Abstract

Trust on the internet is established through the use of public key cryptography (asymmetric cryptography), which is used for encryption and signing. Digital signatures work by having entities communicating with each other sign messages using their secret private keys and then having them verify signatures on messages with the public key of the other. This ensures three important aspects of trustworthy communication - authenticity, integrity and non-repudiation. This works well when entities already know each other and their respective private keys but a new user connecting to the internet does not have that information. Certificate Authorities (CAs) are well-known and trusted entities that map identities like name or email address to a public keys by issuing certificates. CAs are integral to the correct functioning of any modern network infrastructure and serve as the foundation for trust and security on the internet because their root certificates have been hardcoded into the different browser implementations. These central points of trust are often targets for malicious entities, which either try to steal private keys or try to manipulate the validation process.

This thesis analysis different attack vectors on CAs and proposes solutions to harden the certificate issuance process. Furthermore, by analyzing previous work done at the Chair for Network Architectures and identifying the most essential features contained therein this thesis creates a lightweight, tamper-resistant certification authority, The focus of this thesis is to have no single point of failure by combining a certificate control system (how certificate requests are orchestrated) and a distributed signing service (how certificates are validated) into one lightweight mechanism. This thesis contributes a robust design with the two main building blocks being a byzantine fault-tolerant state machine replication algorithm and a threshold signing scheme. Additionally, a working Go implementation in combination with a qualitative and quantitative performance evaluation of the certification system has been provided.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The internet connects but not all connections are trustworthy. Trust is established through the use of public-key cryptography (asymmetric cryptography), specifically digital signatures, whereby entities communicating with each other sign messages using their secret private keys and then verify signatures on messages with the public key of the other. This ensures three important aspects of trustworthy communication - authenticity, integrity and non-repudiation. This works well when entities already know each other and their respective private keys but a new user connecting to the internet does not have that information. This is where a public key infrastructure (PKI) is needed.

There exist many different types of PKIs like PGP web-of-trust, decentralized PKIs or certificate authorities (CA). CAs are central entities that everyone using the internet knows of since their root certificate has been hardcoded into the different browser implementations. They are integral to the correct functioning of any modern network infrastructure because they serve as the foundation for trust and security by mapping identities like name or email address to a public-keys. These CAs underpin the trust of the internet by issuing certificates but since they are this central point of trust they are a target for malicious entities. This thesis analysis different attack vectors on CAs and proposes solutions to harden the certificate issuance process.

The focus of this thesis will be to eliminating this single point of failure by adding a certificate control system (how certificate requests are orchestrated/recorded) and a distributed signing service (how certificates are validated/verified) into one combined lightweight mechanism. This shall be achieved by analyzing previous work and identifying the most essential features contained therein. Research will be directed towards finding existing projects that can act as a basis of a new lightweight system, that fulfills the identified features in a more efficient manner. A key goal will be to reduce complexity and attack surface of the system.

1.1 Research Questions

Q1 Security What attack vectors exist and how do current implementations address these?

Q2 Previous Work What requirements and aspects of previous implementations are useful?

Q3 Performance What BFT protocols and other projects exist to build a more lightweight system that combines authorization and signing?

Q4 Compatability Can the system be embedded into the existing public key infrastructure?

Q5 Scalability Does the system scale and still stay performant?

1.2 Outline

This thesis has the following structure. In Chapter 2, we introduce background information that defines some terminology to better understand the main components, namely consensus algorithms and threshold signing schemes. This is also to show that this problem has been researched since the 1980s and gives a good historical perspective. Next, Chapter 3 defines parameters of the problem, introduces abstractions and analyses previous work in order to define specific requirements the certification system must fulfill. This chapter's main function is to introduce the key ingredients and justifying their use. In Chapter 4, all the components are combined into the design of the certification system. An overview and detailed description of all parts is given. How the working system was implemented is shown in Chapter 5. We introduce concrete technologies used and showcase some interesting aspects of the implementation. The following Chapter 6 evaluates if and how the requirements were met. This chapter also includes some performance measurements to provide evidence of the system's qualities. In order to give some context to this thesis, Chapter 7 picks some real-world projects that try to achieve similar goals of this thesis. Finally, the conclusion in Chapter 8 summarizes the findings and makes suggestions about how to improve on this work in the future.

Chapter 2

Background

2.1 Consensus Algorithms

Mostly known under the name consensus algorithms or the more modern Distributed Ledger Technology (DLT) the exact technical term for the type of problem these algorithms are trying to solve is *byzantine fault-tolerant state machine replication (BFT-SMR)*. The word *byzantine* originates from the 1982 paper *The Byzantine Generals Problem*[1], in which the author Leslie Lamport describes how a distributed system can come to a consensus even if large parts of it experiences faults. As nodes in a distributed system control resources like compute and storage so do generals on the battlefield, controlling resources like soldiers and firepower. These resources have to be coordinated somehow to achieve a certain goal, which in the context of generals could be the capture of a besieged city. In the paper, Lamport describes this problem as a group of generals surrounding a city in different locations and only communicating by messenger, where the decision has to be made whether to attack or not. The caveat being that one or more malicious generals could be giving conflicting messages by telling one general to attack and the other not to. The goal is to find an algorithm that can ensure that all honest generals will reach a common agreement on whether to attack or not.

The paper shows that the consensus is possible if less than a third of the generals are malicious, which means a single malicious general makes consensus impossible if there are a total of three generals in the group. This conceptually simple problem has been studied for almost 50 years, which shows that in practice it is quite difficult. Over the years there have been many proposed solutions to this problem like PBFT[2], Paxos[3], or Nakamoto Consensus[4] but all have subtle trade-offs and drawbacks. The following subsections will describe some of the details that are needed to understand the trade-offs that go into designing consensus algorithms. Section 2.1.3 will also define the what and why of *state machine replication* so that the reader can understand what the term *byzantine fault-tolerant state machine replication* means.

2.1.1 Adversary/Fault Types

In order to define attack vectors, we have to introduce some terminology that describes the different kinds of faults. A fault describes how an adversary can disrupt a process or node in the certification system.

Passive: A corrupted node follows the protocol but the adversary can see all messages received. It can learn information from the viewpoint of the corrupted node.

Crash: A corrupted node crashes and no longer sends or receives messages

Omission: Once corrupted the adversary decides for every send or received message whether to drop or forward it.

Byzantine: The adversary has full control over the node and can construct, send or not send any type of message.

In this thesis, we assume a byzantine adversary. Going down the list every type of corruption is a subset of the above therefore it is a good account of a byzantine adversaries capabilities. Another important factor when defining the adversary is the visibility of messages and the internal state of non-corrupted nodes.

Full information: The adversary can see the full internal state of any node and also read all messages between them.

Private Channels: The adversary cannot see the internal state or content of messages of honest nodes. It can see that messages are sent and can also choose to delay them depending on the communication model.

Another important factor is the number of nodes an adversary can control. Generally f denotes the number of malicious nodes in a network of n nodes. What threshold of the nodes are corrupted by the adversary.

$f < n$: at least one honest node

$f < 1/2n$: only minority corruption

$f < 1/3n$: theoretical upper bound of corrupted nodes in order to reach consensus

2.1.2 The Communication Model

The communication model specifies how messages over a communication medium are delivered and propagated. Message delivery delays are often caused by exogenous effects like overloaded networks or, more relevantly, by the adversary itself. Therefore the communication model also defines the capabilities and powers of the adversary.

Synchronous model: There exists a known finite time bound Δ on message delay, which means that every message sent at time T will be delivered by time $T + \Delta$.

Finding the correct Δ poses the problem in this model because setting a too large one degrades performance and a too small one could suffer from safety violations. The real world never works as predicted and relying on completely synchronous communication makes the system fragile and more easily exploitable by an adversary.

Asynchronous model: Message delay is finite but unknown. There is no bound on the time to deliver a message but every message has to be delivered at some point.

The asynchronous model assumes nothing about the network delays and protocols designed for this model are also the most robust (in regards to communication) because message delays cannot cause unexpected safety violations. Furthermore, asynchronous protocols adapt to the actual latency of the network because they don't use fixed timeout values. You could say that they run optimally in the long run because they make progress at the same rate as the overall network allows.

Asynchrony might sound like the best choice for any robust and secure distributed system but there exist serious drawbacks that manifest themselves in more complexity of the protocol and less robustness when looking at reaching consensus[5].

Partially synchronous model: This represents a mix of the two former models in which there exists a known finite time bound Δ and a special event called GST (Global Stabilization Time). GST can be thought of as an unknown but specific time at which the network behaves synchronously again (think of short-term network delays like routers having full buffers, etc).

Putting both things together results in any message sent at time T to be delivered by time $max(T, GST) + \Delta$. Intuitively this means that the system behaves asynchronously until GST and synchronously after GST.

2.1.3 State Machine Replication

State machines are one of the most fundamental mathematical models for computation. They are comprised of *States, Inputs, Outputs* and functions that take the input and the state and either compute the new state (*transition function*) or the output (*output* *function*). Any kind of computation can be modeled as a state machine but they are an especially helpful method to for implementing *fault-tolerant* distributed systems[6]. State machines deterministically come to the same output for a given input. This means that copying a state machine across independent nodes and then feeding those the same input leads to all state machines *transitioning* to the same *state* and produce the same *output*. Concretely this means concurrently arriving client requests are interpreted as *inputs* to a state machine. These then need to be ordered in a way that all nodes execute the requests in the same way because without this requirement each node could end up in different states. These requests are then executed and the client receives the *output*. Below are some properties that have to be fulfilled:

Log replication: The client requests have to be ordered into an ordered log across all nodes.

Safety: Any two honest nodes store the same sequence of requests in their logs.

Liveness: Honest nodes will eventually execute a request proposed by a client.

Byzantine fault-tolerant means we need processes that can defend against a byzantine adversary. *State machines* provide a framework to define the function of a distributed service or system. *Replication* is a method for replicating logs or requests across different nodes. This is where consensus algorithms like the ones mentioned in Section 2.1 are needed. In summary: A *byzantine fault-tolerant state machine replication algorithm* provides the theoretical framework to design resilient distributed services.

2.2 Threshold Signature Schemes

The cryptographic primitive underlying any digital signature scheme is asymmetric or public-key cryptography. This will not be a technical introduction about how signature schemes work mathematically but rather it should give an intuitive understanding of why they are important and what makes them so useful. In general, these schemes first generate a large secret number called the private key, which is then used in conjunction with a cryptographic algorithm to compute a public key. This public key can then be shared with anyone and be associated with one's identity. To digitally sign something then means taking the document, private key, and a function that takes both as arguments to compute a digital signature. In order to verify the signature on a document, one takes the document, signature, public key, and a function that takes all of those as parameters, which either outputs true or false. Digital signatures provide three key properties:

- **Authenticity:** A valid signature tells the verifier that the signature could only have come from the entity that holds the private key. It can not be forged.
- **Integrity:** The piece of data that is signed cannot have been modified without detection.
- **Non-repudiation:** The signer cannot deny that it has signed a piece of data.

