


Technische Universität München
Department of Informatics

Bachelor's Thesis in Informatics

A Tamper-Proof Certificate Issuance Process Based on Distributed Ledger Technology

Jan Felix Hoops

Technische Universität München

Department of Informatics

Bachelor's Thesis in Informatics

A Tamper-Proof Certificate Issuance Process Based on Distributed Ledger Technology

Ein manipulationssicherer Zertifikatsausstellungsprozess auf Basis von Distributed Ledger Technologie

Author	Jan Felix Hoops
Supervisor	Prof. Dr.-Ing. Georg Carle
Advisor	Dr. Holger Kinkelin, Dr. Heiko Niedermayer
Date	May 14, 2018

Informatik VIII
Chair for Network Architectures and Services

I confirm that this thesis is my own work and I have documented all sources and material used.

Garching b. München, May 14, 2018

Signature

Abstract

Today, many security relevant systems depend on protocols that authenticate entities (users, services, etc.) using asymmetric cryptography. One central question in this context is how the identity of an authenticating entity can be attached to the presented public key. Currently, this is often done by public key infrastructures (PKI) that manage X.509 certificates. However, examples show that the certification process can be attacked, or that certificate authorities can act maliciously or even carelessly. These problems can result in malicious (i.e. illegitimate) certificates being issued, allowing an attacker to pose as a third party in front of other entities.

To mitigate this situation, we present a system that enforces a secure certificate issuance process, heavily inspired by the DFN-AAI's (Deutsches Forschungsnetz Authentifikations- und Autorisierungs-Infrastruktur) certification process. With this thesis we are making two main contributions: Firstly, we conduct a multi-party authorization (MPA) process for certificate signing requests (CSR). Secondly, we provide accounting information that makes this process transparent and trustworthy.

Our solution is based on distributed ledger technology, more specifically, Hyperledger Fabric (HLF). HLF is a framework that offers a Byzantine fault tolerant execution environment that we use to conduct the MPA of CSRs and tamper-resistant, blockchain-based storage we use to account this MPA process.

Our evaluation shows that our system prevents that a single malicious or careless person involved in the certificate issuance process is able to issue a valid certificate. Furthermore, the system itself is resistant to attacks. Our system is not able to defend against attackers able to, for instance, compromise a CA's signing key and issue malicious certificates. However, the attacker will not be able to fake accounting information which is created during a legitimate certificate issuance process. Hence, our system provides valuable information to detect illegitimately issued certificates.

Zusammenfassung

Viele heutige sicherheitsrelevante Systeme hängen von Protokollen ab, die Entitäten (z.B. Benutzer, Services, etc.) mittels asymmetrischer Verschlüsselung authentisieren. In diesem Kontext ist eine zentrale Frage, wie ein öffentlicher Schlüssel einer sich authentisierenden Entität zugeordnet werden kann. Das ist heute vorherrschend durch Public Key Infrastrukturen (PKI), die X.509 Zertifikate verwenden, umgesetzt. Allerdings hat die Vergangenheit gezeigt, dass der Zertifikatsausstellungsprozess angreifbar ist, und dass Certificate Authorities (CA) fahrlässig oder sogar bösartig handeln können. Diese Probleme können zur Ausstellung von bösartigen (i.e. illegitimen) Zertifikaten führen, die es einem Angreifer erlauben, sich vor dritten als jemand anderes auszugeben.

Um dieser Situation zu begegnen, präsentiert diese Arbeit ein System, das einen sicheren Zertifikatsausstellungsprozess umsetzt. Dieser Prozess orientiert sich stark an dem von der DFN-AAI (Deutsches Forschungsnetz Authentifikations- und Autorisierungs-Infrastruktur) genutzten Zertifikatsaustellungsprozess. Hauptsächlich leistet diese Arbeit zwei Beiträge: Erstens, wird für Certificate Signing Requests (CSR) Multi-Party Autorisierung (MPA) durchgeführt. Zweitens, bietet das System Prozessdaten, die den Prozess transparent und vertrauenswürdig machen.

Die in dieser Arbeit vorgestellte Lösung basiert auf Distributed Ledger Technologie, genauer auf Hyperledger Fabric (HLF). HLF ist ein Framework, dass verteilte Programmausführung mit byzantinischer Fehlertoleranz bietet, die im Rahmen dieser Arbeit genutzt wird, um MPA von CSRs durchzuführen. Außerdem verfügt HLF über manipulationsresistenten Blockchain-basierten Speicher, der genutzt wird um den Zertifikatsaustellungsprozess nachvollziehbar zu machen.

Die finale Evaluation zeigt, dass das System verhindert, dass ein einziger bösartiger oder fahrlässiger Teilnehmer am Zertifikatsausstellungsprozess valide Zertifikate ausstellen kann. Des Weiteren ist das System selbst resistent gegen Angriffe. Das System ist jedoch nicht in der Lage, gegen einen Angreifer zu schützen, der zum Beispiel den privaten Schlüssel einer CA übernommen hat und damit direkt bösartige Zertifikate ausstellt. Allerdings kann der Angreifer die Prozessdaten, die während des legitim durchgeführten Zertifikatsausstellungsprozesses erzeugt werden, nicht fälschen. Daher bietet das System wertvolle Informationen, um illegitim ausgestellte Zertifikate zu erkennen.

Contents

1 Introduction 1

2 Background 3
2.1 Public Key Infrastructure . . . 3
2.1.1 Overview . . . 3
2.1.2 X.509 . . . 4
2.1.3 DFN-AAI . . . 5
2.2 Blockchain . . . 6
2.2.1 Overview . . . 6
2.2.2 Permissioned and Private Blockchains . . . 7
2.2.3 Hyperledger Fabric . . . 7
2.2.4 Architectural Overview . . . 8
2.2.5 Transaction Flow . . . 9
2.3 Scenario . . . 10

3 Analysis 11
3.1 Risks and Challenges of Public Key Infrastructures . . . 11
3.2 Requirements . . . 13
3.2.1 Functional Requirements . . . 13
3.2.2 Non-Functional Requirements . . . 13
3.2.3 Security Requirements . . . 13
3.3 Possible Solutions . . . 14

4 Design 17
4.1 System Overview . . . 17
4.2 Distributed Ledger supported Certificate Issuance Process . . . 18
4.3 Chaincode Functionality . . . 19

5 Implementation 25
5.1 Technology Choices . . . 25
5.2 Overview . . . 25
5.3 Hyperledger Fabric Chaincode . . . 27

5.4 Implementation Challenges 30
5.5 Using the Implementation 31
5.5.1 Initial Setup 31
5.5.2 Running the Implementation 32
5.5.3 Terminating the Virtual Machine 32

6 Evaluation 33
6.1 Security Evaluation 33
6.2 Additional Considerations 35

7 Related Work 37
7.1 PGP 37
7.2 Lightweight Directory Access Protocol 37
7.3 Certificate Transparency 38
7.4 Blockchain-Based Identity Management Solutions 38
7.4.1 A Blockchain-Based PKI Management Framework 38
7.4.2 The Internet Blockchain 39
7.4.3 Blockstack 39
7.4.4 Hyperledger Indy 39
7.4.5 Sovrin 40

8 Conclusion 41

Bibliography 43

List of Figures

2.1 Simplified Blockchain Structure 6
2.2 Hyperledger Fabric Transaction Flow 9

3.1 Attack tree for a certificate authority. 12

4.1 Overview of the system. 18

5.1 Simplified overview of the system implementation. 26

6.1 Attack tree for the certificate issuance system. 34

List of Tables

2.1 Distinguished Name Attributes 4

4.1 UserRecord 19
4.2 CSRRecord 20
4.3 RARecord 21
4.4 CertRecord 22

Chapter 1

Introduction

Identity management is an essential part of many infrastructures. Examples range from all kinds of access control systems to government administration applications. Especially with the Internet being more important than ever, secure identity management is vital.

Todays identity management heavily relies on X.509 certificates to authenticate entities like users, servers, devices, etc. The big problem with X.509 is that, despite the certificate authorities being aware of the importance of being thorough in order to be sustainable commercially, malicious (i.e. illegitimate) certificates are discovered periodically. This leaves one to wonder about the undiscovered amount of malicious certificates still in circulation. One instance of this occurred in 2011, when the dutch certificate authority DigiNotar was forced to declare bankruptcy following a successful hack [1]. The hacker gained access to their systems and was able to generate false certificates for several well known sites (e.g. google.com) without anyone immediately noticing. And up to this day, the hacker could not be identified without doubt. This incident has damaged the trust in DigiNotar permanently and also has shown how easily the integrity of a certificate authority can be compromised without anyone noticing. Preventing the recurrence of catastrophic failures like this is a huge challenge our interconnected society has to face.

Two major attack vectors include:

- The *certificate authority's* system is compromised or their private key is.
- A *representative* (e.g. employee) of the certificate authority is careless, intentionally malicious, or their account has been compromised.

With this bachelor thesis aims to improve certificate issuance by creating a secure certificate issuance process based on distributed ledger technology. This new certificate issuance process is intended to enforce *multi-party authorization* and provide *transparency*. These properties lower the chance of a certificate authority representative being able to issue malicious certificates, either by mistake or intent, and make it possible to fully trace an attack on the system.

