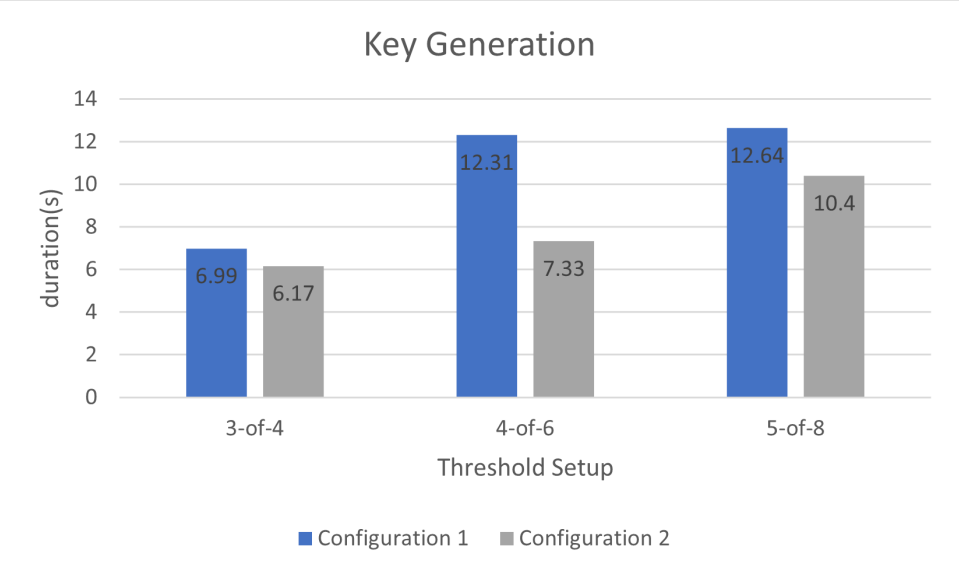


Figure 5.1: A key generation duration (lower is better)

5.5.2 Signing

The signing part is the most important because it directly corresponds to the exposure window. Based on the experiment, the time to produce a single signature is around two seconds on all tested configurations and threshold setups; see Figure 5.2. For comparison, vanilla ECDSA takes around three milliseconds, but the increased duration for threshold signature is expected [31].

The only other solution I found that mentions a term close to the exposure window is [32]. The authors call it a *delay* and define it as "The average delay from the time a transaction is generated to the time it is packaged to a block" [32]. On a 4-node setup, the delay is around six seconds, considerably worse than the approach proposed by this thesis. However, this likely stems from the fact that blockchains tend to have large block times[2] to achieve decentralization and security [34]. To assess the performance of the blockchain, authors usually refer to *transactions per second* or *TPS*, calculated as

2. "Block time measures the time it takes the miners or validators within a network to verify transactions within one block and produce a new block in that blockchain" [33]

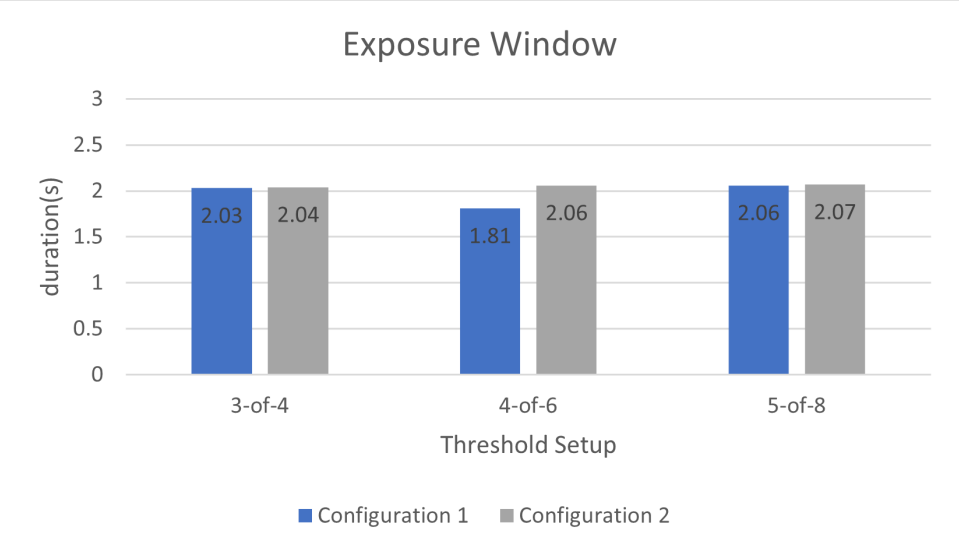


Figure 5.2: Exposure window (lower is better)