In threshold signature schemes this private key is distributed among different entities, whereby a subset of them have to collaborate to compute a valid signature. The private key isn't copied but rather split up into different shares. An example would be four shares where three shares can be combined to form a valid signature. This would be called a 3-out-of-4 signature scheme. The main benefit threshold signatures provide is that a verifying party only has to store one public key and doesn't have to know how many parties collaborated to make this signature. Essentially what these schemes provide is a multi-party authorization, which is beneficial in the context of building a tamper-resistant distributed system.

Chapter 3

Analysis

In this chapter, we define the problem that the current implementations of certificate authorities have. We want to analyze how the CAs can be attacked and how to mitigate those attack vectors. In Section 3.1.1 we outline how a Certificate Authority fits into the overall network infrastructure, why they exist and how they function. Next in Section 3.1.3, we analyze two generalized attack scenarios, how previous work has addressed these and what requirements can be gleaned from them. In Section 3.2, all the requirements are bundled and categorized in order to have a good overview of what will be achieved. Finally, in Section 3.4 and Section 3.5 we select protocols and algorithms to build a lightweight and tamper-resistant certification authority.

3.1 Problem Statement and Scope

3.1.1 The Importance of the Certificate Authority

Certificate authorities (CA) are integral to the correct functioning of any modern network infrastructure because they serve as the foundation for security by mapping identities like name or email address to public keys. Certificates and the issuing CA provide a layer of trust that everyone relies upon but this, in turn, makes them an obvious target for attacks. An example of a successful attack and its consequences can be observed in the DigiNotar hack of 2011[7], where hundreds of false certificates were issued for Google, Microsoft and even for some top-level domains like *.com and *org. There is evidence that this led to some Man-in-the-Middle attacks and because of this devastating loss in trust, the CA DigiNotar went bankrupt. Therefore it is paramount to understand the way CAs issue certificates and how this process provides ample attack surface for malicious actors

The certificate issuance process generally involves three entities as seen in Fig. 3.1: the client, the certificate authority and a registration authority. A client wishing to certify its identity by binding a name to some public key makes a certificate signing request (CSR) to the certificate authority. At the same time, it sends some kind of proof of the aforementioned identity to a registration authority (RA). This RA checks and authorizes the request on behalf of the CA, which then signs the certificate and returns it to the client.

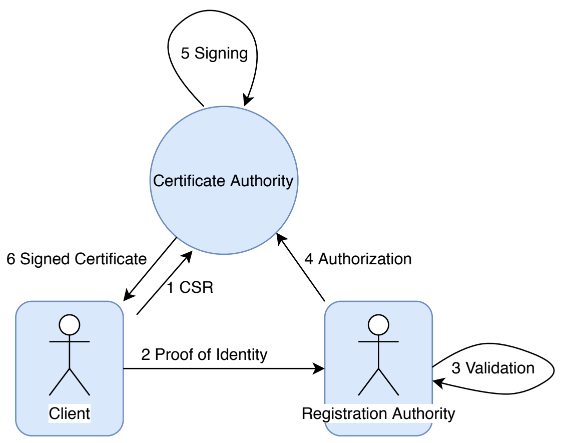

Figure 3.1: Normal certification process

In order to better analyze the certification process, we must recognize some key abstractions that derive from the functions of the client, CA and RA.

Client: The entity that wants to get a certificate.

Signer: The entity that signs a certificate and issues it to the client.

Validator: The entity that knows what makes a CSR valid (no duplicates or spoofing for example).

3.1.2 Use cases

This thesis proposes a general answer to the problem detailed above. Having a tamper-resistant certification process could be helpful to existing Certificate Authorities like DigiCert. It would not only provide their customers with a greater sense of security but also various browser vendors would be more willing to trust certificates signed by CA equipped with a hardened certificate issuance process. It would be similar to a bank upgrading its security system and with that attracting more high-net-worth individuals to store wealth in their vaults.

Another use case would be any company or project which wishes to construct its own secure network infrastructure. An example of this kind of project is called *VITAF* (**V**ertrauenswürdige **IT** für **A**utonomes **F**ahren), which aims to increase the trustworthiness of autonomous driving network infrastructure. In such a setting vehicles are in constant communication with different entities, for example, vehicle-to-vehicle or vehicle-to-infrastructure (traffic-lights/backend). This vehicle-to-X (V2X) communication should happen in a trustworthy environment which means entities need identities and public keys that are made trustworthy by a certificate authority. Since vehicles operate in the real world and can cause real damage, the network infrastructure coordinating them should be as secure and resilient towards attacks as possible. A tamper-resistant certification authority should be an integral part of that.

3.1.3 Attacks

In Fig. 3.2 it can be seen that an attacker can compromise the CA by attacking the validator. An attacker can then authorize any CSR and trick the CA into signing the certificate.

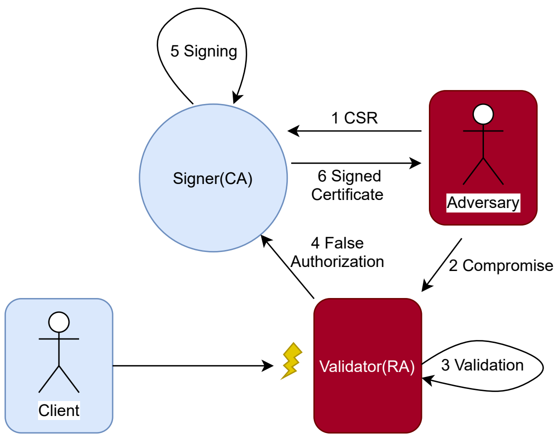

Figure 3.2: Compromising the RA

The second way is more serious because here the CA is compromised as seen in Fig. 3.3. When this happens the attacker most likely gets access to the private keys (for signing) and can issue to itself any kind of certificate with any identity. Even a simple Denial-of-Service attack on the CA could severely hamper the CA's ability to issues new certificates to clients. If old certificates can't be renewed this would also break many parts of the network infrastructure.

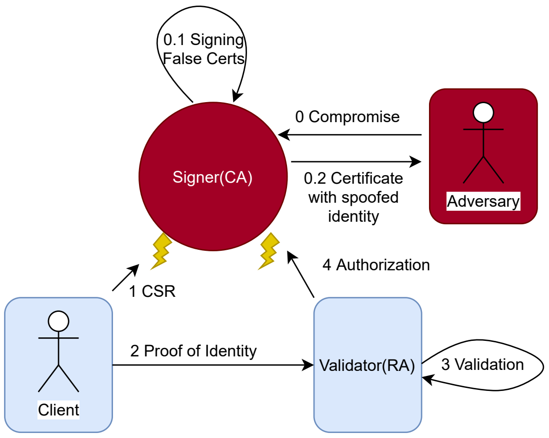

FIGURE 3.3: Gaining access to the CA

From these two attack scenarios, we can therefore distill two key attributes for the certificate issuance process.

1. CSRs need to be authorized in a tamper-resistant way
2. Authorized CSRs need to be signed in a tamper-resistant manner

These two functions the system follow directly from the attack scenarios defined above. In the next section, we will try to define more exactly the requirements a tamper-resistant certificate issuance process has to have.

3.1.4 MITIGATING ATTACKS/ANALYSIS OF PREVIOUS WORK

This thesis is a refinement of previous work that has been done at the Chair. We, therefore, want to look at two theses that have tackled aspects of the problem in order to ascertain some helpful features that could be used in this thesis.

In *Verifiable secret sharing and threshold signatures for tamper-resistant signature services* [8] Filip Rezabek proposes threshold signing service based on RSA, which eliminates the problem of private key compromisability. By private key compromisability we mean the vulnerability of a private key (secret) being stolen from a given node's key store. This is the exact scenario displayed in Fig. 3.3. The thesis achieves this by first having a dealer generate publicly verifiable secret shares and then having a group of nodes collaboratively sign pieces of data using those secret shares. In this manner, a service that produces signatures becomes hardened because now a threshold of servers would have to be compromised in order to steal the private key (secret).

Some key requirements gleaned from this thesis are:

1. **No single point of failure:** Nodes should be distributed over multiple locations and servers. This also means that the system as a whole should continue functioning in case any of the nodes go offline or starts behaving in a byzantine manner.
2. **Node Accountability:** Each node should be able to identify if another node is behaving maliciously. Specifically, it should be able to identify if a node's signature share is correct so that it doesn't compute a full signature with a corrupted share. Ideally, it should also be able to isolate the offending node in the network.
3. **Autonomy:** Nodes should be able to act autonomously. Every piece of data that has to be signed should be identified correctly so the node can decide whether to sign it or not. These rules should be set somewhere as policies for the whole system.
4. **X.509 Compatability:** Any certificate or piece of data should comply with the X.509 standard as far as possible. This ensures compatibility with the rest of the internet's Public Key Infrastructure (PKI) and also assures that the cryptographic primitives used are widely tested and secure.

With this, we have mitigated the attack scenario from Fig. 3.3, which means that if we distribute the signer into many partial signers an attacker can't easily gain control to the private key and give itself any kind of certificate. This setup does not mitigate the attack scenario in Fig. 3.2 because Certificate Signing Requests (CSRs) still have to be validated and sending false authorizations to the partial signers is therefore still possible. Validation always requires state on which the validation logic operates. For example, if a given identity has already been issued, whether a certificate has been revoked or who is allowed to request certificates. To address this we need some way to make the validation and subsequent authorization tamper-resistant by making it a multi-party authorization. Similar to having a threshold of partial signers for a full signature the idea is to have a threshold of validations for full authorization.

In *Tamper-resistant creation of integrity tokens for trustworthy communication in cyber-physical systems* [9] Cristoph Rudolf utilizes a distributed ledger platform name Hyperledger Fabric to coordinate the state of a group of validators. A group of signers can then use this distributed ledger containing the authorization of the validators to sign integrity tokens (pieces of data). This multi-party authorization makes tampering with the authorization process more difficult and therefore hardens any kind of service that relies on validation.

Some key requirements gleaned from this thesis are:

1. **Multi-party authorization:** Tamper-resistance relies heavily on this authorization process because it increases the number of parties needed to compromise. This requirement is also a mix of the above **No single point of failure** and **Autonomy**. The distribution of validation/authorization coupled with the node autonomy provide the foundation of the resilience of the system.
2. **Permissioned system:** This system does not function in an open manner such that anyone can participate. Participation in the certification process is restricted to a group of known identities and public keys.

The solution that Cristoph Rudolf's thesis proposes involves the Hyperledger Fabric framework [10], which is a Distributed Ledger Technology (DLT) library maintained by IBM and the Linux Foundation. The Hyperledger Fabric(HLF) framework is a modular and quite extensive consensus algorithm built for all kinds of enterprise applications that also supports features such as smart contracts. The problem is the computational and bandwidth heaviness coupled with the complex code base and configuration of the system. Under the hood, HLF uses many different components in its tech stack like Kafka, Zookeeper, CouchDB and Docker that increase the complexity and vulnerability immensely. HLF's main component is written in Go, Kafka is written in Scala (runs on JVM), Zookeeper in Java and CouchDB in Erlang. To illustrate the point a bit more these are all the ports that have to be opened on one HLF node: 7050, 2181, 2888, 3888, 9092, 9000, 7054, 7051, 7053, 8053, 2377, 7946, 4789. All of these different components make installation and configuration not only difficult but require continuous maintenance and extensive know-how.