As mentioned earlier, the certificate issuance system shall be based on distributed ledger technology. A distributed ledger provides a means to keep records while maintaining high standards in non-mutability, accountability and reliability. All of these properties are highly desirable. While distributed ledgers are mostly still used for crypto-currencies, other areas of application, such as permissioned ledgers for business, are being explored. The distributed ledger implementation used in this thesis is a permissioned blockchain.

The inspiration for a secure certificate issuance process came from the DFN (Deutsches Forschungsnetz). The DFN already successfully uses one for their public key infrastructure. That process is mostly analog and provides a promising foundation for a software implementation.

The ultimate goal of this thesis is to provide valuable work towards a truly distributed certificate issuance system that automatically enforces and supports the process. The following two **research questions** are at the core of that:

- What advantages does a distributed ledger based certificate issuance system provide?
- What are the limits of what a system like this can accomplish?

The thesis is organized as follows: Chapter 2 provides some essential background knowledge on public key infrastructure and blockchain. Chapter 3 analyzes the current problems in public key infrastructure, discusses requirements for a suitable solution, and proposes a solution based on distributed ledger technology. In chapter 4 a certificate issuance system is designed based on the findings in the prior chapter. Chapter 5 presents selected aspects of the proof-of-concept implementation for the previously designed system. The system is evaluated in chapter 6. Chapter 7 presents related work, and finally, chapter 8 concludes the thesis.

Chapter 2

Background

Before diving deeper into the problem, we have to establish some background knowledge. This chapter introduces essential topics for this thesis, namely *Public Key Infrastructure* and *Blockchain*.

2.1 Public Key Infrastructure

2.1.1 Overview

Public key cryptography is the basis for every identity management system. Every party owns at least one key pair, composed of a public and a private key. By proving ownership of a private key, a party can prove their identity. This is possible by challenging the party with a piece of data (usually a nonce), that they have to sign. Signing means decrypting the data with a private key. The challenging party can verify the identity of the challenged one by using the challenged public key to decrypt the received signature. If this decryption yields the original piece of data, the challenge was successful. This is useful in all kinds of situations, such as verifying a requested website is coming from an authentic server.

However, this still leaves a few essential questions unanswered:

- How is it possible to get someone's public key?
- How is it possible to know who a public key belongs to?
- How is it possible to know if a private key is still safe?

Public Key Infrastructures (PKI) try to answer these questions. They provide a distributed, publicly accessible infrastructure, mapping a public key to some identifier (e.g. a name or a domain).

Table 2.1: Distinguished Name Attributes

Name	Attribute ID	Description
Country	C	The country of origin.
State	ST	The state of origin
Locality	L	Usually the city of origin.
Organization	O	The name of the organization (e.g. company name).
Organizational Unit	OU	The name of the unit within the organization (e.g. company department).
Common Name	CN	The name of the subject (e.g. John Doe).
Email Address	EMAIL	The email address of the subject.

2.1.2 X.509

The Internet PKI is based on X.509 certification [2]. This is a hierarchical, tree-like structured approach, comparable to the directory service LDAP. Each "node" of the tree is one certificate authority (CA). A CA is a commercial party, that signs public keys of subscribers. In doing so, the CA assures a user, that a specific public key belongs to a specific subscriber, assuming the user trusts the CA. Because a CA can also sign other CAs, establishing a tree-like structure, a user only has to trust the root CA of a PKI, in order to be sure that all subscriber certificates in the PKI are correct.

A X.509 certificate consists of three main components: a `TBS certificate`, an `algorithm identifier` and a `signature value`. The TBS certificate mostly holds information about the subject, like the subjects *distinguished name* and public key. A distinguished name consists of attributes [3], similar to the directories in LDAP. Table 2.1 shows an overview over some of the most important distinguished name attributes. Another important value found in the TBS certificate is the validity time, defined by a starting epoch `notBefore` and an ending epoch `notAfter`. Certificates are not indefinitely valid because with time, the risk of a key having been compromised rises. Back to the remaining two components, the algorithm identifier specifies which algorithm was used to sign the certificate by the CA and signature value contains the actual signature.

X.509 did not have *certificate revocation* from the start. However, in order to guarantee the integrity of a PKI, the need to invalidate an issued certificates eventually arises. Reasons can range from a certificate holder's key being compromised before the certificate expiration date to the certificate having been issued by a malicious party that temporarily gained control over the CA. Certificate revocation was later on implemented through

the addition of certificate revocation lists (CRL). These lists are usually maintained by each CA for the certificates they issued and allow users to check if a certificate they have been presented with was revoked.

2.1.3 DFN-AAI

Since October of 2007, the DFN (Deutsches Forschunksnetz) operates a PKI for scientific institutions [4]. This PKI is called the DFN-AAI (DFN Authentifikations- und Autorisierungs-Infrastruktur) and consists of a number of German scientific institutions. Applications using the DFN-AAI range from administration systems to e-learning platforms. The DFN CA is signed by the German Telekom, assuring that all certificates issued by the DFN CA itself, or one of its child CAs, are accepted in browsers and operating systems.

In order to actively participate in the DFN-AAI, an institution can apply to become a DFN IdP (Identity Provider). A DFN IdP is in charge of a CA that is operated by the DFN and can be used in order to certify associated web servers, as well as email addresses for its members (e.g. university for students and staff). This federated approach is cheaper in terms of hardware, software and staff, than each institution operating their own CA. The certificates issued conform to the X.509 standard. This is a rough outline of the application process required in order to become a DFN IdP:

1. Signing of a contract between the DFN and the institution.
2. Participation of the institution in the DFN-AAI-Test, a test environment used to familiarize new members with the infrastructure.
3. Finalization of all configurations and transition to the production environment.

DFN IdPs can match several different security classifications (lowest to highest):

1. **DFN-AAI Test** This class only exists for testing purposes.
2. **DFN-AAI Basic** This is the lower security class.
3. **DFN-AAI Advanced** The highest security class.

These classifications dictate the standards an institution has to uphold regarding identification, authentication and data storage. A service provider can specify the minimum security classification required in order to access their service in accordance to the sensitivity of their service.

A user, affiliated with an institution that is a DFN IdP, can apply for a certificate confirming his identity. According to DFN-AAI Advanced, he has to personally identify himself in front of a trusted party (e.g. the DFN representative at the user's institution) using an official document (e.g. passport).

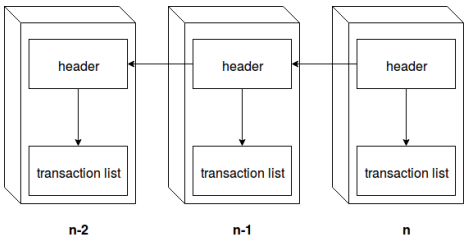

Figure 2.1: Simplified Blockchain Structure
Every arrow symbolizes a connection through a hashing operation. The arrow points from the place the hash is stored to the origin of the hash.

2.2 Blockchain

2.2.1 Overview

A blockchain is an implementation of a *distributed ledger* that has rapidly gained popularity with the rise of Bitcoin starting in 2008 [5]. It can be seen as a distributed database requiring nodes to reach quorum for any modification and featuring high redundancy. The blockchain is replicated using a peer-2-peer system and every participating node keeps a full copy. The major benefit of a blockchain is that it provides high trustworthiness without relying on a trusted third party.

In contrast to a traditional database, a blockchain does not store tables, but transactions. Generally, a transaction performs some change to the state of the ledger (e.g. transfer an asset). Transactions are grouped up into Blocks. These blocks consist of a header for meta data and a list of transactions. Every block is connected to its predecessor by containing the hash of the predecessors header in its own header. The header hash is used to uniquely identify a block. By constructing a merkle tree over all transactions of a block and storing the root in the header of said block, it is ensured that the transactions also influence the block header hash. This secures the transactions, because it allows for easy detection of manipulation.

The essence of this structure is visualized in Figure 2.1. The schematic shows part of a blockchain with emphasis on the chain of hashes. Every arrow symbolizes a hashing operation. It can be seen, that the header of block `n` contains the hash of the header of block `n-1`. These are the horizontal arrows. The vertical ones show the transactions of a block being hashed into its header.

Only the very first block of a blockchain has to deviate from the described structure as it has no predecessor. This block is a kind of dummy block. It is called the genesis block and contains placeholder or initial values. This block usually is hardcoded in every client for a blockchain.

A consensus mechanism is used to decide on each new block in order to ensure a

consistent state of the ledger across the network. There are roughly two groups of consensus mechanisms in use: *leader election* (e.g. Bitcoin mining), also known as "Nakamoto consensus" named after the Bitcoin creator [6], and traditional byzantine fault tolerant algorithms like *Practical Byzantine Fault Tolerance* (PBFT) [7]. The choice of consensus mechanism defines most of the qualities of a blockchain, like transaction throughput and scalability.

With a definite chain of blocks, the state of the ledger can be computed from the blocks. This is possible by simply applying all transactions to the initial state in order from oldest to newest.