$TPS = (block_size/transaction_size)/block_time$. That means that although the exposure window is relatively large, the system can process many transactions in that block period. The [34] achieved around 70 TPS, although some other authors, such as [2], claim to reach around 300 TPS. If the results from Figure 5.2 would be expressed as a *log/second ratio*, which is the number of logs the system can handle per second (which are slightly different but comparable metrics), the proof-of-concept implementation would achieve around 0.5, which is much worse.

Drawing upon the acquired knowledge, an examination was conducted to ascertain the potential enhancement of results through the utilization of the asynchronous design explained in chapter 4. The log to the second ratio of sending 1, 5, 10, 20, and 30 signatures in parallel was calculated, and the results are visualized in the Figure 5.3 and Figure 5.4. The *Y-axis* is adjusted so that both figures are directly comparable.

Unsurprisingly, the asynchronicity did not help in Configuration 1, which has only two cores to work with. However, in Configuration 2, the ratio increases as the number of parallel signatures increases, reaching around 3.14 log/second in the 3-of-4 case. However, even

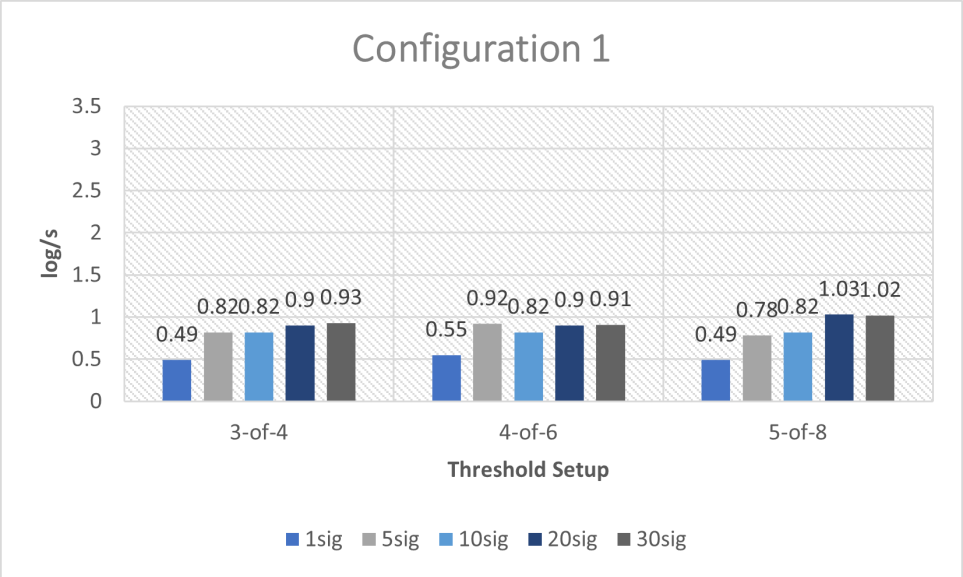


Figure 5.3: Configuration 1: log/second ratio (higher is better)