The KISS Principle, which stands for Keep It Simple Stupid was coined by a Lockheed Skunk Works (built military aircraft and other complex machinery) engineer named Kelly Johnson. Systems designed with this principle in mind tend to be simpler in their construction so that average engineers can maintain and repair them. HLF programmers use a lot of cutting-edge components in their design but fail to realize that the fusion of these components into one makes the system unmanageable. Without manageability, the average systems engineer does not possess the knowledge to deploy HLF into a production environment.

We have looked at two different systems that tackle aspects of the problem we are trying to solve and listed some of the requirements. We've also examined why those proposed solutions are inadequate alone or how their implementations tend to be too complex. In the next section, we will aggregate some of the requirements and lessons learned to have a clear picture of how to design a system that works in a more tamper-resistant and lightweight manner.

3.2 Requirements

This section describes the guiding design philosophy of the system. Now we will have a look at two high-level requirements for the certification authority that will not only be used to create the system but more importantly be used to analyze the trade-offs that have to be made.

Lightweight: In the context of this thesis lightweight defines the impact the certification process has on its surroundings e.g. the server it runs on and the network it uses. Concretely this can be measured in the memory and CPU usage, and the message complexity of the protocol. Another aspect of this designation is the ease of maintennance and configuration of the system. The goal is to reduce the complexity of configuring nodes and maintaining the codebase.

- **Simplicity:** Reduction of complexity by reducing number of dependencies and components.
- **Maintainability:** Code provides easy-to-understand interfaces and binaries can be deployed to a cluster of nodes with ease.
- **Performance:** CPU usage and network use is minimized.

Tamper-resistance: Tampering always implies someone is doing something malicious but in this context, it is also supposed to mean fault-tolerant. Notice that this does not mean tamper- or fault-proof; only that we are trying to reduce the attack vectors or at least make them more difficult to realize. Below you will see further refinements of this requirement.

- **No single point of failure:** There exists no single server which can severely disrupt the certification process.
- **Autonomy:** Nodes autonomously decides what to do with every input they receive. It is difficult to spoof or trick a node into making a wrong decision because it controls how state is changed.
- **Security/Safety:** No invalid certificates are issued.
- **Liveness:** The certification process will eventually issue certificates to all valid certificate signing requests it has received.
- **Multi-party authorization:** A certificate is only issued when a threshold of nodes can independently validate a Certificate Signing Request.

- **X.509 compliant:** The X.509 standard defines the format for certificates and other cryptographic tools used for secure communication on the internet. Using well-known, standardized and tested cryptography is very important to making this system tamper-resistant.

The structure of the requirements perfectly reduces to the title of this thesis: *A Lightweight and Tamper-Resistant Certification Authority.* The title is the root of a tree with two branches, which in turn have children nodes that further specify what is meant. We have defined **lightweight** through the notions of simplicity, maintainability and performance. **Tamper-resistance** defines itself through distributing decision making and authorization by using standardized cryptography. With these requirements in mind, we will now model the certification process, analyze exact points of failure and see how to mitigate those.

3.3 Approaching the solution

We have introduced the abstractions of **signer**, **validator** and **client** but now it is helpful to zoom out a bit more and abstract the certification as state machine as seen in Table 3.1. We've introduced the basic concept of state machines in the Background chapter and will now apply it to the problem of tamper-resistant certificate issuance. The goal is to construct the theoretical architecture of the system and in doing so identifying the basic building blocks.

The two inputs are provided by a **client** the state and transition functions are partitoned across a **validator** and **signer**. The **signer** produces the signed certificate as the output to this theoretical state machine. The transition function is composed of many distinct sub-functions that manipulate the state in order to deterministically drive the state machine to either issuing or not issuing a certificate. The highlighted parts are state and transition functions that have to be set by the certificate authority as some sort of policy. This policy defines what makes a valid Proof of Identity and who is allowed to request certificates. The input CSR and output certificate are defined by the X.509 standard.

Table 3.1: Modelling the CA as a state machine

Input	**State**	**Transition function**	**Output**
• CSR • Proof of Identity	• White/Blacklist of long-term identities • Authorized CSRs • Issued certificates • CA private key	1. Validating Proof of Idenitity 2. Authorizing CSR 3. Receiving authorization 4. Signing certificate 5. Returning certificate to client	• Signed Certificate

Table 3.1 shows the functioning of the overall system but in practice, the input, state and transition function is split between the RA and CA as seen below in Fig. 3.4. This partitoning of state and logic opens up the attack scenario described in Fig. 3.2, where the validator is compromised and the transition function of authorizing a CSR is used to spoof the CA into signing an invalid certificate. The delegation of validation is done out of practicality but this leads to a less secure system. The logical next step would be to make the validator and signer one server so that the state and transition functions are consistent can't be spoofed. State has to be thought of as the context for decision-making. If the context/data for deciding is compromised then the server isn't secure. One of the requirements is **autonomy**, which encapsulates that in order to make the system secure, every participant has to be able to make their own decisions and not rely on spoofed data.

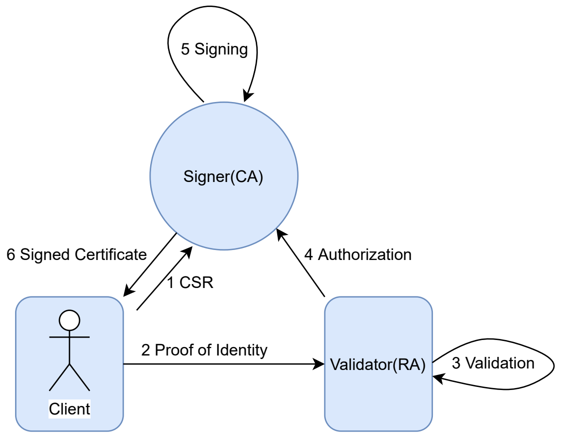

Figure 3.4: Abstracted certification process

Fusing the validator and signer into one server makes the system more secure from the state machine perspective because the state and transition functions are all internal and therefore cannot be spoofed by an outside attacker. The key requirement to make this system tamper-resistant is **no single point of failure** so having on server hosting all the functionalities of the certification process can not be accepted. The key idea behind this system is to distribute all the functions across many servers and perform a **multi-party authorization** as seen in Fig. 3.5.

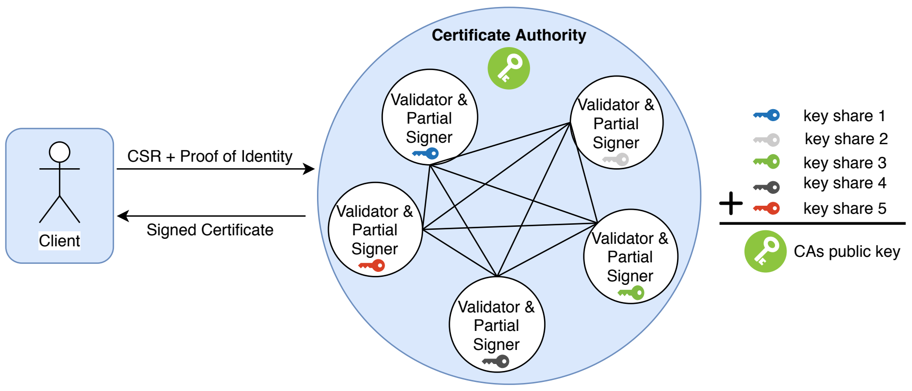

Figure 3.5: Fusing the functions and distributing the system

This system does not change the basic trust model because we still have a central authority that manages the system. All we are doing is adding a layer of resilience to the certification process by distributing the functions of a single server across multiple ones.

The state of the different servers has to be synchronized somehow given that it's operating in a malicious environment so the goal is to create a way to replicate the state across the different servers in the cluster. This is called State Machine Replication (SMR) and this is where we need a byzantine fault-tolerant state machine replication algorithm (aka consensus algorithm) so that every server has the same state through which the validator logic can approve a CSR.

In comparison to the normal certification process seen in Fig. 3.1 we are doing two things: First, signer (CA) and validator (RA) are fused together into one entity. This ensures higher cohesion which makes the process more **tamper-resistant** by making every node more **autonomous** since the transition function is now internal and cannot be spoofed by an outsider. Second, this fused entity is replicated among different nodes so that a **multi-party authorization** is needed to issue a certificate. We think that this fusion of validation and signing and subsequent replication and distribution makes the whole system more tamper-resistant and at the same time more lightweight.

3.4 Selecting the right consensus algorithm

We've introduced byzantine fault-tolerant state machine replication (BFT-SMR) aka consensus in the background chapter and looked at one of the oldest implementations called Paxos[3]. In this section, we want to decide what kind of consensus algorithm to use so that it fits the requirements. There are three algorithms that are interesting to analyze: PBFT, HoneyBadgerBFT and HotStuff. First, we will give a short introduction and description of the different algorithms and then argue why we selected HotStuff. Below is a quick comparison of the different algorithms.

Table 3.2: Comparing consensus algorithms

	Communication Model	**Message Complexity**	**Latency**	**Actively maintained repo**
PBFT [2]	Partial synchrony	$O(n^2)$	2 rounds	No
HoneyBadger [11]	Asynchrony	$O(n)$	3 rounds	No
HotStuff [12]	Partial synchrony	$O(n)$	3 rounds	Yes

There exist many nuances in the trade-offs that go into designing consensus algorithms. Trying to compare them and finding the best one is not a straightforward task but nevertheless, this thesis has come up with some metrics that offer a way to compare them. As described in the background chapter (reference) the **communication model** describes what assumptions are needed for message propagation in a network. The **message complexity** measures the overall amount of messages that have to be exchanged between nodes to reach consensus. **Latency** measures the number of round trips it takes to commit a command. The last column tells us if there exists an **actively maintained repository** on GitHub in a common programming language.

HotStuff is a leader-based byzantine fault-tolerant state machine replication protocol that works in the partially synchronous network model. It assumes a reliable and secure peer-to-peer network but unlike HoneyBadgerBFT it relies on partial synchronicity, which means that there is an upper bound on the message transmission delay. One key improvement of HotStuff over other BFT protocols is the change from mesh communication to star communication, which means all messages are sent to the leader, who then forwards them to other nodes. This drastically reduces message complexity. HotStuff is capable of driving the protocol to consensus at the pace of actual network delay, which is a property called responsiveness. This enables the throughput of the system to dynamically match that of the network.

Clients propose requests to any of the nodes and wait for the group of replicated state machines to process those requests. The protocol progresses in a succession of *views*, where each has a dedicated leader known to all nodes. The underlying consensus data structure looks like a blockchain since every node stores a tree of pending requests batched into blocks that reference a parent block. During the protocol, a monotonically growing branch of this tree of blocks becomes committed by having the leader of a view collect votes for a block from the nodes in three phases.