2.2.2 Permissioned and Private Blockchains

With the success of public blockchain applications like Bitcoin, the interest in applying blockchain technology to corporate scenarios has grown. These new blockchain solutions often are *permissioned* and some additionally are *private* [8]. A permissioned blockchain is a blockchain that restricts participation in the consensus mechanism using additional security systems. A private blockchain is a blockchain that restricts read and write access. This restriction can be achieved by an access restricted blockchain deployed on a public network, as well as a standard blockchain deployed on a private network. Both of these restrictions usually result in a lot fewer nodes being part of the network and typically all of these nodes can be authenticated.

2.2.3 Hyperledger Fabric

Hyperledger [9] is a project aimed towards the development of permissioned and private blockchains for business use. This initiative was brought to life by the Linux Foundation in 2015 and is intended to form a community developing permissioned blockchains together. It is a global collaboriation with prestigious members like IBM, Intel, SAP and many more. There are several different projects being developed, and all of them are open source.

One of these projects is *Hyperledger Fabric* [10]. It was originally proposed by Digital Asset and IBM in 2016 as the result of a hackathon [11]. In the meantime, it has become one of the more advanced Hyperledger projects and hit version `Alpha 1.0` in July of 2017 [12].

Fabric is not only a distributed ledger, but also a *smart contract engine*. This means that Fabric does not just function as distributed data storage, but also as distributed execution environment. Thanks to this, Fabric is able to securely and automatically enforce arbitrary business logic called *chaincode*.

2.2.4 Architectural Overview

This section is based on the Fabric documentation [8] for `version 1.1`. As fabric is still in active development, future versions may differ, even though the basic architecture should be untouched.

Fabric features two types of transactions:

- **Deploy Transaction** These transactions are used to deploy (i.e. install) chaincode on the blockchain.
- **Invoke Transaction** These transactions are used to invoke (i.e. execute) chaincode functions on the blockchain. When successful, these may change the state of the ledger.

The state, influenced by the invoke transactions, is a *key-value store*. It maps a key to a tuple containing a value and a version. This version is increased every time the key is written to.

A Fabric network consists of three different types of nodes:

- **Client** A client submits chaincode invokation transactions to the network.
- **Peer** These nodes maintain the state of the ledger and commit new transactions. Additionally, a peer node can also have the role of an *endorser*. This endorser role is specific to one particular chaincode and entails that the node can endorse a transaction before it is committed.
- **Orderer** Together, these nodes form the *ordering service*. The ordering service is Fabrics consensus mechanism. It is responsible for ordering all transactions and broadcasting them to all peers in the network. The exact implementation of this service is free to choose for the individual setup.

Being a permissioned blockchain, Fabric is access restricted. To enforce this, the system needs to be able to authenticate users and nodes. Being designed to be run by multiple organizations, that do not necessarily have to trust each other, Fabric implements this using *membership service providers* (MSP), whose exact implementation is free to choose (e.g. standard x.509 certificate authority). These MSPs are affiliated with one of the participating organizations (every organization needs to have at least one) and identify their members.

For further privacy, a Fabric network is split up into channels. These allow for confidential transactions to be conducted without the rest of the network being able to read them. A channel is defined by its members, the member's anchor peers (i.e. peers listed in the configuration, that allow further peer discovery), the shared ledger, installed chaincode and the ordering service nodes.

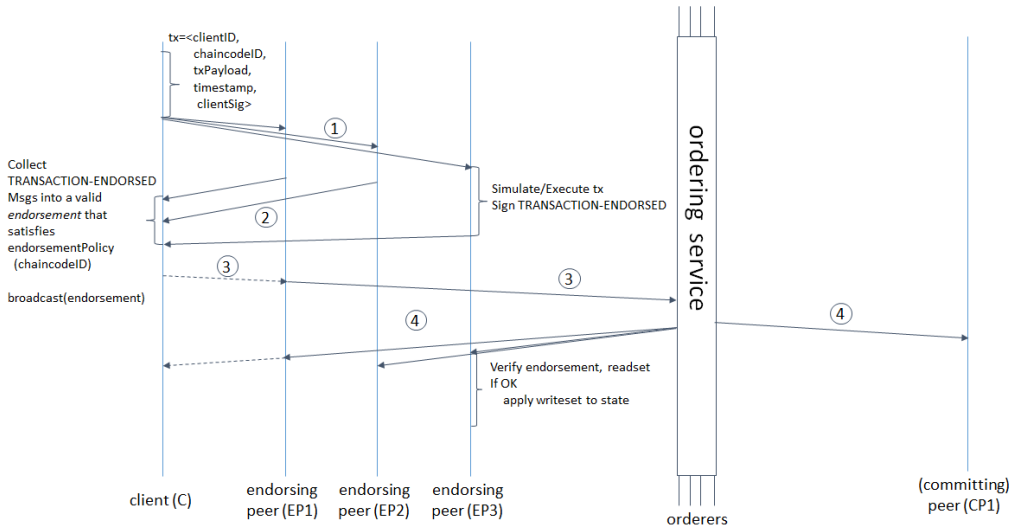

Figure 2.2: Hyperledger Fabric Transaction Flow
This illustration has been taken from the Fabric architecture documentation [13]. It displays the common-case path a transaction takes.

2.2.5 Transaction Flow

In order to show how the previously described components of Hyperledger Fabric work together, we will now consider the common-case path of a transaction as pictured in Figure 2.2.

1. The client creates a transaction proposal and sends it to a set of endorsing peers. This set is determined by the *endorsement policy* associated with the invoked chaincode. If for instance, the policy only requires an endorsement of at least one peer from one specific organization, the client only sends the transaction proposal to peers of that very organization.

2. Each endorsing peer simulates the transaction after confirming that it is valid regarding form and issuer and has not been submitted yet. If the simulation is successful, the peer returns a signed *proposal response* that contains a *read-write set* (i.e. set of all ledger keys being read from or written to).

3. After having received enough proposal responses to satisfy the endorsement policy, the client assembles them into a transaction. This transaction is sent to the ordering service.

4. The ordering service groups the transactions into blocks and sends them to all peers. Each peer verifies the transactions of the block (i.e. verifies that the endorsement policy has been met) and commits them to their copy of the ledger. Additionally the client is notified of the successful commit of his transaction via

an event.

2.3 Scenario

The inspiration for the scenario considered in this thesis is the DFN issuing certificates to users affiliated with an institution being a DFN IdP. However, there are some notable differences.

This thesis is focused on the scenario of only one certificate authority issuing certificates to a select group of users. This CA is directly operated by a federation, in contrast to the DFN CA creating CAs for every federation member. Users affiliated with a federation member can obtain certificates confirming their identity through a certificate issuance system.

The entire (successful) process is as follows:

1. The user creates a certificate signing request and deposits it in the system.
2. The user physically appears in front of a representative of the CA and identifies himself with an official document (e.g. passport).
3. The representative checks the certificate signing request and sends his approval to the system.
4. The CA fulfills the approved certificate signing request by issuing a corresponding certificate through the system.

Chapter 3

Analysis

3.1 Risks and Challenges of Public Key Infrastructures

Identity management is an essential part of many infrastructures and failure in identity management often puts the underlying system(s) at risk. For instance, a private data server is no longer private if the access control mechanism relies on manipulated identity data.

This thesis is focused on identity management through a public key infrastructure adhering to the x.509 certification standard. The main goal of an attack on a system like this is nearly never compromising the PKI itself. It rather is to obtain malicious certificates that allow the attacker to pose as some third party and thus can be further used to high effect in subsequent attacks. The exact consequences of such an attack highly depend on the attack's target and could be anywhere in the range from an attacker being able to access classified data on secure servers supporting client certificate authentication to providing an attack with the possibility to harvest user passwords from every Internet platform on a massive scale.

There are generally two basic options an attacker has when wanting to impersonate someone: He can either try to directly obtain a malicious certificate or he can try to obtain an inconspicuous certificate with certificate signing privilege. While the impact of the first option is not to be neglected, the second one is catastrophic, because it effectively turns the attacker into a trusted certificate authority allowing him to impersonate anyone at will. Both of these options are reachable following very similar attack vectors and will just be viewed as one under the name of malicious certificate from here on.

For security analysis, this thesis considers a single attacker who has exactly one of the following roles:

- **User** The attacker can be a normal user within the public key infrastructure

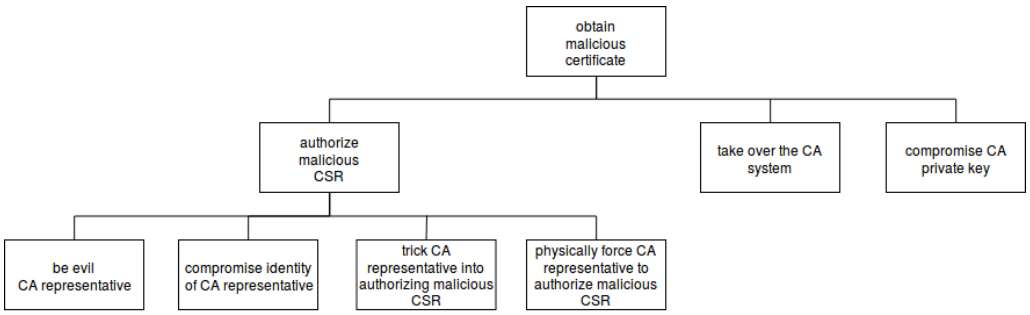

Figure 3.1: Attack tree for a certificate authority.

without any special privileges.