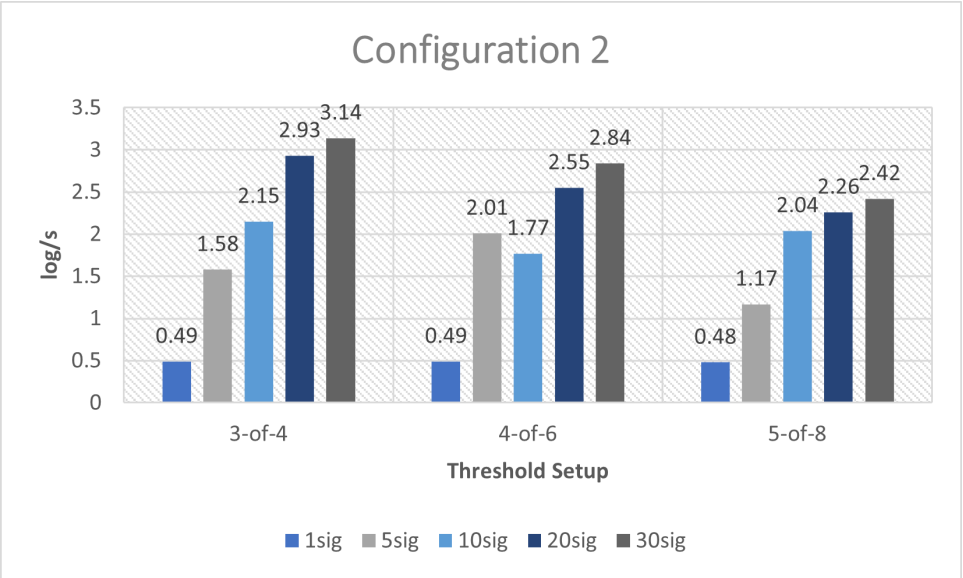


Figure 5.4: Configuration 2: log/second ratio (higher is better)

though the log/second ratio has been improved almost six times, it is still much worse than the blockchain.

Despite the results not being very favorable, I do not consider a TPS or log/second ratio to be a more relevant metric than the exposure window. That is because logs can be signed in batches rather than one by one. Suppose a 2-second exposure window is good enough; the logs can be collected at the servers for this period and then signed together. This implies that as the traffic volume increases, the batch size also increases while the timeframe for exposure remains constant. The better TPS would only mean smaller batches and not necessarily a smaller exposure window. Regarding the blockchain, block time can be decreased and thus reduce the exposure window size, although it remains a question of how small block times would still be reasonable.

In summary, the blockchain is doing much better when the TPS is considered. While the performance of the threshold signature scheme concerning TPS can likely be significantly improved by leveraging better protocol, the chapter argues that it is not, in fact, a relevant metric. That is because a large TPS does not mean a small exposure window, as was demonstrated with the block times. For these reasons, an exposure window size metric should be used to compare solutions.

5.5.3 Verification

The last part of the evaluation was concerned with the verification performance. Here, the test sends 1000 logs to each combination of configuration and threshold setup and measures the time it took to verify it. Then, it calculates the log/second ratio to see how many logs per second the solution verified; see Figure 5.5.

Unfortunately, the measurements were highly unstable in the testing environment. The numbers jumped from 200 log/s to 60 log/s in two consecutive measurements, regardless of the setup. Even more surprisingly, the configuration with fewer cores and less memory was doing consistently better. Therefore, it is advisable to approach the findings with a degree of caution.

Fortunately, since the resulting signature from the threshold protocol is indistinguishable from standard ECDSA, the standard verification mechanism can be used. As such, the papers analyzing the performance of ECDSA verification could provide an estimate of what