Ultimately we have decided to go with HotStuff because it has an actively maintained GitHub repository that can be leveraged when implementing the system. Although HoneyBadger is the more robust protocol it is the first asynchronous BFT-SMR protocol and therefore not very proven. Additionally, the complexity of the protocol is also higher than HotStuff because it has so many subprotocols. In order to meet the requirement of **lightweight**, our implementation should be as **simple** and **maintainable** as possible. So choosing an easier-to-understand algorithm with an actively maintained codebase is right in line with those requirements.

3.5 Selecting the threshold signature scheme

We have introduced the general concept of digital signatures and more specifically the idea of threshold signature schemes. We are employing a threshold signature scheme because we want a group of nodes to collaboratively sign a certificate in order to make it valid under the assumption that not all of the nodes are available or that some might even be malicious. Only needing a threshold of nodes and not all of them to produce a full signature furthers our goal of **tamper-resistance** because it furthers the property of **liveness**. Even if some nodes are unavailable the distributed signature algorithm can produce enough signature shares to compute a full signature, which means that part of the issuance process is non-blocking.

In *Practical Threshold Signatures* [13] Victor Shoup introduces a threshold signature scheme based on RSA that has the following properties:

1. Unforgeable and robust in the random oracle model, assuming the RSA problem is hard.
2. Signature share generation and verification are completely non-interactive.
3. The size of an individual signature share is bounded by a constant times the size of the RSA modulus.

The signature produced through this scheme is standard RSA signature which means that the format of the public key and the verification algorithm are the same as for any RSA signature.

Point 1. of this scheme's properties holds that it is unforgeable and robust in the random oracle model. This coupled with the fact that the paper specifies a formal security proof makes this scheme fit our requirement of **tamper-resistance**.

Another important property that supports **tamper-resistance** is the fact that signature share generation and verification are completely non-interactive. This means that this scheme could work in an asynchronous environment, which is the most difficult environment a distributed system can find itself in. More concretely this means that the **liveness** property is held up. Reducing the message complexity and therefore making the certification system more **lightweight** also results from the non-interactivity of the signature scheme. It simplifies the protocol by **reducing complexity**.

Furthermore, in RFC 3279 [14], which defines the X.509 standard we see under point 2.2 three supported signatures algorithms: RSA, DSA and ECDSA. Since RSA is one of them this threshold signature scheme adheres to the requirement of **X.509 Compliance**.

3.6 Summary

In this chapter we've outlined how a certificate authority operates by abstracting key functions and identifying vulnerabilities. Descriptions of attacks on the certification process and previous work on the chair have been used to distill requirements for designing a **lightweight and tamper-resistant certification authority**. Through the state machine approach, we've seen the need for higher cohesion between the validation and signing function which leads to the fusing of those functions into one entity. The requirement of **no single point of failure** then leads to the conclusion that we need a byzantine fault tolerant state machine replication algorithm and a threshold signing scheme. With these two ideas, we can now imagine a new approach to designing a certification authority.

Chapter 4

Design

4.1 System Overview

The key idea behind making the certification authority more tamper-resistant is having **no single point of failure**. This means we need a group of nodes that each can perform the same functions of validation of a CSR and subsequent signing of a certificate. In order for this group or cluster of nodes to be **tamper-resistant**, they have to hold two properties discussed in the Analysis Chapter: Liveness and termination. To start, we need to understand the four high-level phases of the certification process and what a general outline of the system looks like:

1. Client connects to the cluster and makes a request
2. Certification cluster replicates and validates request
3. Certificate is signed through threshold signing scheme
4. Gateway node returns fully signed certificate to client

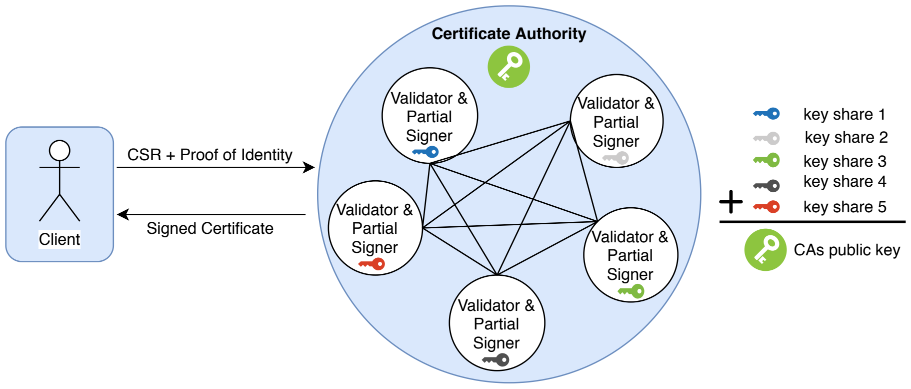

Figure 4.1: High-level overview

In Fig. 4.1, we can see the system overview but it is more instructive to explore the design of the system through the different views different entities have on the system. First we can see the different entities that make up the system:

- Client
- Node = Validator + Signer + Client Server (Gateway)

4.2 Client's View

From the **client's view** it is interacting with a normal CA. It may have multiple addresses where it can contact the CA but from its view, it should be completely oblivious to the way the certificate is generated.

4.2.1 Components

X.509 root certificate: A public-key certificate that identifies the HotCertification cluster as a root Certificate Authority. It is a self-signed certificate that is generated when the cluster is created and stores the public key and other important information. It is stored client-side similar to browser vendors hard coding root certificates into their releases. The client uses these to validate that it has received a valid signature on its certificate.

X.509 CSR: This is the standardized Certificate Signing Request that holds information about the client such as a public key, email address and any kind of extension defined in the X.509 standard. The client generates this every time it wishes to acquire a new identity with a corresponding certificate. **Proof of Identity:** Some sort of proof that the identity the client wishes to certify is held by the client itself. This thesis does not define what a proof of identity looks like but a real-life example would be a passport.

4.2.2 Message Sequence

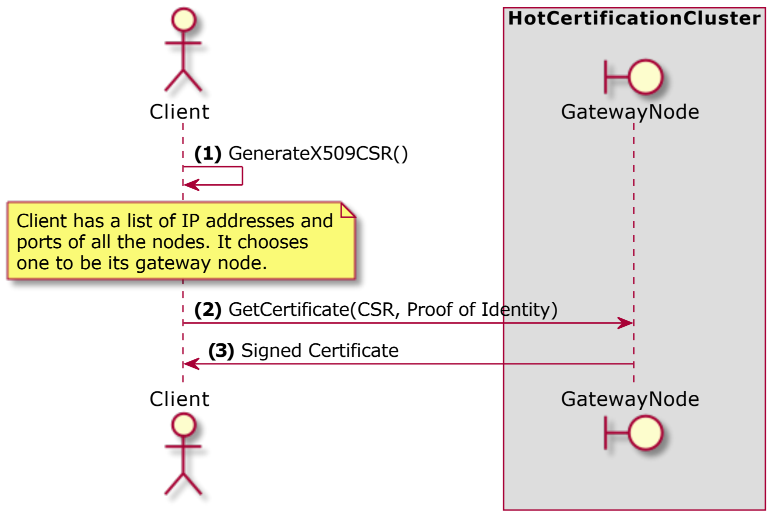
HotCertificationCluster
Client
GatewayNode
(1) GenerateX509CSR()
Client has a list of IP addresses and ports of all the nodes. It chooses one to be its gateway node.
(2) GetCertificate(CSR, Proof of Identity)
(3) Signed Certificate
Client
GatewayNode

Figure 4.2: Client's view of the system

As can be seen in Fig. 4.2 from the client's perspective it looks like interacting with a normal CA with the only difference being that this CA can be connected to through multiple endpoints. The client stores a root certificate and a list of known gateway nodes (every node of the cluster can be a gateway node) in its database. It then generates a CSR with its desired identity and tries to send it to an arbitrary node. If it doesn't get a certificate in a specified timeout it tries to send a request to another node of the cluster. It only has to be able to contact one of the nodes in order to be able to get a fully signed certificate and does not care or know about the complexities of how it is created. This simple abstraction makes integrating the system into existing code bases easy.

4.3 Node's View

The **node's view** possesses more complexity. A node represents one of many nodes that have the same functionality and together make a HotCertification cluster. All nodes are equal, which means there is no special node with more privileges or functionality. This means that it does not matter which node initially receives a CSR because they work together to synchronize their state and produce a valid certificate. So from the node's view, it has to manage two interfaces a public one for clients and a private one for internal communication between the nodes of the cluster.

4.3.1 Components

Each node has three subsystems that run concurrently:

Client Server (Gateway): This is the public enpoint/interface a client connects to in order to get a certificate. It manages connections, parses requests, passes them onto the replication subsystem and returns a fully signed certificate as soon as the certification process is done.

Replication Server: Internal endpoint which is a byzantine fault-tolerant state machine replication algorithm that takes concurrently arriving requests, orders and validates them so that every node has the same state.

Signing Server: Internal endpoint which is a threshold signing scheme that creates valid X.509 certificate signatures.

4.3.2 Message Sequence

Fig. 4.3 shows how the different components to produce a certificate. Fig. 4.3 assumes that the process experiences no faults and has simplified some aspects in order to make it clear how everything works together. The **client server** acts as a gateway to the certification system and orchestrates the certification process for a given client's request.

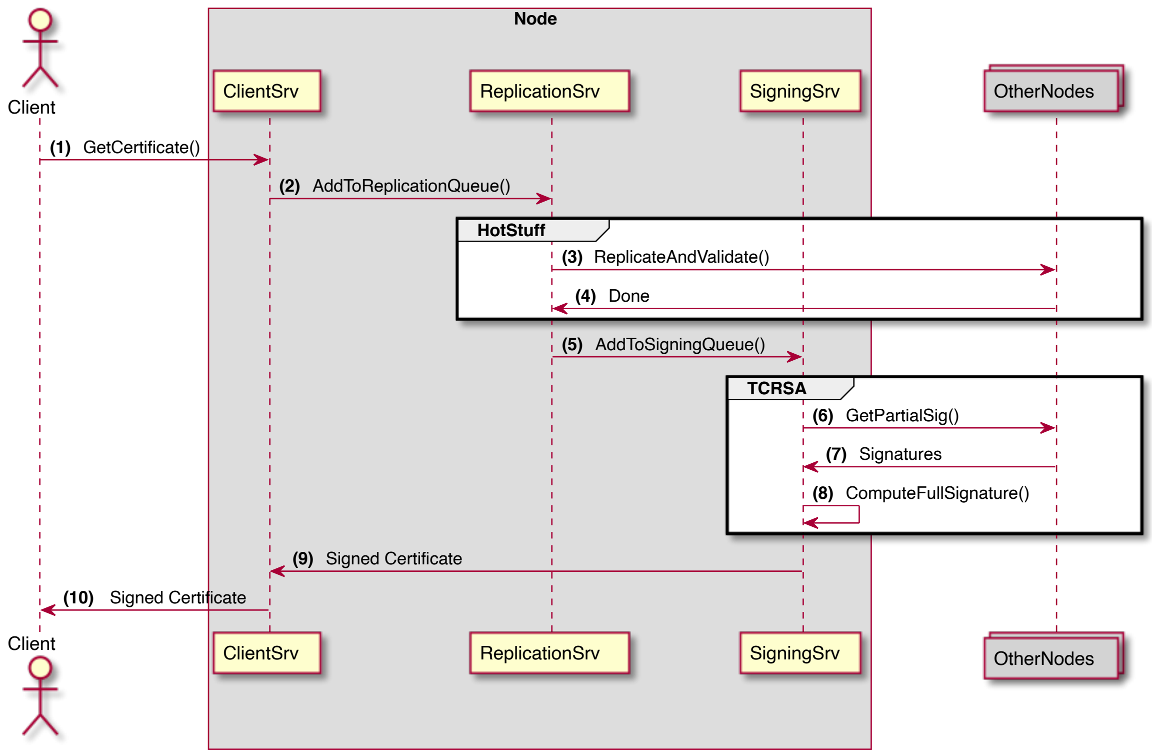

Figure 4.3: Node's view of the system

The **replication server** coordinates a multi-party authorization of a certificate signing request (CSR). Concretely this means that a threshold of nodes has to validate the CSR independently, which means that all the nodes have to get the same state somehow. Since the system works in a possibly hostile environment (byzantine adversary) and we want to make it very **tamper-resistant** we are using the HotStuff replication protocol. One can think of it as a distributed database where every transaction is also validated before being persisted. The fusion of the replication process with the validation process is one of the key insights of this thesis and what makes it **lightweight**.