- **CA Representative** A CA representative is anyone with the authority to authorize certificate signing requests and thus to issue certificates on behalf of the CA.

- **CA Administrator** An administrator of the CA system has full access to the software and possibly to the hardware as well.

In order to impersonate someone in a traditional public key infrastructure, an attacker has several attack vectors to choose from. These are depicted in the attack tree in Figure 3.1. The attack vectors can be grouped how they attack the system:

- **CA Representative** The CA representative could be deceived through social engineering, forced to authorize a malicious certificate signing request, be evil, or have their identity stolen. This attack is possible for any of the previously listed attackers, but especially easy if the attacker is a CA representative himself.

- **CA system** The CA system itself can be compromised allowing direct issuance of malicious certificate. This is possible and has happened in the past (e.g. DigiNotar [1]). An attack like this is executable for a standard user (and thus also for a CA representative) and easy for a CA administrator.

- **CA Private Key** The CA private key could be stolen enabling an attacker to create malicious certificates. This attack could technically be executed by any of the previously listed attacker roles, but would be a lot easier for a CA administrator.

Ultimately, all of the possible attack vectors, except for stealing the CAs private key, rely on somehow exploiting the certificate issuance process. This strengthens the belief that a well designed certificate issuance process can significantly reduce a public key infrastructure's risk of corruption.

3.2 Requirements

The DFN certificate issuance process, being proven in a real-world environment, has been chosen as the model for a software enforced certificate issuance system. This section is dedicated to defining requirements for this new system with regards to the risks identified previously.

3.2.1 Functional Requirements

The set of functionality defined here is a minimal set necessary in order to build a proof of concept system. The chosen focus of the system is email certification:

- The system must enforce a certificate issuance process similar to the one of the DFN.
- The system must allow a user to obtain a certificate for their email address(es).
- The system must support look-up operations for certificates by name.

3.2.2 Non-Functional Requirements

This section is defining quality goals for the system. The following requirements are not security-critical:

- The name strings have to be unique.
- The system has to expect users to possibly have a very slow interaction/reaction time. It might take a user a full 24 hours or more to react. This entails that it cannot require (near) instant user interaction.
- Look-up operations should be executed in no worse than linear time and never take longer than one second. It is important that the system is conveniently usable.

3.2.3 Security Requirements

These non-functional requirements are defining security-related quality goals for the system. The previously discussed risks and challenges in public key infrastructure lead to some general mitigation strategies:

- **authentication** The system has to be aware of the identity of a user that is accessing it.

- **face-to-face authentication** Every user has to be authenticated in person at least once. This increases the difficulty of deceiving a CA representative.
- **authorization** Only a restricted group of users is able to authorize certificate signing requests.
- **multi-party authorization** Some processes in the system are so critical that they need to be overseen by multiple people. One example of this is the authorization of a new CA representative. This limits the damage one single representative can inflict on the system.
- **tamper-resistance** The system needs to be resistant against attacks from within and outside. All data has to be secure and immutable.
- **accountability and traceability** Every action on the system must be traceable in order to be able to hold the actor accountable. This helps identifying attacks on the system as well as errors.
- **integrity** All data needs to undergo some sanity- and plausibility-checks. This decreases the risk of a CA representative endangering the system by accident or intent.
- **availability** In order to guarantee the correct functionality of all dependent systems, the system must be available at all times.

3.3 Possible Solutions

One option for implementing this new certificate issuance system is employing a secure, central server. This server holds the access control system as well as all data. One benefit of this solution is that it allows an easy setup process, because only one server has to be configured. This server could enforce authentication, authorization and multi-party authorization. But that is where one central server is reaching its limits. Availability is going to suffer as it is only one server and every server will at least need to shut down for system maintenance once in a while. The big problem in this solution is that the central server is a single point of failure of the system. This endangers the requirements of tamper-resistance, integrity, availability and accountability and traceability.

The next logical conclusion is to employ a cluster of servers. All of them hold a copy of the data and also serve as access points to the system. This still provides authentication, authorization and multi-party authorization, but also improves availability and integrity through redundancy. While harder to guarantee, tamper-resistance as well as accountability and traceability are also achievable through a well planned, traditional system. This is due to these security requirements heavily depending on the way the cluster servers interact with each other. For instance, just replicating the new entries on

one server onto all other servers of the cluster would also lead to replicating malicious data in the case of a security breach and thus would improve integrity and availability, but only slightly improve tamper-resistance and accountability and traceability.

However, this thesis shall explore a completely new approach to designing a certificate issuance system, hoping that it will be an improvement over traditional solutions. Using a blockchain as data storage and distributed execution environment for the administration logic fulfills the security requirements of tamper-resistance, accountability and traceability and availability as a blockchain does so by design. The remaining security requirements, namely being authentication, authorization, multi-party authorization and integrity, can be enforced in the distributed execution environment provided by the blockchain.

In addition to being suitable for fulfilling the security requirements, a system like this also possesses several other beneficial properties:

- independence of secure hardware
- business processes are enforced in the fundamental logic of the system
- operation by multiple stakeholders with limited trust is possible

This raises hopes that a new certificate issuance system harnessing the capabilities of blockchain technology can be implemented successfully.

Chapter 4

Design

4.1 System Overview

The proposed system is designed in order to enforce the DFN certificate issuance process, while also minimizing the potential for human error. A general overview of the system can be seen in Figure 4.1. With the actors applicant, registration authority (RA), as well as certificate authority (CA), the system maps the basic three parties involved in the DFN certificate issuance process.

The system itself mainly consists of two parts: an application environment and a distributed ledger. This application environment acts as an interface to the distributed ledger for users and possibly also registration authorities. It facilitates access to the ledger, because it frees users from the need to own a node of the ledgers peer-to-peer network or be a registered entity on the ledger level. Especially this last point would pose another certification problem. The distributed ledger is the core of the system. It acts as distributed data storage, as well as distributed computation environment.

The data storage encompasses all data required for the certificate issuance process:

- **CSR** The *certificate signing request* contains all data required for the certificate issuance.
- **Registration Authorization** The registration authorization is an endorsement referring to a certificate signing request issued by a registration authority.
- **Certificate** The issued certificates have to be stored with a reference to the certificate signing request they fulfill. This ensures that the origin of each certificate is traceable.

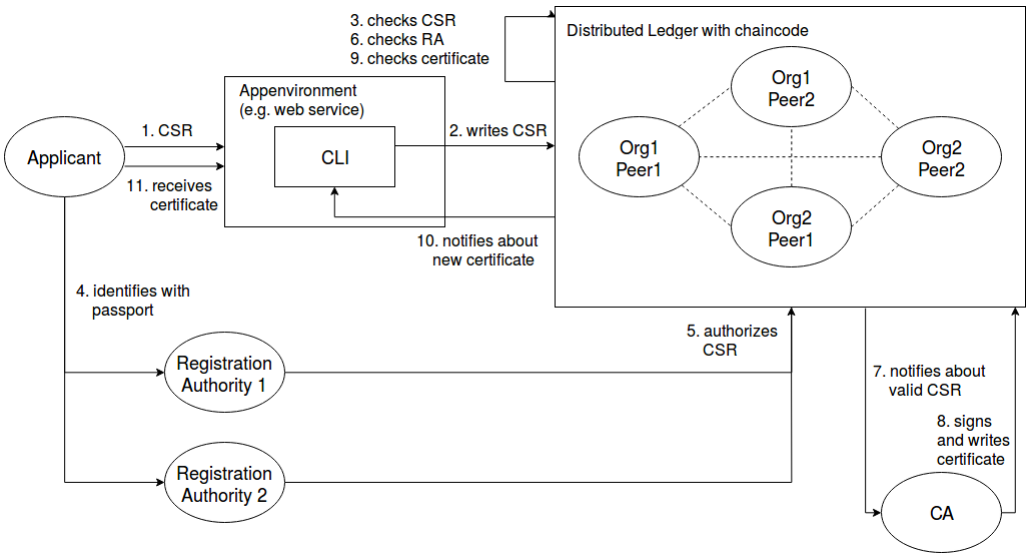

Figure 4.1: Overview of the system.

4.2 Distributed Ledger supported Certificate Issuance Process

A typical certificate issuance process using this system might look like this:

1. The Applicant creates a certificate signing request (CSR) and sends it to the application.
2. The application commits the CSR to the ledger.
3. The ledger runs some sanity checks and saves the CSR.
4. The applicant physically appears before a registration authority and identifies oneself with their passport (or comparable document).
5. After verifying the CSR, the registration authority creates a signed registration authorization and sends it to the ledger.
6. The ledger runs some sanity checks and saves the registration authorization. In this process it also flags the corresponding CSR valid. If the user is an RA himself, the CSR is only flagged valid after getting two registration authorizations from two different RAs.
7. The ledger notifies the certificate authority that there is a certificate to be issued.
8. The certificate authority creates the certificate and commits it to the ledger.
9. The ledger runs some sanity checks and writes the certificate.
10. The ledger notifies the application about the newly issued certificate.