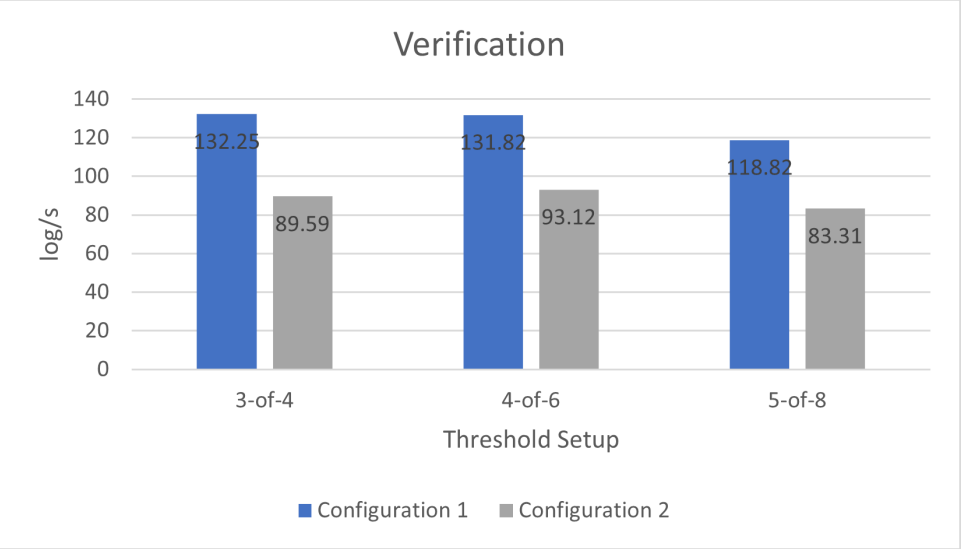


Figure 5.5: Verification log/second ratio (higher is better)

is achievable. According to the [31], the verification on *secp256k1* (the same curve as the GG20 implementation uses) takes around 1.6 milliseconds, which would be around 600 logs per second. This is slightly better compared to the proof-of-concept implementation, which, on average, achieved approximately 100 logs per second, so a single signature takes around 10 milliseconds.

5.6 Summary

This chapter evaluated the performance of key generation, signing, and verification in two possible system configurations. It only tested the solution on the local machine, so the results represent an upper bound.

For key generation, the thesis showed that the number of servers directly impacts its performance. It analyzed the duration of the operation and showed that it should not be a limiting factor, even when the key rotation is relatively frequent.

In the signing part, the thesis demonstrated that the exposure window with the protocol used is around 2 seconds. While hoping to achieve better results, the feasibility of signing in batches remains, and

improvements are likely with a better protocol. Moreover, the thesis showed that performance can be slightly improved by leveraging asynchronous signing since the logs are not dependent on each other.

Lastly, the implementation achieved around 100 logs/second for the verification, which can be improved by parallel signing on all servers. While it is likely not good enough for real-time verification, it is still a solid performance that will not be a limiting factor for investigations. For more information regarding how the evaluation was done, refer to the Appendix A.

6 Conclusion

The ultimate goal of this thesis was to develop a method that can proactively secure the potential digital evidence, minimizing the exposure window. On top of that, it should allow an independent third party to reexamine the evidence and should experimentally demonstrate the exposure window size.

To attain the specified objective, the thesis first developed a threat model and analyzed the security properties of potential digital evidence (logs) under baseline security measures. It then leveraged the model to analyze state-of-the-art solutions: blockchain, threshold signatures, and timestamping services. Based on this analysis, it proposed an approach using the threshold ECDSA scheme, implemented the proof-of-concept, and evaluated the average length of the crucial operations, including the exposure window.

In the evaluation phase, the thesis assessed key generation, signing, and signature verification performance. It determined that both key generation and verification times meet satisfactory levels for securing potential digital evidence. Concerning signatures, the proof-of-concept implementation achieved a 2-second exposure window. This demonstrated comparable or potentially superior performance to blockchains and can likely be improved with better threshold signature protocols. Lastly, the chapter highlighted that even though blockchains can handle larger transaction volumes, their exposure windows tend to be larger in size.

In conclusion, the thesis successfully illustrated the capability of threshold signatures to ensure the integrity and authenticity of potential forensic evidence, emphasizing their potential to reduce exposure windows. On top of that, the thesis provided a proof-of-concept implementation and experimentally demonstrated the exposure window size, which is the metric mostly neglected in this area of research.

A Archive Content

Apart from the thesis itself, the archive with the implementation was provided as part of the submission, which includes the *LS server* implementation, integration/performance tests, GitHub pipeline definitions, Docker definitions, and other supporting scripts. For further guidance, refer to the **README.md**.

A.1 LS Server

The most important part is the *LS server*, which can be found in the *src/* directory. For a more detailed tutorial on how to deploy the implementation or how to try different operations, refer to the *README.md*.

A.1.1 Prerequisities

- Python (I used 3.11.5)
- Docker (I used 24.0.6)
- Docker Compose (I used v2.21.0-desktop.1)
- All python packages can be installed by running `pip install -r ./evaluation/requirements.txt`

Do not forget to add the root of the project into PYTHONPATH, e.g., `$env:PYTHONPATH = "D:\log-signing-mpc;$env:PYTHONPATH"`.