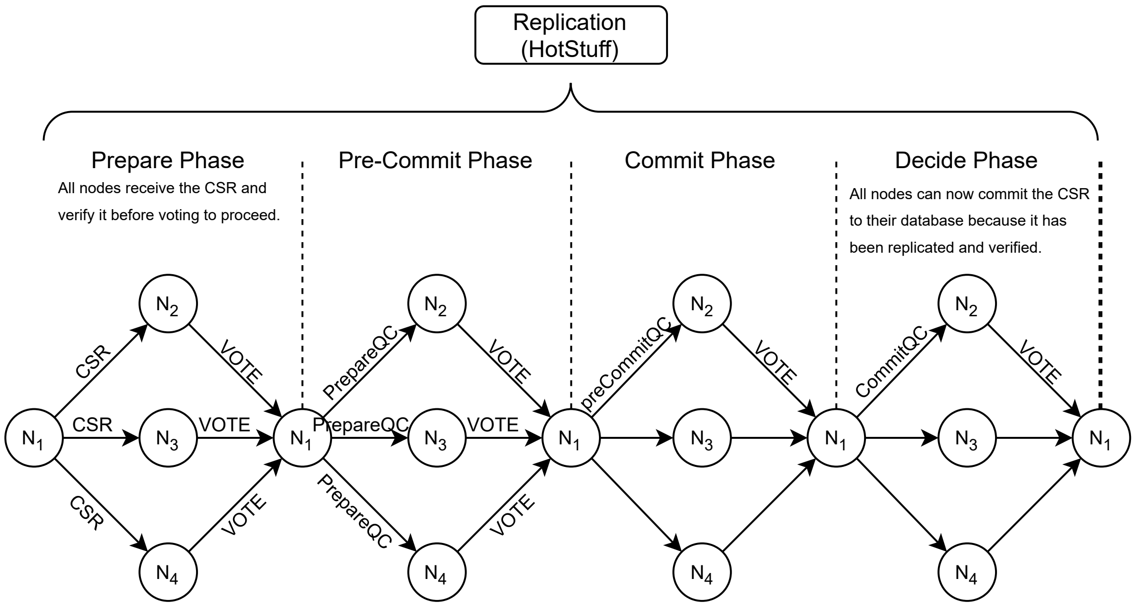
Replication (HotStuff)
Prepare Phase
All nodes receive the CSR and verify it before voting to proceed.
Pre-Commit Phase
Commit Phase
Decide Phase
All nodes can now commit the CSR to their database because it has been replicated and verified.
CSR
VOTE
PrepareQC
preCommitQC
CommitQC
N1
N2
N3
N4

Figure 4.4: Replication and validation message exchange

Fig. 4.4 shows the different phases of the replication protocol and which messages are passed in between the nodes. There are four nodes (minimum amount) in this example setup named N1 to N4, where N1 is the "Node" from Fig. 4.3 and the N2, N3 and N4 are the "OtherNodes" from that figure. PreparePhase is the propagation phase in which every node receives and also validates a CSR but does not commit it to its database. This is done in the DecidePhase, at which stage a node can be sure that enough nodes have validated and also committed a CSR to their database. At this point, the system as a whole is synchronized to the same state and can progress to the next stage, which is signing. The exact functioning of the different replication phases is not so important for the design discussion but what should be understood is that they ensure the **liveness** and **security/safety** of the certification process.

Once the state is synchronized across the different nodes we can perform the **multi-party authorization** on a certificate by means of a threshold signing scheme. We've picked a non-interactive threshold RSA implementation with a trusted dealer setup been customized to work with X.509 certificates. Although we are already validating in the replication stage the final authorization happens through each node's partial signature. As seen in Fig. 4.5 the node responsible for handling a client's CSR initiates the signing protocol, collects partial signatures and computes the full signature for the certificate. This certificate is then returned to the client, completing the certification process.

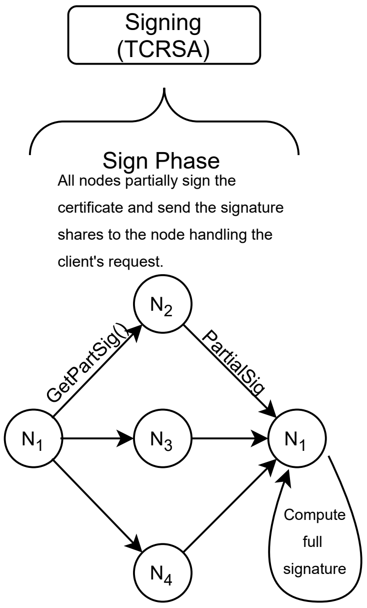
Signing
(TCRSA)
Sign Phase
All nodes partially sign the certificate and send the signature shares to the node handling the client's request.
GetPartSig()
PartialSig
N_1
N_2
N_3
N_4
N_1
Compute full signature

Figure 4.5: Signing phase message exchange

4.4 Summary

In this chapter, we explored the high-level design of the certification system by exploring how the different actors see and interact with it. The client's view showed us that the complexity of the tamper-resistant certification process can be easily hidden, making implementing clients that use this CA easy. Through the node's perspective, we saw that three distinct subsystems work together to produce a certificate. The client server is the public interface to the certification process. The replication server abstracts the HotStuff algorithm for us, which replicates and validates requests. The signing server implements a threshold signature scheme, which aggregates signatures and combines them into a full signature for a certificate. We have left out a lot of detail and subtleties of the system in order to make the design of the system clear. In the next chapter, we will dive into more detail to show exactly how the system was implemented.

Chapter 5

Implementation

The Analysis Chapter showed which problem we are trying to solve and what kind of algorithms we need to tackle it, namely a byzantine fault-tolerant state machine replication algorithm (BFT-SMR) and a threshold signing scheme. The Design Chapter outlined how to construct a tamper-resistant certification process by having **no single point of failure**. It also designated the main building blocks: A replication server and a signing server. The replication server provides the interface to the BFT-SMR, for which we choose to go with the HotStuff algorithm. The HotStuff algorithm has many concrete implementations but the most actively maintained repository[15] is one from the University of Stavanger in Norway by a research group called Resilient Systems Lab (Relab)[16]. The programming language used is Golang (Go), which is an ideal language for building concurrent, distributed and performant systems.

The next building block is the signing server, which provides the interface to the threshold signing scheme. The TCRSA github repository[17] provides only barebone implementation, with just the core functionality. As part of this thesis' implementation, a networking and serialization stack had to be wrapped around the TCRSA implementation. Furthermore, the scheme had to be made to work with the X.509 standard in order to produce valid X.509 certificates. The main tools used are protocol buffers[18]for serialization and gRPC for the general networking stack. gRPC[19] is a transport protocol developed by Google that works over HTTP and enables a way to easily generate API stubs for different programming languages.

This implementation also uses many other libraries, languages and technologies like Docker, Python and shell scripts for deploying the system. For the name of the code repository, we choose **hotcertification** since the core algorithm is HotStuff and the main function is certification. The code can be found on Github under raphasch/hotcertification[20] and is mainly written in Go.

5.1 Project Structure

In this chapter, we will use this directory structure in Fig. 5.1 to showcase how the system was implemented. For now, we can already identify some of the building blocks from the Design chapter, namely in the **replication** and **signing** directories.

```
.
|-- Dockerfile
|-- Makefile
|-- README.md
|-- benchmark
       |-- benchmark
       |-- main.go
       |-- measurements
       |-- plot.py
       |-- plotting_test.ipynb
       |-- scripts
       +-- test-flow.md
|-- cmd
       |-- certificationserver
       |-- client
       +-- keygen
|-- crypto
       |-- crypto_test.go
       |-- serialization
       |-- threshold.go
       +-- threshold_key.go
|-- go.mod
|-- go.sum
|-- hotcertification.go
|-- hotcertification.toml
|-- logging
       +-- logging.go
|-- protocol
       |-- client.pb.go
       |-- client.proto
       +-- client_grpc.pb.go
|-- replication
       +-- replication_server.go
|-- run_servers_localhost.sh
+-- signing
     |-- signing.pb.go
     |-- signing.proto
     |-- signing_gorums.pb.go
     +-- signing_server.go
```

Figure 5.1: Directory structure

5.2 Executables

The implementation consists of three executables that work together to form the hotcertification process. The code for the executables can be found in the **cmd** directory seen in Fig. 5.2 in which has three subfolders containing the main file of the respective executable. All are written in Go and can be compiled with the help of a Makefile by running `make` in the base directory of the repository.

```
cmd
|── certificationserver
      |── certserver
      |── client_server.go
      +── main.go
|── client
      |── client
      +── main.go
+── keygen
    |── keygen
    +── main.go
```

Figure 5.2: Project executables

certserver: This executable bundles together all the functions that have to be performed for the certification process: validating, replicating and signing client requests. It reads a .toml file for configuration and information on other nodes that are part of the distributed certification authority. Crucially, it has to be given two secret keys: an ecdsa key for TLS and HotStuff and a threshold key, which is a serialized custom data structure for the threshold signing scheme that produces the signature for a valid certificate.

client: An example client implementation that creates a X.509 CSR and sends it to one of the servers running the certserver executable.

keygen: This stands for key generation and creates all the keys and TLS certificates necessary for deployment. Here one also has to define the parameters for the threshold signing scheme like the number of nodes and the threshold value. All the keys are put into a directory that has to be passed as a command-line argument.

