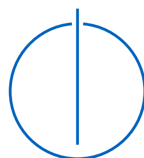
TECHNICAL UNIVERSITY OF MUNICH

DEPARTMENT OF INFORMATICS

Master's Thesis in Informatics

Exploring the Use of SSL/TLS Certificates for Identity Assertion and Verification in Ethereum

Friederike Groschupp



TECHNICAL UNIVERSITY OF MUNICH

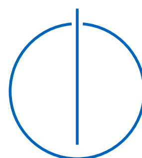
DEPARTMENT OF INFORMATICS

Master's Thesis in Informatics

Exploring the Use of SSL/TLS Certificates for Identity Assertion and Verification in Ethereum

Analyse der Nutzung von SSL/TLS Zertifikaten zur Identitätsbereitstellung und -verifikation in Ethereum

Author:	Friederike Groschupp
Supervisor:	Prof. Dr. rer. nat. Florian Matthes
Advisor:	Ulrich Gallersdörfer, M.Sc.
Submission Date:	May 15, 2020



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

Munich,

Friederike Groschupp

Acknowledgments

I would first like to thank my thesis advisor Ulrich Gallersdörfer for his support in the process of creating this thesis, for his valuable input, our productive conversations, and his constant feedback on my work. Furthermore, I would like to thank Prof. Dr. Florian Matthes for the opportunity to explore this exciting topic and the possibility to conduct this research under his supervision at the Chair of Software Engineering for Business Information Systems. I appreciate his contribution during the definition of the topic of this thesis. I would also like to thank Clemens Brunner, Joachim Neu, and Markus Sprunck for taking the time and discussing my work. Their feedback has certainly improved the outcome of this thesis. I am very grateful that we were granted research access to the extensive datasets provided by Censys.

Finally, I would like to extend my thanks to my family for their support throughout my studies. I appreciate that Anne Groschupp has always taken the time to proofread my work and that Peter Wauligmann was always there to listen when I had something to say.

Abstract

While blockchain technology promises a new era of transparent and secure distributed applications, there is a lack of an established identity management process. This poses a problem for applications requiring smart contract owners to be authenticated. One issue that previously proposed solutions face is the accumulation of a critical mass of trusted data that makes the system usable. In this work, we propose an identity assertion and verification framework for Ethereum that overcomes this bootstrapping problem. It achieves this by leveraging SSL/TLS certificates, which are part of the established infrastructure that is commonly used for authenticating internet connections.

We design and implement an SSL/TLS certificate-based authentication framework whose key features are the smart contract-based validation and storage of certificates and address-identity bindings. Looking at the current SSL/TLS ecosystem, we find that a large share of all domain certificates is issued by a small number of intermediate and root certificates. Therefore, we decide to store and maintain certificates in a central database to minimize processing costs. The evaluation of our prototype implementation shows that the associated cost of our system is within a feasible operating range, with the costs of submitting a new certificate currently averaging around 2.40 $ and the cost of creating an address-identity binding averaging around 1.30 $. The cost of verifying an address-identity binding averages around 0.08 $ or 1.02 $ depending on the deployment scheme. Our system is a pragmatic and, most importantly, quickly bootstrapped method for an identity assertion and verification framework for Ethereum.

Contents

Acknowledgments **iii**

Abstract **v**

1 Introduction **1**
1.1 Problem Statement 2
1.2 Research Questions 3
1.3 Methodology 3
1.4 Contribution 4
1.5 Structure 5

2 Background **7**
2.1 Ethereum 7
2.1.1 Blockchain Technology 7
2.1.2 Ethereum Fundamentals 8
2.1.3 Ethereum Virtual Machine 9
2.1.4 Smart Contracts 10
2.1.5 Bridging the Gap Between Off-chain Data and On-chain Computations 11
2.1.6 Ethereum Addresses 12
2.2 Authentication of Web Servers and Trust Establishment on the Internet . 13
2.2.1 X.509 Certificates and Certification 13
2.2.2 Public Key Infrastructure 17
2.2.3 CA/Browser Forum Guidelines 20
2.2.4 Certificate Validation 21

3 Analysis **25**
3.1 Use-Case Scenarios 25
3.1.1 Address-independent Payments 25
3.1.2 Authenticity of Provided Information 26
3.1.3 Automated Authentication and Authorization of Accounts 26
3.2 Stakeholders 27
3.3 Requirements Analysis 28
3.4 Survey of the TLS Certificate Landscape 29
3.4.1 Distribution of Certificate Types 30
3.4.2 Cryptographic Algorithms in Use 34

4 Related Work **37**
4.1 Blockchain-based PKI Solutions 37
4.2 Ethereum Name Service 39
4.3 did:web 40
4.4 AuthSC 40

5 Design **43**
5.1 Endorsement of Account Addresses 43
5.1.1 Using TLS Certificates for Endorsements 43
5.1.2 Endorsement Content 44
5.1.3 Internal and External Endorsements 45
5.1.4 Storage and Distribution of Endorsements 46
5.2 Enabling On-chain Decisions 48
5.2.1 Using Oracle Services Versus Migrating the PKI On-chain 49
5.2.2 Central Certificate Database 52
5.3 Evaluation of the Design 53