A.1.2 Simple Setup

There are multiple ways *LS servers* can be deployed; this is the most straightforward one that requires the least amount of steps:

1. Build the docker image:

   ```
   docker compose build build-service-24
   ```

2. Run the 3-of-4 scheme:

   ```
   docker compose -f docker-compose-24.yml
      up
   ```

3. From a separate terminal in the archive root directory, run the tests:

```
pytest -k "TestKeyGen24 or TestSigning24
    or TestVerify24"
```

This should demonstrate how the key generation, signing, and verification works.

A.1.3 Operations

To try the operations yourself, I recommend using scripts in the *evaluation/simple_operations* directory. These are:

```
python key_generation.py <number_of_nodes>

python signing.py <threshold> <data_to_sign>

python verification.py <server_id> <
   signed_data> <timestamp> <signature_in_hex
   >
```

The *key_generation.py* takes a number of servers a scheme has and generates keys. The *signing.py* takes the threshold of the scheme with data that you want to sign and prints to the standard output the signature in hex and the timestamp used. The *verification.py* takes the server ID on which you want to do verification, data you signed, and timestamp with a signature provided by the signing.py script. It then prints whether the signature is *valid* or *invalid*.

Examples:

```
python .\key_generation.py 4

python .\signing.py 2 data123

python .\verification.py 4 data123
   1702548777 7b22...
```

A.2 Evaluation

Evaluation tests can be found in the *evaluation/* directory. The *evaluation/tests/* directory contains integration tests that are used as part of the pipelines. The *evaluation/performance* directory contains scripts that were used for performance testing. For further information, refer to the *README.md*.

A.3 Deployment

The solution supports two deployment options:

1. Bare metal
2. Docker

The *Docker deployment* is much simpler to set up but requires Docker. The *bare metal deployment* works both on Linux and Windows. For further information, refer to the *README.md*.

Bibliography

1. DAUBNER, Lukas; MACAK, Martin; BUHNOVA, Barbora; PITNER, Tomas. Towards verifiable evidence generation in forensic-ready systems. In: *2020 IEEE International Conference on Big Data (Big Data)*. 2020, pp. 2264–2269. Available from DOI: 10.1109/BigData50022.2020.9378035.
2. SHEKHTMAN, Louis; WAISBARD, Erez. EngraveChain: A Blockchain-Based Tamper-Proof Distributed Log System. *Future Internet*. 2021, vol. 13, p. 143. Available from DOI: 10.3390/fi13060143.
3. AFZAAL, Muhammad; DI SARNO, Cesario; COPPOLINO, Luigi; D'ANTONIO, Salvatore; ROMANO, Luigi. A Resilient Architecture for Forensic Storage of Events in Critical Infrastructures. In: *2012 IEEE 14th International Symposium on High-Assurance Systems Engineering*. 2012, pp. 48–55. Available from DOI: 10.1109/HASE.2012.9.
4. CHUVAKIN, A.; SCHMIDT, K.; PHILLIPS, C. *Logging and Log Management: The Authoritative Guide to Understanding the Concepts Surrounding Logging and Log Management*. 2012.
5. MADANI, Afsaneh; REZAYI, Saed; GHARAEE, Hossein. Log management comprehensive architecture in Security Operation Center (SOC). In: *2011 International Conference on Computational Aspects of Social Networks (CASoN)*. 2011, pp. 284–289. Available from DOI: 10.1109/CASON.2011.6085959.
6. MATULEVIČIUS, Raimundas. Domain Model for Information Systems Security Risk Management. In: *Fundamentals of Secure System Modelling*. Cham: Springer International Publishing, 2017, pp. 17–30. ISBN 978-3-319-61717-6. Available from DOI: 10.1007/978-3-319-61717-6_2.
7. MARTY, Raffael. Cloud Application Logging for Forensics. In: *Proceedings of the 2011 ACM Symposium on Applied Computing*. TaiChung, Taiwan: Association for Computing Machinery, 2011, pp. 178–184. SAC '11. ISBN 9781450301138. Available from DOI: 10.1145/1982185.1982226.