5.3 Setup

The first step to starting a hotcertification cluster starts with generating keys. This involves defining the parameters of the system such as the number of servers (-n) and the threshold of servers needed to be honest (-t). The threshold signature scheme produces an RSA signature and the key size options define the size in bytes that the key should have. Fig. 5.3 shows an example command.

```
keygen -n 4 -t 3 --key-size 2048 keys
```

Figure 5.3: How to generate cryptographic material

The next step would be using the hotcertification.toml file as a template to configure important information needed by the nodes to connect to each other. Fig. 5.4 shows a configuration for starting four nodes on a local computer. This configuration file defines only public information or where to find it, the private keys have to be passed to the certserver executable at startup. Since this thesis implements a CA we need to describe this CA cryptographically, which means we need a root certificate that tells the world the public key and other information. Keygen generates that certificate as well and it can be found in the directory where the keys are stored as root.crt. Every node has a simple integer as id, a public key and a certificate that wraps that public key so it can be used in TLS. Furthermore, every node exposes three ports: The client port which takes handles requests and two ports that are responsible for the internal functions of replication and signing.

```
# For TLS
root-ca = "keys/root.crt"

# HotStuff config options
pacemaker = "round-robin"
view-timeout = 100

# Size of RSA key that certificates are signed with
# must be same size as set when generating keys with keygen executable
key-size = 2048

# This is the information that each node is given about the other nodes
[[nodes]]
id = 1
pubkey = "keys/n1.key.pub"
tls-cert = "keys/n1.crt"
client-srv-address = "127.0.0.1:8081"
replication-srv-address = "127.0.0.1:13371"
signing-srv-address = "127.0.0.1:23371"

[[nodes]]
id = 2
pubkey = "keys/n2.key.pub"
tls-cert = "keys/n2.crt"
client-srv-address = "127.0.0.1:8082"
replication-srv-address = "127.0.0.1:13372"
signing-srv-address = "127.0.0.1:23372"

[[nodes]]
id = 3
pubkey = "keys/n3.key.pub"
tls-cert = "keys/n3.crt"
client-srv-address = "127.0.0.1:8083"
replication-srv-address = "127.0.0.1:13373"
signing-srv-address = "127.0.0.1:23373"

[[nodes]]
id = 4
pubkey = "keys/n4.key.pub"
tls-cert = "keys/n4.crt"
client-srv-address = "127.0.0.1:8084"
replication-srv-address = "127.0.0.1:13374"
signing-srv-address = "127.0.0.1:23374"
```

Figure 5.4: Example configuration file for a four node cluster

When starting a certification server it looks for the hotcertification.toml file in the working directory but it can also be passed in with –config flag. The private key, the threshold key and the id are also passed in. The certification server starts up and tries to connect to the other servers with a timeout parameter currently set to 30 seconds.

```
certserver --id 1 --thresholdkey keys/n1.thresholdkey --privkey keys/n1.key
```

Figure 5.5: Starting a certification server/node

Finally, we can use an example client implementation in Fig. 5.6 that connects to a specified server and writes the received certificate to a file.

```
client --server-addr "127.0.0.1:8081" client.crt
```

Figure 5.6: Using the reference client implementation

5.4 Implementation Details

Now that we have shown the basic usage of the system we can investigate what the inner workings of the certification process are. The goal of the implementation can be seen in Fig. 5.7, which shows all phases of the process and messages that have to be exchanged in a four node cluster. It's a linear best-case progression from a client seding a CSR to it receiving a certificate. In practice, the ClientAPI, Replication and Signing stages run concurrently in order to allow the parallel processing of CSR from different nodes. This modularization of the different phases has the added benefit of allowing an easy replacement of the replication protocol from HotStuff to another algorithm if necessary. The threshold signing algorithm could also be replaced in that manner.

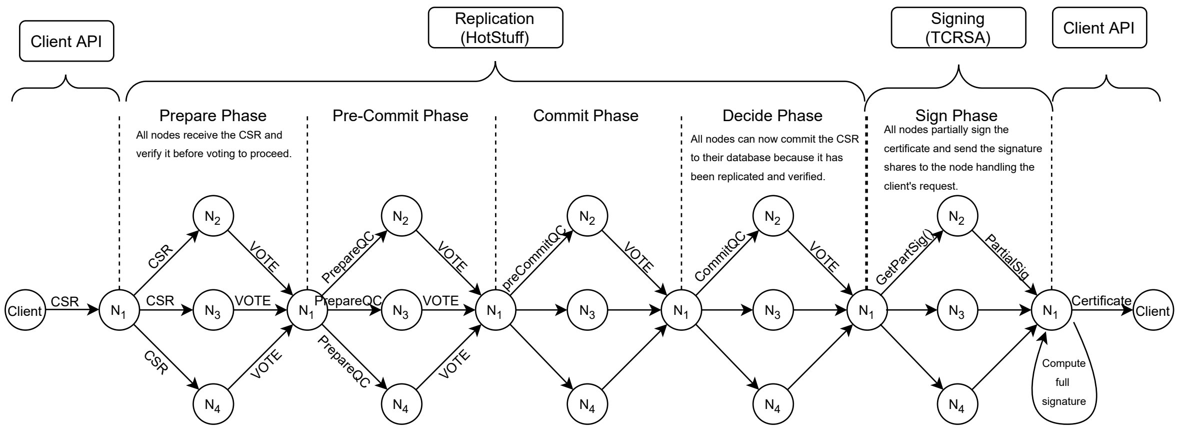
Client API
Replication (HotStuff)
Signing (TCRSA)
Client API
Prepare Phase
All nodes receive the CSR and verify it before voting to proceed.
Pre-Commit Phase
Commit Phase
Decide Phase
All nodes can now commit the CSR to their database because it has been replicated and verified.
Sign Phase
All nodes partially sign the certificate and send the signature shares to the node handling the client's request.
Client
CSR
N1
N2
N3
N4
VOTE
PrepareQC
preCommitQC
CommitQC
GetPartSig()
PartialSig
Compute full signature
Certificate

Figure 5.7: All phases

5.4.1 Interfaces

The client API seen in Fig. 5.8 was written with the in gRPC library, which defines a *service* with functions that can be called. As seen below we have a Certification Service that has one function called GetCertificate(). As arguments, this remote procedure call takes a *message* which defines exactly what kind of data is stored and the field index of the serialized data structure of a specific piece of data.

```
service Certification {
    rpc GetCertificate(CSR) returns (Certificate) {}
}

message CSR {
    uint32 ClientID = 1;
    bytes CertificateRequest = 2;
    bytes ValidationInfo = 3;
}

message Certificate {
    bytes Certificate = 1;
}

message Batch { repeated CSR CSRs = 1; }
```

Figure 5.8: The client API

The signing API in Fig. 5.9 has the same structure as the client API. It is not a public API and is only used internally for the signing phase of the certification process. As part of this thesis, the TCRSA[13] signature algorithm had to be wrapped in a networking and serialization layer, which was achieved with this gRPC interface definition. The message *TBS* stands for To Be Signed and is sent in a GetPartialSig() call to the other nodes. As can be seen, it identifies which CSR is to be signed by the hash of it. At this stage, all nodes already have validated and stored the CSR in their database to just sending the hash as the database key is enough.

```
service Signing {
    rpc GetPartialSig(TBS) returns (SigShare) {
        option (gorums.quorumcall) = true;
        option (gorums.custom_return_type) = "ThresholdOf";
    }
}

message TBS {
    string CSRHash = 1;
    bytes Certificate = 2;
}

message SigShare {
    bytes Xi = 1;
    bytes C = 2;
    bytes Z = 3;
    uint32 Id = 4;
}

message ThresholdOf {
    repeated SigShare SigShares = 1;
}
```

Figure 5.9: The signing API

5.4.2 Orchestration

Fig. 5.10 shows a small part of the Coordinator struct, which is responsible for orchestrating the **client, signing** and **replication server**. These different components are orchestrated in the **hotcertification.go** file, which connects the different components through go channels and a database. The Go channels are called queues here to better understand their function. These queues are necessary since these three processes run concurrently, so they have to process data asynchronously. Queues are the ideal data structure to buffer data in between processing steps. After receiving a client request it is passed into the AddRequest() function, which adds it to the ReplicationQueue. As part of the replication phase, the Accept() function is called, which defines exactly what makes a valid CSR. This is where policies like what a valid Proof of Identity looks is can be defined. After the CSR has been replicated and validated by all nodes the Exec() function is called which writes the CSR to the database and adds it to the SigningQueue. These are just some of the functions that are performed in order to orchestrate the different, asynchronously running processes.

```go
type Coordinator struct {
    Mut                 sync.Mutex
    ReplicationQueue    chan *protocol.CSR
    SigningQueue        chan *protocol.CSR
    FinishedCerts       chan *protocol.Certificate
    Database            map[string]*RequestInfo // simulating a basic database; the key the hash of
    Marshaler           proto.MarshalOptions    // for translating into hotstuff.Command
    Unmarshaler         proto.UnmarshalOptions  // for checking semantics of a request
    HS                  *hotstuff.HotStuff
    Log                 logging.Logger
}


func (c *Coordinator) AddRequest(csr *protocol.CSR) {...}

func (c *Coordinator) Accept(cmd hotstuff.Command) bool {...}

func (c *Coordinator) Exec(cmd hotstuff.Command) {...}
```

Figure 5.10: The Coordinator

Finally, in Fig. 5.11 the final sequence diagram can be seen. All the stages a CSR goes through are presented. Square brackets mean that a [**data structure**] is sent or added to a specific component.