Table 4.1: UserRecord

Name	Data Type	Description
name	string	Full, human-readable name of the user.
is_ra	boolean	A flag indicating, whether this user is a registration authority (true) or not (false).
csr_list	CSRRecord array	A list of all certificate signing requests associated with this user.
ra_list	RARecord array	A list of all registration authorizations associated with this user.
cert_list	CertRecord array	A list of all certificates associated with this user.

11. The user receives his certificate.

4.3 Chaincode Functionality

As can be seen from the previous description of the process, the ledger is required to handle some of the logic. This is done using chaincode, Fabric's powerful smart contracts (refer to section 2.2.3). The following python-like pseudo code demonstrates all required chaincode functions.

The first required function is the one that creates a new user record on the system. This `UserRecord` contains all data on a user and is further detailed in Figure 4.1. To begin with, the record is initialized with a name and a boolean indicating, whether this user is a registration authority. All other fields are empty lists that are used by other functions. This is the pseudo-code for this function:

```
function create_user(name, is_ra):
  if ledger.has_UserRecord(name):
    return error

  user_record = new UserRecord(name, is_ra, [], [], [])

  ledger.write(name,user_record)
```

The next function required in the certificate issuance process is one allowing the user to submit a certificate signing request to the system. The certificate signing request is assembled into a `CSRRecord` that is then saved into the corresponding users UserRecord.

Table 4.2: CSRRecord

Name	Data Type	Description
hash	string	This hash is used to uniquely identify the certificate signing request.
date	date	A time stamp for the creation of the certificate signing request.
csr_blob	PEM	The x.509 certificate signing request itself.
valid	boolean	An internal flag used to indicate if the certificate signing request has been approved.

Details on the CSRRecord can be taken from Figure 4.2. This is what the function looks like in pseudo-code:

```
function create_csr(csr_blob):
  # has to parse csr_blob somehow
  if !ledger.has_UserRecord(get_name_csr(csr_blob)):
    return error

  user_record = ledger.read_UserRecord(get_name_csr(csr_blob))

  date = current_date()
  csr_record = new CSRRecord(hash(date + csr_blob), date, csr_blob,
    false)
  add_csr_to_UserRecord(user_record, csr_record)

  ledger.write(user_record.name, user_record)
```

The written certificate signing request has to be read by a registration authority in order to be approved. This is implemented through the following function:

```
function query_csr(name, csr_hash):
  # checks if the referenced csr exists
  if not has_CSRRecord(name, csr_hash):
    return error

  user_record = ledger.read_UserRecord(name)
  csr_record = get_CSRRecord(user_record, csr_hash)

  return csr_record
```

The next function is used by a registration authority to approve a certificate signing request by issuing a registration authorization. The used data record is described in

Table 4.3: RARecord

Name	Data Type	Description
hash	string	This hash is used to uniquely identify the registration authorization.
csr_hash	string	The hash of the certificate signing request that is authorized.
last_passport_digits	string	The last digits of the passport number of the registrar.
ra_name	string	The human-readable name of the registration authority.
ra_signature	string	The signature by the registration authority confirming the authenticity of the registration authorization.

Figure 4.3. The authorization is validated and then written to the user record corresponding to the certificate signing request it refers to. In the end, it is checked whether the certificate signing request has enough registration authorizations to be flagged valid. This is the function in pseudo-code:

```
function create_ra(name, csr_hash, last_passport_digits, ra_name,
    ra_signature):
  # checks if the RA exists
  if !ledger.has_UserRecord(ra_name):
    return error

  # checks the signature
  if not is_ra_signature_valid(csr_hash, last_passport_digits, ra_name,
     ra_signature):
    return error

  # checks if the referenced csr exists
  if not has_CSRRecord(name, csr_hash):
    return error

  user_record = ledger.read_UserRecord(name)
  csr_record = get_CSRRecord(user_record, csr_hash)

  ra_record = new RARecord(hash(csr_hash + last_passport_digits +
    ra_name), cs_hash, last_passport_digits, ra_name, ra_signature)

  user_record = add_ra_to_UserRecord(user_record, ra_record)

```

Table 4.4: CertRecord

Name	Data Type	Description
csr_hash	string	The hash of the certificate signing request that this certificate is issued for.
cert_blob	PEM	The certificate itself.

```
  # if the user is an ra, two approvals are needed instead of the usual
    one
  # also does some sanity checks, like comparing the
    last_passport_digits field of all RARecords
  if csr_has_sufficient_ras(csr_hash, user_record)
    user_record.valid = true

  ledger.write(user_record.name, user_record)
```

This next function is used by the certificate authority in order to fulfill a valid certificate signing request by issuing a matching certificate. The record the system uses for certificate is defined in Figure 4.4. The function parses the certificate and validates it, before appending it to the user record. In pseudo-code this looks like this:

```
function create_cert(csr_hash, cert_blob):
  # needs to parse the certificate blob
  # checks if the CSR exists
  if !ledger.has_CSRRecord(get_name_cert(cert_blob)):
    return error

  user_record = ledger.read_UserRecord(get_name_cert(cert_blob))
  csr_record = get_CSRRecord(user_record, csr_hash)

  # checks if the referenced csr is flagged valid
  if not csr_record.valid:
    return error

  # does sanity checks, like checking the certificate extensions and
    issuer
  if not check_cert(cert_blob):
    return error

  cert_record = new CertRecord(csr_hash, cert_blob)

  user_record = add_cert_to_UserRecord(user_record, cert_record)
  ledger.write(user_record.name, user_record)
```

The last function left to be discussed is one allowing for the querying of issued certificates. This query can be executed by anyone and returns all certificates for a given user. It is implemented as following:

```
function query_cert(name):
  if !ledger.has_UserRecord(name):
    return error

  user_record = ledger.read_UserRecord(name)

  return user_record.cert_list
```

With this, the complete set of functionality is defined.

Chapter 5

Implementation

5.1 Technology Choices

The very first decision was to select an operating system to develop and test on. The host system chosen was `Ubuntu 16.04 LTS`, because it is a widespread Linux distribution everyone has access to.

The most important choice for the certificate issuance system is the choice of distributed ledger to base the system on. For this we have chosen Hyperledger Fabric, because it implements a distributed ledger as well as smart contracts, and thus offers everything needed for this system.

For Hyperledger Fabric, at the time of writing, there are two software development kits: one for `Javascript` and one for `Golang`. We went with the latter one, because Golang is known for its good readability. The used sdk version was `v1.0.0-alpha3` together with `Golang 1.8.6`.

The implementation also needs to create certificate signing requests and certificates. They are created using `OpenSSL 1.0.2g` on command line, because this is a reliable and easy way.

The setup uses two approaches to virtualization. One of them is employing `Vagrant 2.0.2` in conjunction with `VirtualBox 5.2.6 r120293`. This is used to containerize the host system of the implementation. The other virtualization solution is `Docker 17.05.0-ce` and is used to run peers for the Hyperledger Fabric network.

5.2 Overview

An overview of the actual system implementation can be seen in Figure 5.1. The implementation relies heavily on containerization in order to make testing and development

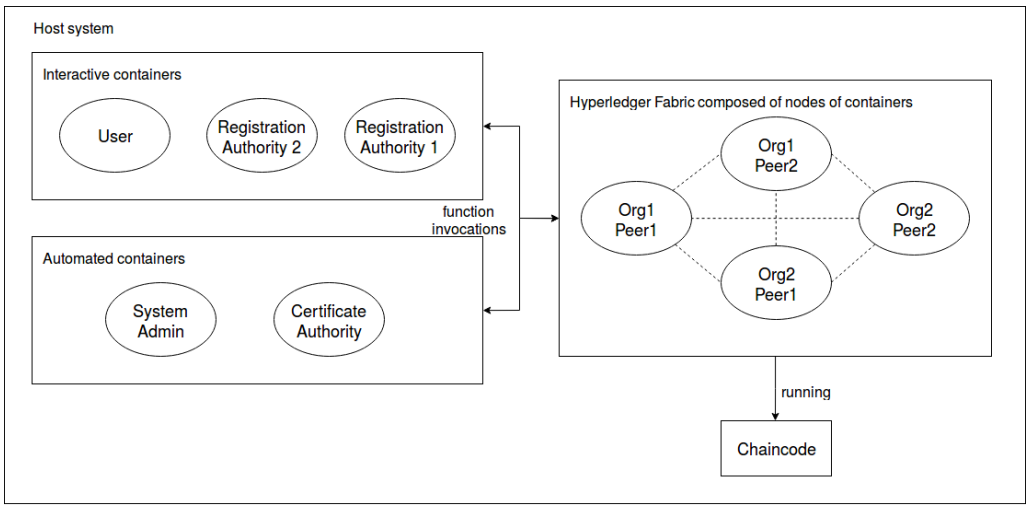

Figure 5.1: Simplified overview of the system implementation.

easier. This begins with the whole system being containerized using Vagrant in order to be independent from the physical host and its main operating system.