8. JIANG, Peng; HUANG, Ruizhe; LI, Ding; GUO, Yao; CHEN, Xiangqun; LUAN, Jianhai; REN, Yuxin; HU, Xinwei. Auditing Frameworks Need Resource Isolation: A Systematic Study on the Super Producer Threat to System Auditing and Its Mitigation. In: *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, 2023, pp. 355–372. ISBN 978-1-939133-37-3. Available also from: https://www.usenix.org/conference/usenixsecurity23/presentation/jiang-peng.

9. LIANG, Xueping; SHETTY, Sachin; TOSH, Deepak; KAMHOUA, Charles; KWIAT, Kevin; NJILLA, Laurent. ProvChain: A Blockchain-Based Data Provenance Architecture in Cloud Environment with Enhanced Privacy and Availability. In: *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 2017, pp. 468–477. Available from DOI: 10.1109/CCGRID.2017.8.

10. GUO, Jingjing; WEI, Xuliang; ZHANG, Yuling; MA, Jianfeng; GAO, Huamin; WANG, Libo; LIU, Zhiquan. Antitampering Scheme of Evidence Transfer Information in Judicial System Based on Blockchain. *Security and Communication Networks*. 2022. Available from DOI: 10.1155/2022/5804109.

11. LI, Meng; LAL, Chhagan; CONTI, Mauro; HU, Donghui. LEChain: A blockchain-based lawful evidence management scheme for digital forensics. *Future Generation Computer Systems*. 2021, vol. 115, pp. 406–420. ISSN 0167-739X. Available from DOI: https://doi.org/10.1016/j.future.2020.09.038.

12. LINDELL, Yehuda. *Secure Multiparty Computation (MPC)* [Cryptology ePrint Archive, Paper 2020/300]. 2020. Available from DOI: 10.1145/3387108. https://eprint.iacr.org/2020/300.

13. *What are multi-signature wallets and how do they work?* [online]. https://www.bitpanda.com/ [visited on 2023-10-27]. Available from: https://www.bitpanda.com/academy/en/lessons/what-are-multi-signature-wallets-and-how-do-they-work/.

14. LUÍS T. A. N. BRANDÃO (STRATIVIA), Rene Peralta (NIST). *NIST First Call for Multi-Party Threshold Schemes*. 2023. Tech. rep. Available from DOI: https://doi.org/10.6028/NIST.IR.8214C.ipd.

15. SHOUP, Victor. Practical threshold signatures. In: *Advances in Cryptology—EUROCRYPT 2000: International Conference on the Theory and Application of Cryptographic Techniques Bruges, Belgium, May 14–18, 2000 Proceedings 19*. Springer, 2000, pp. 207–220.

16. IVAN DAMGARD, Maciej Koprowski. *Practical Threshold RSA Signatures Without a Trusted Dealer* [https://iacr.org/]. 2020. Available also from: https://iacr.org/archive/eurocrypt2001/20450151.pdf.

17. CANETTI, Ran; GENNARO, Rosario; GOLDFEDER, Steven; MAKRIYANNIS, Nikolaos; PELED, Udi. *UC Non-Interactive, Proactive, Threshold ECDSA with Identifiable Aborts* [Cryptology ePrint Archive, Paper 2021/060]. 2021. Available from DOI: 10.1145/3372297.3423367. https://eprint.iacr.org/2021/060.

18. DAS, Sourav; REN, Ling. *Adaptively Secure BLS Threshold Signatures from DDH and co-CDH* [Cryptology ePrint Archive, Paper 2023/1553]. 2023. Available also from: https://eprint.iacr.org/2023/1553. https://eprint.iacr.org/2023/1553.

19. HEPP, Thomas; SCHOENHALS, Alexander; GONDEK, Christopher; GIPP, Bela. *it - Information Technology*. 2018, vol. 60, no. 5-6, pp. 273–281. Available from DOI: doi:10.1515/itit-2018-0020.

20. TAKURA, A.; ONO, S.; NAITO, S. A secure and trusted time stamping authority. In: *1999 Internet Workshop. IWS99. (Cat. No.99EX385)*. 1999, pp. 88–93. Available from DOI: 10.1109/IWS.1999.810997.