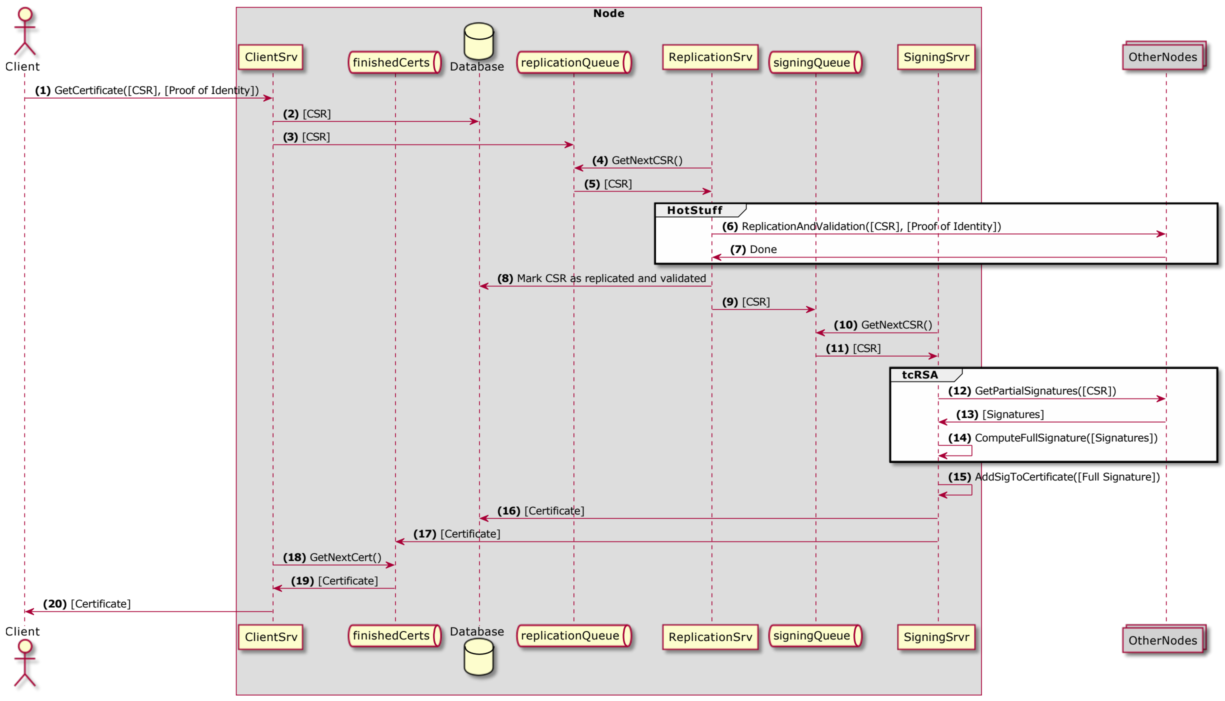

Figure 5.11: Detailed message sequence

Chapter 6

Evaluation

In this chapter, we want to evaluate the performance of the system by trying to create test scenarios that mirror a real-life deployment. The purpose of these tests is to see if the requirements and design goals have been met. Before looking at the results it should be known that this system has not been fully optimized for performance, it is still a kind of proof of concept. A couple of optimizations like batching requests and more efficient signature aggregation could still be added. As we will see the system still performs quite well. Section 6.4 discusses whether and how the requirements defined in Section 3.2 from the Analysis Chapter have been met.

6.1 Metrics and Parameters

The metrics and parameters will be tweaked to define different scenarios. The metric we will be using is the time in milliseconds that it takes from the client sending a CSR to when it receives a valid and fully signed certificate. The metric will be called ***time-to-certificate*** and will be a latency measurement of the system.

Three parameters will be tuned to provide different scenarios. The ***payload-size*** parameter will vary the amount of data that is sent in every request. Since we will be using a standard X.509 certificate signing request data structure but we can easily vary the size of the Proof of Identity/Validation Info. To test the scalability of the system the ***number-of-nodes*** parameter will vary how many nodes are part of the validation and signing. In these tests, we will assume a static byzantine adversary that controls a fraction of all participating nodes. This adversary can crash nodes to simulate a Denial-of-Service attack or hack nodes and send arbitrary data to other nodes. The parameter ***adversary-fraction*** will measure how much of the system is compromised, where the theoretical upper bound of the system is 1/3 of the nodes being compromised. So the parameters and metrics we will be varying are the following:

- ***time-to-certificate***
- ***payload-size***
- ***num-nodes***
- ***adversary-fraction***

The exact permutation of parameters will be defined in Section 6.3.

6.2 Methodology

In order to simulate a real-life deployment, the certification system is started with a certain amount of normal and malicious nodes. To approximate concurrent requests entering the system through different nodes, we let many benchmark clients connect to different nodes. These benchmark clients then cumulatively send 1000 CSRs to the nodes and measure the ***time-to-certificate***. This setup best approximates a real-life scenario where requests arrive from many different communication vectors. The certification system has to synchronize and coordinate itself to issue certificates.

An example setup can be seen in Fig. 6.1 where we have ***num-nodes=4*** and 4 benchmark clients sending 250 requests each adding up to a total of 1000 measurements of ***time-to-certificate***.

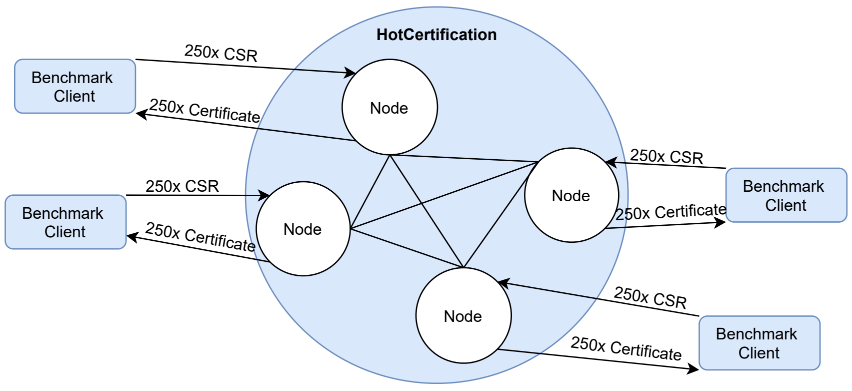

Figure 6.1: Test setup with 4 nodes

The binaries are in docker containers and networked together in a docker network. All containers are run on the same machine with the following specs:

CPU: AMD EPYCTM7551P 32-Core processor

Memory: 64 GB

Disk: Virtual disk - runs in RAM

OS: Debian 10

6.3 Test scenarios

6.3.1 Varying the payload size:

This scenario tests the throughput of the certification system. Since clients have to attach some sort of Proof of Identity to their Certificate Signing Request we want to see how increasing the payload size of the Proof of Identity is reflected in the time it takes to get a certificate. As mentioned before we have 4 benchmark clients sending these requests at the same time, which means the system is under full load.

Parameter	Value
number-of-nodes	4
adversary-fraction	0.0
payload-size	100 / 1KB / 10KB / 0.1MB

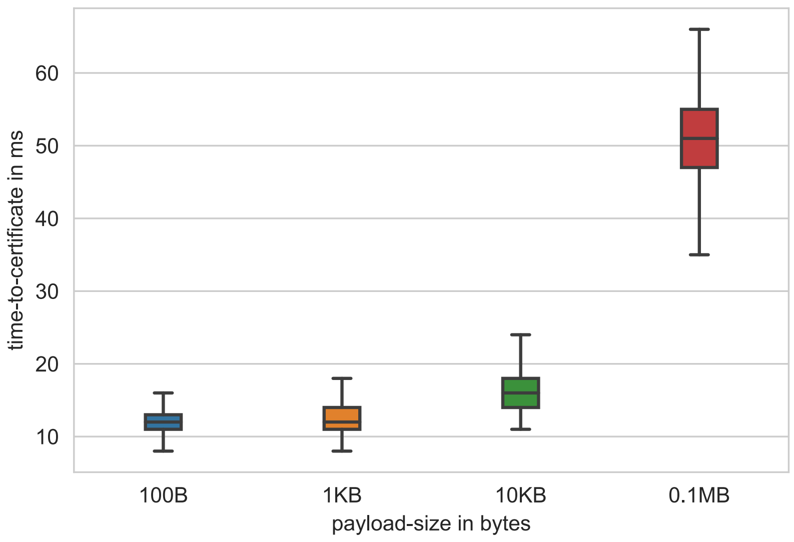

Figure 6.2: Scaling the CSR size

The x-axis of the chart in Fig. 6.2 can be thought of as logarithmic since every step increases the payload size by 10x. As we can see the trend is exponential but since the x-axis is logarithmic the time to certificate increases linearly with the number of bytes in the payload. This is the result of the message complexity of the certification process being in $O(n)$. Another measurement with a payload size of 1MB has been omitted because it took about 200ms (so continuing the trend) and it would have made the chart unreadable.

6.3.2 Scaling the number of nodes:

Increasing the number of nodes should increase the security and fault-tolerance of the system since an attacker would have to compromise more nodes. So it is instructive to see what cost, in the form of higher latency, is incurred by the system. More nodes means more overhead so this should increase the ***time-to-certificate***.

Parameter	**Value**
num-nodes	4 / 10 / 20 / 30
adversary-fraction	0.0
payload-size	100 Bytes

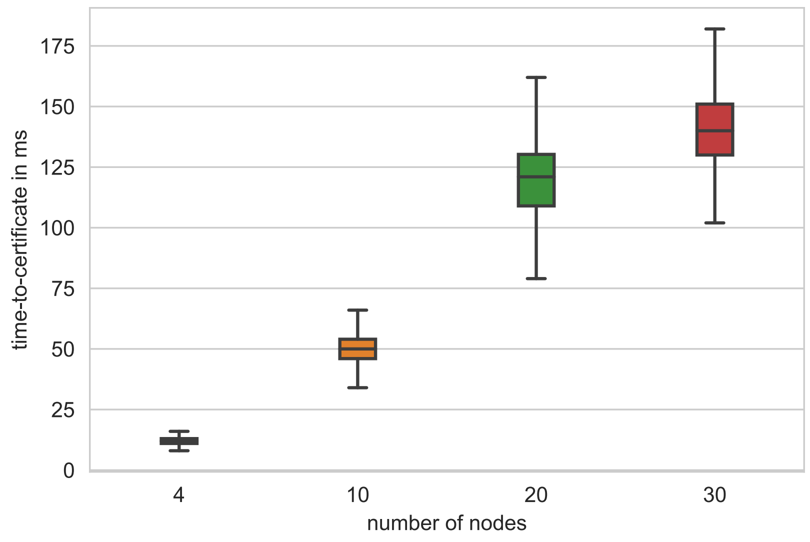

Figure 6.3: Scaling the nodes

As can be seen in Fig. 6.3 we are testing configurations with 4, 10, 20 and 30 nodes. The relationship between scaling the nodes and the ***time-to-certificate*** is linear. The protocol seems to scale quite well.

6.3.3 Adding an adversary:

In this scenario, we want to investigate the impact an adversary has on the ***time-to-certificate***. For this, a static byzantine adversary controls a fraction of the nodes. This is achieved by randomly crashing and starting up the docker containers in which the attacked nodes reside. This also leads the nodes to send faulty information to the other nodes, making this a close approximation of a byzantine adversary. In this test, we use a configuration of ten nodes and then either compromising one, two or three of the nodes. Note that the maximum amount of malicious nodes is 1/3 of the nodes because this is bounded by the consensus algorithm used for replication.

Parameter	**Value**
number-of-nodes	10
adversary-fraction	0.1 / 0.2 / 0.3
payload-size	100 Bytes

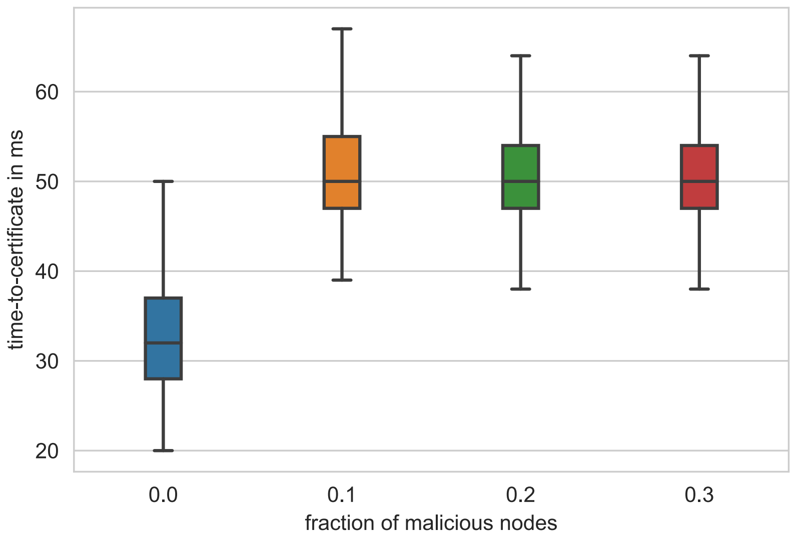

Figure 6.4: Impact of the adversary

It is interesting to see that as soon as we add a malicious node the ***time-to-certificate*** increases and then stays constant. This could be a function of the consensus algorithm since its view-change protocol is linear. The view-change protocol is an integral part of the HotStuff algorithm because it drives the protocol to consensus by defining what happens when there is a leader failure.