The main part of the system, being the distributed ledger, has been realized with Hyperledger Fabric as designed. The individual nodes making up the network are containers.

All other components of the system are split into two container groups: interactive and automated. The former one is automatically started together with the Fabric network and consists of a *system admin* and the *certificate authority*. The system admins role is to automatically sets the network up for operation. This includes creating a channel and installing the chaincode. The certificate authority connects to the network and continues by listening for an event indicating that a certificate should be issued. It then proceeds to issue it and waits for the next event.

The other group of containers, the interactive ones, can be created at will in order to execute individual steps of the certificate issuance process (e.g. creating a certificate signing request). These containers are taking the roles of a *user* and a *registration authority*.

The deviations from the original design are due to the simply not yet existing application environment, intended to be used by normal users, and the system being intended to be a proof of concept. However, the most important part of the implementation, the chaincode, is independent of the overarching network and application architecture. This makes the chaincode usable in future versions of the system.

5.3 Hyperledger Fabric Chaincode

The most important piece of this implementation is the chaincode. In this section we are going to familiarize ourselves with the actual implementation by exemplary examination of one chaincode function. The function in question is the one responsible for issuing a registration authorization. Due to the length of the code, it is presented in snippets.

Before actually looking at the function he first snippet naturally is the definition of the Function:

```
func (t *IdManagementChaincode) IssueRA(stub shim.
    ChaincodeStubInterface, args []string) pb.Response {
  ...
}
```

The function returns a peer response. This response can be a success message of type `[]byte` or an error message of type `string`. The two arguments are the interface of the ledger and an array of all chaincode invocation arguments. The interface is needed to read from and write to the ledger. The array of arguments starts with the name of the particular invoked function of the chaincode and continues on with the respective arguments. These are parsed in the beginning of the function:

```
var name string
var csrId string
var lastPassportDigits string
var raName string
var raSignature []byte

var err error

if len(args) != 6 {
  return shim.Error("Incorrect number of arguments. Expecting 6,
    function followed by a name, csrId, lastPassportDigits, raName and
    raSignature")
}

name = args[1]
csrId = args[2]
lastPassportDigits = args[3]
raName = args[4]
raSignature, err = hex.DecodeString(args[5])
if err != nil {
  return shim.Error("Failed to decode hex")
}
```

Before going any further, the functions ensures that this is not a registration authority trying to authorize their own certificate signing request. It is also ensured that the last passport digits are given:

```
// check that the user and RA are different
if name == raName {
  return shim.Error("Cannot authorize own CSR")
}

// check that lastPassportDigits is not empty
if lastPassportDigits == "" {
  return shim.Error("Last passport digits have to be given")
}
```

Next, the signature of the registration authorization has to be verified. This is done by fetching all certificates known for the registration authorization and testing all currently valid ones until one matches the signature:

```
// verify signature
// test all currently valid cert keys until one fits
verifiedSig := false
raUser, err := t.GetUserRecord(stub, raName)
if err != nil {
  return shim.Error(err.Error())
}
// check that the user is actually an RA
if !raUser.IsRA {
  return shim.Error("Signer is not an RA")
}
// test all known certificates
for i := 0; i < len(raUser.CertList); i++ {
  block, _ := pem.Decode([]byte(raUser.CertList[i].CertBlob))
  if block == nil {
    return shim.Error("Failed to decode PEM")
  }
  certificate, err := x509.ParseCertificate(block.Bytes)
  if err != nil {
    return shim.Error("Failed to parse certificate")
  }

  // check that certificate is currently valid
  currentTime := time.Now()
  if currentTime.Before(certificate.NotBefore) || currentTime.After(
    certificate.NotAfter) {
    break
  }

  pubkey := certificate.PublicKey.(*rsa.PublicKey)
  hashed := sha256.Sum256([]byte(name + csrId + lastPassportDigits +
    raName))
  err = rsa.VerifyPKCS1v15(pubkey, crypto.SHA256, hashed[:],
    raSignature)
  if err == nil {
    verifiedSig = true
    break
```

```
  }
}
if !verifiedSig {
  return shim.Error("Registration authority signature is incorrect")
}
```

After verifying the signature, the function proceeds to do some sanity checks, such as checking wether the referenced certificate signing request actually exists:

```
// get user record
userRecord, err := t.GetUserRecord(stub, name)
if err != nil {
  return shim.Error(err.Error())
}

// check if the referenced CSR exists
csrIdx := -1
for i := 0; i < len(userRecord.CsrList); i++ {
  if userRecord.CsrList[i].Id == csrId {
    csrIdx = i
    break
  }
}
if csrIdx == -1 {
  return shim.Error("Failed to find the referenced csr")
}

// stop if the CSR is already valid
if userRecord.CsrList[csrIdx].Valid {
  return shim.Error("CSR is already valid")
}

// check if the RA has been already submitted
for i := 0; i < len(userRecord.RaList); i++ {
  if userRecord.RaList[i].CsrId == csrId && userRecord.RaList[i].RaName
     == raName {
    return shim.Error("CSR has already been submitted")
  }
}
```

If no discrepancy is found, the registration authorization record is created and added to the users user record. After that the code checks if the certificate signing request has gotten enough endorsements to be flagged as valid and flags it if it has:

```
// create ra and add it to the user record
raObject := RaRecord{csrId, lastPassportDigits, raName, string(
    raSignature)}
userRecord.RaList = append(userRecord.RaList, raObject)

// flag the CSR valid if conditions are met
  if !userRecord.IsRA {
```

```
    userRecord.CsrList[csrIdx].Valid = true
  } else {
    endorsements := 0
    var lpd string
    for i := 0; i < len(userRecord.RaList); i++ {
      if userRecord.RaList[i].CsrId == csrId {
        endorsements += 1
        if lpd == "" {
          lpd = userRecord.RaList[i].LastPassportDigits
        } else if lpd != userRecord.RaList[i].LastPassportDigits {
          return shim.Error("Last passport digits do not match")
        }
      }
    }
    if endorsements >= 2 {
      userRecord.CsrList[csrIdx].Valid = true
    }
  }
```

If the certificate signing request has become valid, the certificate authority is notified. Finally, the user record is written to the ledger and the function returns an empty success message, because there is no need for any return data:

```
// notify CA
if userRecord.CsrList[csrIdx].Valid {
  if err := stub.SetEvent("ca", []byte(csrId)); err != nil {
    return shim.Error("Unable to set CC event: caEvent. Aborting
    transaction ...")
  }
}

// write user record to the ledger
err = t.SetUserRecord(stub, userRecord)
if err != nil {
  return shim.Error(err.Error())
}

return shim.Success(nil)
```

5.4 Implementation Challenges

During implementation, some challenges were encountered. The biggest one of them was caused by Hyperledger Fabric still being so early in development. This entailed frequent SDK changes that required most of of the implementation to be updated every time.

Another time sink was the discovery, that the Golang certificate package seemingly failed randomly to parse certificates created with OpenSSL. After some more investi-

gation the parsing turned out to always work when the certificate did not have any extensions. However, according to the documentation [14] parsing of extensions is possible. There was no time to conduct further tests, so the problem was circumvented by not using any extensions for the demo implementation.

5.5 Using the Implementation

This section is intended to deliver a compact guide to running the demo implementation. For exact versions of used software, please refer to section 5.1.

5.5.1 Initial Setup

Before beginning, the host system needs to have Git, Vagrant and VirtualBox installed. The setup begins with downloading Fabric and navigating to the directory containing all important files for the virtual development machine:

```
$ git clone https://github.com/hyperledger/fabric.git
$ cd fabric/devenv
```

At this point it is recommended to set the directory that Vagrant links to the virtual machine as local development directory. This directory contains the Fabric application. The following example assumes that the code is in a directory called `dev` in the users `Documents` directory:

```
$ export LOCALDEVDIR="/home/USER/Documents/dev"
```

This step might not be necessary anymore in the future. At the time of writing, the fabric environment is configured to install a version of `Golang` that is too old to run fabric SDK. The fix is to change the version installed with the `setup.sh` script to `1.8.6`.

Next, the virtual machine has to be started using the `Vagrantfile` that came with the git repository.

```
$ vagrant up
```

Once the virtual machine is up, it can be connected to:

```
$ vagrant ssh
```

First step on the virtual machine is to get the `Fabric Golang SDK` and make sure all dependencies are available.

```
$ sudo -E env "PATH=$PATH" go get -u github.com/hyperledger/fabric-sdk-go
$ cd $GOPATH/src/github.com/hyperledger/fabric-sdk-go
$ sudo -E env "PATH=$PATH" make depend-install
$ sudo -E env "PATH=$PATH" dep ensure -update
```

Now, Fabric can be built:

```
$ cd $GOPATH/src/github.com/hyperledger/fabric
$ export DEBIAN_FRONTEND="noninteractive"
$ sudo -E env "PATH=$PATH" make dist-clean all
```

At this point everything should be set to run the implementation.