21. ZHANG, Rui; XUE, Rui; LIU, Ling. Security and Privacy on Blockchain. *ACM Comput. Surv.* 2019, vol. 52, no. 3. ISSN 0360-0300. Available from DOI: 10.1145/3316481.

22. *Medusa: Blockchain Powered LogStorageSystem* [online]. Hao Wang, Desheng Yang, Nian Duan, Yang Guo, Lu Zhang [visited on 2023-06-17]. Available from: https://arxiv.org/ftp/arxiv/papers/2002/2002.03588.pdf.

23. FAIRWEATHER, Elliot; WITTNER, Rudolf; CHAPMAN, Martin; HOLUB, Petr; CURCIN, Vasa. Non-repudiable Provenance for Clinical Decision Support Systems: A Non-repudiation Architecture for Decision Support. In: GLAVIC, Boris; BRAGAN-

HOLO, Vanessa; KOOP, David (eds.). *Provenance and Annotation of Data and Processes*. Cham: Springer International Publishing, 2021, pp. 165–182. ISBN 978-3-030-80960-7. Available also from: https://link.springer.com/chapter/10.1007/978-3-030-80960-7_10.

24. *Multi-signature* [online]. Wikipedia [visited on 2023-12-08]. Available from: https://en.bitcoin.it/wiki/Multi-signature.

25. GENNARO, Rosario; GOLDFEDER, Steven. *One Round Threshold ECDSA with Identifiable Abort* [Cryptology ePrint Archive, Paper 2020/540]. 2020. Available also from: https://eprint.iacr.org/2020/540. https://eprint.iacr.org/2020/540.

26. *TSSHOCK*. https://www.verichains.io/. Available also from: https://www.verichains.io/tsshock/.

27. GAVENDA, Jiří. *Threshold ECDSA Performance Analysis*. 2021. Available also from: https://is.muni.cz/th/fuwu0/thesis.pdf. Bachelor's Thesis.

28. PLAUSKA, Ignas; LIUTKEVIČIUS, Agnius; JANAVIČIŪTĖ, Audronė. Performance Evaluation of C/C++, MicroPython, Rust and TinyGo Programming Languages on ESP32 Microcontroller. *Electronics*. 2023, vol. 12, no. 1. ISSN 2079-9292. Available from DOI: 10.3390/electronics12010143.

29. KEVIN ANDRIAN SANTOSO, Oey; KWEE, Catherine; CHUA, William; NABIILAH, Ghinaa Zain; ROJALI. Rust's Memory Safety Model: An Evaluation of Its Effectiveness in Preventing Common Vulnerabilities. *Procedia Computer Science*. 2023, vol. 227, pp. 119–127. ISSN 1877-0509. Available from DOI: https://doi.org/10.1016/j.procs.2023.10.509. 8th International Conference on Computer Science and Computational Intelligence (ICCSCI 2023).

30. *Fearless Concurrency*. doc.rust-lang.org. Available also from: https://doc.rust-lang.org/book/ch16-00-concurrency.html.

31. KOPPL, Martin; SIROSHTAN, Dmytro; ORGON, Milos; POCAROVSKY, Stefan; BOHACIK, Antonin; KUCHAR, Karel; HOLASOVA, Eva. Performance Comparison of ECDH and ECDSA. In: *2021 2nd International Conference on Electronics, Communications and Information Technology (CECIT)*. 2021, pp. 825–829. Available from DOI: 10.1109/CECIT53797.2021.00149.

32. HUANG, Jiansen; LI, Hui; ZHANG, Jiyang. Blockchain Based Log System. In: *2018 IEEE International Conference on Big Data (Big Data)*. 2018, pp. 3033–3038. Available from DOI: 10.1109/BigData.2018.8622204.

33. *What Is Block Time? What It Measures, Verification, and Example* [online]. investopedia.com/ [visited on 2023-12-11]. Available from: https://www.investopedia.com/terms/b/block-time-cryptocurrency.asp.

34. *Transactions Per Second (TPS) Meaning* [online]. ledger.com [visited on 2023-12-10]. Available from: https://www.ledger.com/academy/glossary/transactions-per-second-tps.