6.4 Requirements Analysis

This section discusses whether and how the requirements defined in Section 3.2 from the Analysis Chapter have been met. The structure of the requirements listing is the as in the aforementioned section with the difference being that instead of definition we have explanations of that requirement has been met.

Lightweight: In comparison with a similiar implementation this system shows to be more lightweight. The requirements below further qualify this statement.

- **Simplicity:** The system can be deployed in a single binary/executable and only needs three ports to communicate with the other nodes. It is also easy to deploy as part of Docker container.
- **Maintainability:** All the code and dependencies are written in the same programming language (Go), which means there is no need to be proficient in many different languages to debug the system. Furthermore, all configuration is done with a single configuration file with easy-to-understand parameters.
- **Performance:** Section 6.3.1 shows that the performance of the system scales linearly with the amount of load put on the system. The throughput scales in $O(n)$.

Tamper-resistance: The system can be actively attacked by an adversary or just experience normal faults and still keep running. The requirements below specify exactly how tampering with the certification process has been made more difficult.

- **No single point of failure:** As seen in Section 6.3.3 multiple nodes of the certification cluster can fail without the whole system failing.
- **Autonomy:** Each node receives the same information and can decide on its own whether to authorize (by signing) a given Certificate Signing Request (CSR). A node cannot be tricked into altering its state from the outside since it relies on its own state to make decisions.
- **Security/Safety:** A policy of what constitutes a valid Proof of Identity can be set and therefore no invalid certificates can be issued.
- **Liveness:** This also follows from the results of Section 6.3.3, which shows that as long as the adversary controls less than a third of the nodes the system will keep issuing certificates. In more practical terms this means it is harder to perform a Denial-of-Service attack.

- **Multi-party authorization:** The results from Section 6.3.2 show that the system can increase the number of parties needed to authorize a CSR. We've met this requirement and also made it scalable and customizable.
- **X.509 compliant:** This implementation uses X.509 Certificate Signing Request for client requests and issues X.509 Certificates that are signed with RSA. This system can easily be used in real life since it adheres to the X.509 Standard defined in RFC3279 [14].

Chapter 7

Related Work

In this chapter, we survey some other projects that change the way identities are bound to public keys. Some of the systems are completely open and cannot be customized to specific use cases, whereas others are proprietary software that is used internally at big software companies. These systems are useful to compare this system to since they share similar properties as the HotCertification protocol, namely that there exists no single point of failure by performing some sort of multi-party authorization.

7.1 Namecoin

Namecoin[21] is a Bitcoin fork whose primary purpose is to store key-value pairs on a decentralized, public and censorship-resistant ledger. Keys can be any arbitrary identity and values can store anything from email addresses or domain names to PGP key fingerprints. Anyone wishing to get information regarding an identity (key) queries the blockchain for the information and can be sure that the information is correct. Since it is a fork of the Bitcoin project it relies on economic incentives to get the system to add an entry to the ledger. The native currency, called namecoin(NMC), is given to miners as transaction fees. This system is a radical departure to the existing centralised system and also relies on computationally inefficient consensus mechanisms such as Proof of Work.

7.2 Microsoft's ION

ION [22] is quite a new development and a Layer 2 protocol on top of the Bitcoin Blockchain. It uses the Sidetree[23] protocol to composes an overlay network of independent peer nodes that interact with the underlying Bitcoin Blockchain to issue Distributed Identifiers(DID)[24]. This is done by leveraging Bitcoin's strong security guarantees to produce an eventually strongly consistent view of all DIDs in the network. DIDs are controlled and owned by individuals and not central entities like companies. This means that once a person owns an identity it can't be taken away by anyone. At its core, it's a decentralized protocol that protects people from violation of their digital rights.

7.3 Google's Certificate Transparency

A distributed, public, append-only log of all issued certificates that aims to speed up detection of mis-issued and stolen certificates as well as identifying rogue CAs[25]. Consistency and integrity of the ledger is guaranteed by log monitors that constantly check whether different log servers are following rules. Additionally, browsers implement certificate auditing clients that ask the log servers about revoked or false certificates. This system is designed for earlier detection, faster mitigation and better oversight of the vast and often complex certificate issuance environment. Unfortunately, this does not address the fundamental problem of false certificates being issued in the first place but rather a way to quickly recognize a breach in the chain of trust.

Chapter 8

Conclusion

This thesis presents a relatively lightweight implementation of a Certificate Authority, that can issue certificates in a malicious environment. The main goal was to have **no single point of failure** in order to make the certificate issuance process **tamper-resistant**. Previous work that used a complex Distributed Ledger Technology (DLT) framework was used to distill requirements for this more **lightweight** implementation. This was achieved by designing a system that consists of a byzantine fault tolerant state machine replication algorithm (BFT-SMR) and a threshold signature scheme. The BFT-SMR algorithm served the purpose of copying the functions of certificate signing request validation across a cluster of nodes so that all nodes deterministically synchronize to the identical state. The threshold signing scheme made signing a certificate a **multi-party authorization**, so that issuing invalid certificate would become more difficult for an adversary. These two building blocks were subsequently used to implement and deploy a distributed certification authority. Subsequents tests, that mimicked a real-life environment, showed that the certification system proved to be **lightweight** because it showed low latency response to Certificate Signging Requests.

8.1 Future Work

At the moment this implementation is still only a proof of concept. It works but there still exist a lot of inefficiencies. Below are a couple of things that could be improved and added in the future.

Setup/key generation: To generate cryptographic material for TLS and the threshold signature signing scheme we rely on a trusted dealer phase. This means when setting up the system an administrator holds all the secrete keys and then distributes them among the group of nodes. In the future, it would be great to implement a setup phase in which all participants use an interactive key generation algorithm whereby all nodes compute the shared public key from their self-generated private key without that private key ever leaving their own database.

Batching: Every CSR entering the system is handled alone, which means there is a lot of overhead when processing a single request. In order to decrease the average network bandwidth, a method for buffering CSRs and batching them into blocks should be implemented.

Signing generic data: X.509 compatibility was a key requirement of this thesis, so the only data structure currently able to be certified by the system are X.509 certificates. This method for issuing certificates could be abstracted to validate and sign any type of data. An example would be signing cryptographic bearer tokens like macaroons[26] or JWTs and implement an authentication service like Auth0[27].

Bibliography

[1] L. Lamport, R. Shostak, and M. Pease, *The byzantine generals problem*, 1982. [Online]. Available: https://lamport.azurewebsites.net/pubs/byz.pdf.

[2] M. Castro and B. Liskov, *Practical byzantine fault tolerance.* [Online]. Available: http://pmg.csail.mit.edu/papers/osdi99.pdf.

[3] L. Lamport, *The part-time parliament*, 1998. [Online]. Available: https://lamport.azurewebsites.net/pubs/lamport-paxos.pdf.

[4] S. Nakamoto, *Bitcoin: A peer-to-peer electronic cash system*, 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf.

[5] M. J. Fisher, N. A. Lynch, and M. S. Paterson, *Impossibility of distributed consensus with one faulty process*, 1985. [Online]. Available: https://groups.csail.mit.edu/tds/papers/Lynch/jacm85.pdf.

[6] F. B. Schneider, *The state machine approach: A tutorial.* [Online]. Available: https://www.cs.cornell.edu/fbs/publications/ibmFault.sm.pdf.

[7] *The weakest link in the chain: Vulnerabilities in the ssl certificate authority system and what should be done about them.* [Online]. Available: https://www.accessnow.org/cms/assets/uploads/archive/docs/Weakest_Link_in_the_Chain.pdf.

[8] F. Rezabek, *Verifiable secret sharing and threshold signatures for tamper-resistant signature services.*

[9] C. Rudolf, *Tamper-resistant creation of integrity tokens for trustworthy communication in cyberphysical systems.*

[10] L. Foundation, *Hyperledger fabric.* [Online]. Available: https://www.hyperledger.org/use/fabric.

[11] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Songs, *The honey badger of bft protocols.* [Online]. Available: https://eprint.iacr.org/2016/199.pdf.

[12] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, *Hotsuff: Bft consensus in the lens of blockchain.* [Online]. Available: https://arxiv.org/abs/1803.05069.

[13] V. Shoup, *Practical threshold signatures.* [Online]. Available: `https://www.iacr.org/archive/eurocrypt2000/1807/18070209-new.pdf`.

[14] *Algorithms and identifiers for the internet x.509 public key infrastructure certificate and certificate revocation list (crl) profile.* [Online]. Available: `https://datatracker.ietf.org/doc/html/rfc3279`.

[15] J. I. Olsen, BondeKing, and H. Meling, *Relab/hotstuff.* [Online]. Available: `https://github.com/relab/hotstuff`.

[16] *Relab: Resilient systems lab.* [Online]. Available: `https://www.uis.no/en/relab-reliable-systems-lab`.

[17] *Golang threshold cryptography library - rsa implementation.* [Online]. Available: `https://github.com/niclabs/tcrsa`.

[18] *Protocol buffers are a language-neutral, platform-neutral extensible mechanism for serializing structured data.* [Online]. Available: `https://developers.google.com/protocol-buffers`.

[19] *Grpc: A high performance, open source universal rpc framework.* [Online]. Available: `https://grpc.io/`.

[20] R. Schleithoff, *The hotcertification ca.* [Online]. Available: `https://github.com/raphasch/hotcertification`.

[21] *Namecoin.* [Online]. Available: `https://www.namecoin.org/`.

[22] *Ion.* [Online]. Available: `https://identity.foundation/ion/`.

[23] *Sidetree protocol.* [Online]. Available: `https://github.com/decentralized-identity/sidetree`.

[24] *Decentralized identifiers (dids).* [Online]. Available: `https://w3c.github.io/did-core/`.

[25] *Certificate transparency.* [Online]. Available: `https://certificate.transparency.dev/`.

[26] A. Birgisson, J. G. Politz, Úlfar Erlingsson, A. Taly, M. Vrable, and M. Lentczner, "Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud", in *Network and Distributed System Security Symposium*, 2014.

[27] *Auth0: Access tokens.* [Online]. Available: `https://auth0.com/docs`.