5.5.2 Running the Implementation

It is assumed that the virtual machine is running and connected to. All scripts that can be used to run the implementation as a whole or in part can be found here:

```
$ cd $GOPATH/src/gitlab.lrz.de/tumi8/idmanagement/implementation/datatier
```

These scripts namely are:

- **up.sh** Starts the Hyperledger Fabric network.
- **userrecord.sh** Creates a user record for Alice.
- **csr.sh** Creates a certificate signing request for Alice.
- **ra.sh** Creates a registration authorization for the previously created certificate signing request.
- **query.sh** Queries the ledger for certificates associated with Alice.
- **down.sh** Shuts the Hyperledger Fabric network down.
- **runall.sh** Performs all of the steps above in sequence.

5.5.3 Terminating the Virtual Machine

The machine can be disconnected from and stopped with the following:

```
$ exit
$ vagrant halt
```

Chapter 6

Evaluation

The certificate issuance system has been implemented to enforce the DFN certificate issuance process leveraging distributed ledger technology. Naturally, the first step in evaluating this system would be to evaluate the distributed ledger it is built upon (i.e. Hyperledger Fabric) before continuing with the system itself. However this is far beyond possible given the time and resources for this thesis. Instead we are directly skipping to the evaluation of the system itself, assuming that the ledger is behaving according to specification and not considering any possible security flaws.

6.1 Security Evaluation

Figure 6.1 shows an attack tree for the system. It is laid out the same way the attack tree shown in Figure 3.1. The ultimate goal, to issue a malicious certificate, is the trees root and each path from a leaf to the root is one way to achieve that goal. In the following, we will discuss these paths from left to right in detail for the single attacker introduced in chapter 3. The corresponding attacker roles for the new system are:

- **User** A normal user accessing the system via the application environment.
- **Registration Authority** This is corresponding to a CA representative.
- **CA Administrator** An administrator of the CA system has full access to the data (e.g. private key), software and possibly also hardware.

First option to attack the system is being an evil registration authority, meaning the attacker needs to be an RA. This should be unlikely, because the employees appointed as registration authority are expected to be chosen carefully. Also, the fact that the system easily allows to hold the participants accountable for their actions entails that one transgression certainly means losing their job and facing criminal prosecution. This is a lot of deterrence. However, this still might not be sufficient, as a registration

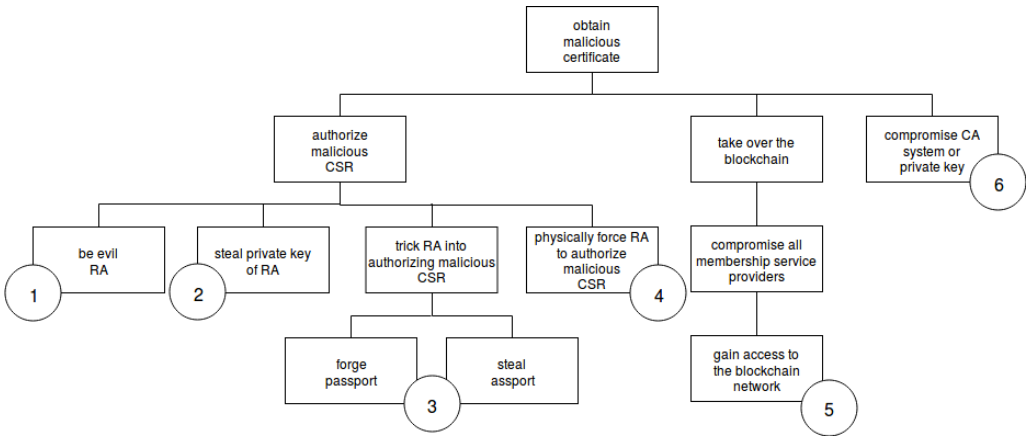

Figure 6.1: Attack tree for the certificate issuance system.

authority might try to disguise their criminal actions as a mistake. So in comparison to the original attack tree, this is no change and the attack is still possible.

The **second** path attacks by identity theft. Stealing a registration authority's private key enables an attacker to impersonate them and authorize malicious certificate signing requests. This attack could be performed by an attacker of any role, but assuming that every registration authority keeps their keys safe, is unlikely to succeed. Again, this is no change to the original attack tree.

The **third** path involves deceiving a registration authority in order to get an authorization for a malicious certificate signing request. This attacks chance of success does not depend on the role the attacker has. This requires the attacker to either forge or steal the passport of the person they are trying to impersonate, because the registration authority needs to see it. That, already posing a significant difficulty, is only half the attack. The attacker would also need to deceive the registration authority in person, meaning for instance that he would need to look like the image on the passport. While this is certainly not impossible, it is so challenging, that it can be considered unlikely. In contrast to the original attack tree, this has gotten harder. Deceiving a registration authority now requires a passport and a lot of face-to-face social engineering.

The **fourth** attack circumvents the challenges of the previous two by simply threatening the registration authority (e.g. with a weapon). This attack again is independent of the attackers role. While not particularly inventive, this approach will most likely be successful for an attacker that does not shy away from violence. Hence this is a possible attack, just as it was in the original attack tree.

The **fifth** attack vector is focused on breaking the distributed ledger (i.e. Hyperledger Fabric) itself. While it was said at the beginning of this section that Hyperledger Fabric is assumed to be flawless, this still leaves open attacks that are possible by design. With enough nodes under their control, an attacker would be able to conduct transactions at

will. The attack is still considered unlikely, because this is exactly the type of attack a (permissioned) blockchain is designed to be resistant against. Also the attacker would need access to the potentially closed network the ledger is running in. That is why this probably is the most unrealistic attack vector. This is a huge improvement over the original attack tree.

The **sixth** and last attack vector is circumventing the system by gaining access to the system of the certificate authority or stealing their private key. While this is theoretically possible for all attacker roles and very possible for a CA administrator, this only allows the attacker to issue certificates outside the system. The validity of a certificate can still be easily determined by querying the blockchain. Only valid certificates are written to the ledger together with all associated accounting information created during the issuance process.

Thanks to the distributed ledger, the system is *tamper-resistant*, provides *accountability and traceability* and enforces *multi-party authorization*. These properties lower the chance of a single registration authority inflicting damage on the system, while also ensuring traceability of attacks through complete and immutable accounting data on the ledger.

Summing up, the new certificate issuance system prevents a single attacker in the role of user or registration authority from issuing malicious certificates. Additionally, the system is highly resistant to attacks thanks to the distributed ledger. Finally, while not being able to hinder a compromised certificate authority from issuing malicious certificates, the system can assist in identifying these certificates based on the accounting data which is produced when issuing a legitimate certificate.

6.2 Additional Considerations

An important consideration for real-world deployment of the system is the trade-off between security and effort concerning multi-party authorization. All previously discussed attack paths involving a registration authority become highly unlikely with a rising number of registration authorities required for a certificate. The implementation created in this thesis only requires one registration authority for a normal certificate and two for a certificate for a registration authority. In a production use case these numbers could be increased to even further reduce attack surface.

Another consideration to be made is about the network size. A small network might not provide *non-eraseability* due to the small amount of existing data copies.

The network size, together with the number of different organizations operating the network and the complexity of the ledger's endorsement policy, is also important for the *tamper resistance* of the ledger. All of these factors determine the difficulty for

an attacker to control enough nodes to single-handedly fulfill the endorsement policy, allowing them to create any transaction they wants to. That is due to the attacker requiring full control of all those organizations membership service providers in order to create new peers for every organization that is necessary to satisfy the endorsement policy of the ledger.

Chapter 7

Related Work

7.1 PGP

Pretty Good Privacy (PGP) [15] is the other big certification (and encryption) standard besides X.509. In contrast to X.509, PGP does (normally) not form a hierarchical structure of trust, but a web of trust. Users can accumulate trust over time by being referred by other users that know them. This referral consists of signing a users key, expressing trust in the identity of a user. If one (or better, multiple) chain of trust exists from one user to another, the former can trust the latter one. PGPs main field of application is in email.

7.2 Lightweight Directory Access Protocol

The Lightweight Directory Access Protocol (LDAP) [16] is an asynchronous protocol specifying how a client interacts with directories on a server following the X.500 standard. The directories hold other directories as well as directory entries. A directory entry is the most basic type of information held by the system. Each entry consists of a set of key-value-pairs, enabling it to store arbitrary data, and is identified by a relative distinguished name. This relative distinguished name itself is a set of key-value-pairs. The concatenation of an entry's relative distinguished name and its parent's distinguished name is the entry's distinguished name and identifies the entry globally. Possible interactions with the directories include searching, adding and deleting.

LDAP is not limited to one directory server, but can instead work with a set of multiple directory servers. For instance, if the queried directory server cannot fulfill a request, it may refer to another directory server.

7.3 Certificate Transparency

Certificate Transparency [17] is an effort by Google to combat malicious certificates. For this purpose, three new components are introduced to a public key infrastructure:

- **Certificate Log** Certificate logs are publicly accessible network services storing certificates. The certificates are typically submitted by the issuing certificate authority and appended to the log. Not data is ever deleted. Anyone can query this log to find out whether the log is working correctly (via cryptographic proof) or whether a specific certificate has been logged.
- **Monitor** A Monitor is a server monitoring Certificate Logs, looking for suspicious certificates. A certificate could stand out for instance by having unusual extensions (e.g. certificate signing privilege). Upon finding something, the Monitor proceeds to notify the corresponding entity. These Monitors could be run by stakeholders, as commercial services or just privately by one person.
- **Auditor** An Auditor is a piece of software. It could for instance be a standalone service or part of a browser. It can check that a Certificate Log is working correctly and that a specific certificate is logged in a Certificate Log. The latter check is important, because every certificate has to be logged. A certificate not being logged indicates the certificate being malicious.

This system is intended to make it impossible for a new certificate to be issued without the corresponding domain owner being notified about it. It also allows anyone interested to determine whether a certificate was issued illegitimately, be it by intent or mistake. In summary it is designed to protect the public key infrastructure from being compromised and by doing so protects users from malicious certificates. In contrast to the system developed in this thesis, Certificate Transparency only starts after the certificate has been issued.

7.4 Blockchain-Based Identity Management Solutions

All of the following projects leverage blockchain technology to manage certificates or other identity data.

7.4.1 A Blockchain-Based PKI Management Framework

The paper "A Blockchain-Based PKI Management Framework" [18] presents an open-source blockchain system supporting certificate issuance, validation and revocation. This system currently runs on the Ethereum blockchain. The certificate authorities are mapped as smart contracts on the chain, keeping all relevant data to allow anyone

to validate the complete chain of trust. The system uses X.509 certificates that are extended by PKI environment information in the extension fields, such as the smart contract address of the issuing certificate authority. This infrastructure is accessible for certificate authorities and users via a Restful service.

7.4.2 The Internet Blockchain

Another effort towards blockchain-based identity/asset management worth mentioning is the "The Internet Blockchain" [19]. In the paper it is proposed to create a blockchain for Internet resource transactions. These resources for instance include IP address assignments and domain names. Administering these resources with a blockchain, instead of the currently used autonomous organizations, would eliminate traditional PKI dependency as well as roots of trust, and with that the single point of failure for all of the Internet. The biggest issue holding this concept back is scalability.

7.4.3 Blockstack

Blockstack [6] is an alternative Internet built with Blockchain. While Blockstack still uses the lower layers of the traditional Internet, it reinvents the application layer. This includes creating a new DNS and its own public key infrastructure.

Blockstack's DNS is the Blockchain Name System (BNS). This blockchain-based naming system allows for human-readable and unique names while being fully decentralized (Zooko's Triangle). The BNS also eliminates a need for a traditional PKI in the Blockstack ecosystem, because the BNS already associates public keys with domain names.

A core feature of Blockstack is that it is built to run on any underlying Blockchain and even supports changing the underlying Blockchain. Blockstack uses a Virtualchain. This Virtualchain is an arbitrary state machine running on top of a real Blockhain (e.g. Bitcoin). This abstraction provides independence from the Blockchain and allows future implementation changes and additions.

7.4.4 Hyperledger Indy

Hyperledger Indy [20] is an identity management system based on distributed ledger technology with a big focus on privacy. It is being developed by the Hyperledger community since May 2017 and is still very early in development. Indy focuses on a *self-governed identity*. Private data is never written to the ledger and only directly shared with peers upon the users consent. All identifiers used in relations (e.g. for a purchase with a shop) are *pairwise-pseudonymous identifiers*. This prevents tracking and also lessens the consequences of an identifier being compromised. The ledger only functions

as an anchor for data. This means that even though the ledger is public, the user is in full control of who can read what part of their data. Also Indy supports zero-knowledge proofs in order to minimize data disclosure.

7.4.5 Sovrin

The Sovrin Network [21] was launched in July of 2017 and is designed to be a blockchain-based, self-sovereign identity provider. Every identity is tied to a *distributed identifier* (DID) and stored in a global blockchain created for this (and only this) purpose. The blockchain framework chosen for Sovrin is Hyperledger Indy 7.4.4 which is targeted at identity management and already supports much of the needed functionality. But Sovrin does not stop there. Whenever a verifiable claim (i.e. confirmation of an identity feature, for instance age) is presented to a verifier, this has value to the owner of the claim and to the verifier. One example of this is a car rental agency (i.e. verifier) verifying that the customer (i.e. owner) has a drivers license. Sovrin introduces a token that can be exchanged for the verification of a claim and aims to build an ethical digital market place for identity.

Chapter 8

Conclusion

The problem addressed in this thesis is public key infrastructures being compromised, leading to malicious certificates being issued. This is a single point of failure for all systems relying on these certificates (e.g. the entire Internet). It was suggested that a strict certificate issuance process could limit the risk of corruption a compromised CA or CA representative poses for the public key infrastructure, while also making attacks on the system fully traceable.

The thesis successfully shows that it is possible to adapt the certificate issuance process used by the DFN as a tamper-resistant certificate issuance process based on distributed ledger technology. We began by defining requirements for this system and proceeded to show how a system like this could be designed. Finally a prototype application was developed in order to prove the system's feasibility.

The final evaluation suggests that the designed system is resistant against a single attacker, as long as that attacker does not compromise the certificate authority. The system can not protect against a compromised certificate authority's system or private key, as this allows an attacker to issue (malicious) certificates at will. However, it has been shown that the system can expose a compromised CA issuing certificates without adhering to the issuance process, due to the ledger information documenting the process that would be missing in this case.

While this thesis provides a solid foundation, there are still several aspects open for exploration in future work, such as the problem of certificate revocation. Other possible topics for future work include improving and augmenting the sanity checks automatically executed by the distributed ledger's chaincode and developing the application environment (e.g. web service) that is intended to provide access to the distributed ledger system for normal users. Yet another topic would be to modify the system in order to support multiple hierarchical certificate authorities comparable to a typical public key infrastructure. This could distribute the effort required to run the system better between multiple participating organizations.

Bibliography

[1] L. Whitney, “DigiNotar files for bankruptcy,” 2011.

[2] “Internet X.509 Public Key Infrastructure: Certificate and Certificate Revocation List (CRL) Profile,” https://tools.ietf.org/html/rfc5280, retrieved April 2018.

[3] “IBM Knowledge Center: Distinguished Names,” https://www.ibm.com/support/knowledgecenter/en/SSFKSJ_7.5.0/com.ibm.mq.sec.doc/q009860_.htm, retrieved April 2018.

[4] “DFN-AAI,” https://www.aai.dfn.de/, retrieved February 2018.

[5] S. Nakamoto, “Bitcoin: A Peer-to-Peer Electronic Cash System,” 2008.

[6] M. Ali and J. Nelson, “Blockstack: A Global Naming and Storage System Secured by Blockchains,” 2016.

[7] M. Castro and B. Liskov, “Practical Byzantine Fault Tolerance,” 1999, https://www.usenix.org/legacy/events/osdi99/full_papers/castro/castro_html/castro.html, retrieved April 2018.

[8] “Hyperledger Fabric Documentation,” http://hyperledger-fabric.readthedocs.io/en/release-1.1/index.html, retrieved April 2018.

[9] “Hyperledger Website: About,” https://www.hyperledger.org/about, retrieved April 2018.

[10] “Hyperledger Website: Fabric,” https://www.hyperledger.org/projects/fabric, retrieved April 2018.

[11] “Hyperledger Wiki: Hyperledger Fabric,” https://wiki.hyperledger.org/projects/fabric#history, retrieved April 2018.

[12] “Hyperledger Announces Production-Ready Hyperledger Fabric 1.0,” https://www.hyperledger.org/announcements/2017/07/11/hyperledger-announces-production-ready-hyperledger-fabric-1-0, retrieved April 2018.

[13] "Hyperledger Fabric Documentation: Architecture Reference," https://hyperledger-fabric.readthedocs.io/en/latest/arch-deep-dive.html, retrieved April 2018.

[14] "Golang Documentation: x509," https://golang.org/pkg/crypto/x509/, retrieved April 2018.

[15] E. Gerck, "Overview of Certification Systems: x. 509, CA, PGP and SKIP," 2000.

[16] "Lightweight Directory Access Protocol (LDAP): Technical Specification Road Map," https://tools.ietf.org/html/rfc4510, retrieved February 2018.

[17] "Certificate Transparency Website," https://www.certificate-transparency.org, retrieved April 2018.

[18] A. Yakubov, W. M. Shbair, A. Wallbom, D. Sanda, and R. State, "A Blockchain-Based PKI Management Framework," 2018.

[19] A. Hari and T. V. Lakshman, "The Internet Blockchain: A Distributed, Tamper-Resistant Transaction Framework for the Internet," 2016.

[20] "Hyperledger Welcomes Indy," https://www.hyperledger.org/blog/2017/05/02/hyperledger-welcomes-project-indy, retrieved December 2017.

[21] Sovrin Foundation, "Sovrin™: A Protocol and Token for Self Sovereign Identity and Decentralized Trust," 2018.