6 Implementation **55**
6.1 Prototype Structure 55
6.2 Certificate Framework 57
6.2.1 Certificate Database 57
6.2.2 Certificate Parsing and Validation 59
6.3 Endorsement Framework 64
6.3.1 External Endorsement Database 64
6.3.2 Internal Endorsement Contract 67

7 Evaluation **71**
7.1 Compatibility 71
7.2 Costs and Performance 72
7.3 Security Considerations 75
7.3.1 Security of the Certificate and Endorsement Frameworks 75
7.3.2 Security of the TLS Ecosystem 76
7.3.3 Mapping Domain Names to Real-world Identities 77
7.4 Research Questions 78

8 Conclusion **81**

List of Figures **83**

List of Tables **85**

Bibliography **87**

1 Introduction

Trust is the foundation for interacting with others and being sure about the identity of other parties is the foundation for trust. This is also true for blockchain-based systems, where many envisioned applications require the ability to link the human-unfriendly byte strings that are used as account addresses to real world entities and vice versa. When sending funds to an account, you want to be sure that the receiver is who you intend and that the funds do not get lost. When you are validating diploma information you need to be certain that the information you are validating against is actually provided by the respective institution. Further, to buy an airplane ticket through a blockchain application, you want to be absolutely positive that the seller is the actual airline.

Currently, address information is exchanged through off-chain mechanisms that do not involve the blockchain. Most commonly, addresses are announced on websites. The user then copies the address and uses it as the transaction receiver to perform the desired action, for example transferring a certain amount of funds to it. The problem with this approach is, that the user has to assert the authenticity and integrity of the address and has to transfer it to the blockchain without errors. There are no on-chain means to verify the correctness of the address and that it is really controlled by the designated entity. This does not only facilitate errors made by the users when copying the address, but also permits exploits where an attacker deludes the user into using the attacker's address. Additionally, this approach only allows reactive authentication: the user knows the identity of the target account, but cannot authenticate itself actively to it.

These problems show the need for an on-chain authentication framework that enables the assertion and verification of identities. However, many different solutions with high entry costs and weak bootstrapping strategies have led to a slow adoption of identity management for blockchains. We believe that for the successful adoption of a solution it must provide effortless bootstrapping without the need of building new infrastructure and it must not rely on networking effects, which means that it must be feasible even with a very small number of people participating.

Therefore, we propose to leverage the infrastructure of a system that has been successfully in place for many years now to solve a very similar problem: the SSL/TLS public key infrastructure. The SSL/TLS public key infrastructure is used to distribute trusted certificates that map fully qualified domain names to public keys. Web servers can then use these certificates to authenticate themselves to users and users can be certain

that they are communicating with the right server. In this work, we investigate the potential benefits of using SSL/TLS certificates for an identity assertion and verification framework for Ethereum, the currently most widely used general-purpose blockchain. The great strength of the approach that we propose is that it does not rely on network effects and does not require a bootstrapping phase, making it ready-to-use for any domain or certificate owner.

1.1 Problem Statement

With our proposed system, we aim to increase the trust in information and services provided on Ethereum by providing an on-chain authentication framework that overcomes the bootstrapping problem of naming services. In general, it is difficult to bootstrap such systems due to a simple reason: before a system does not comprise a critical mass of trusted information, people are not incentivised to participate, in turn slowing the accumulation of trusted information. By using SSL/TLS certificates, we leverage the fact that people are used to mapping real-world identities to domain names, already know a large number of domains that they trust, and that a well-established system for the authentication of these domain names exists. Our approach is promising as the required infrastructure already exists and as it allows individuals to join independently. However, we expect some problems with using this already existing system:

- **Different perceptions of truth.** The decision whether a certificate or signature is accepted as valid depends on different factors, such as the root store of the verifying party and the point of time of validation. Routines for Ethereum, however, must be deterministic so that all parties computing them always come to the same result.

- **New use case.** The SSL/TLS certificate system was not designed with our use case in mind and evolved according to the requirements that come with the authentication of web domains. Therefore, we have to study the SSL/TLS certificate ecosystem carefully to adapt it for our use case.

- **Limited participation.** Leveraging an already open system entails that entities can perform exactly the actions that they already can. While anybody can act as verifier in a SSL/TLS certificate-based identity management system, only owners of certificates can prove their identity. In our case, this limits the set of identities that can be linked to an Ethereum address to web domains with an existing SSL/TLS certificate.

In this work, we focus on solving the first two problems by developing a strategy that allows us to integrate authentication based on SSL/TLS certificates to Ethereum applications.

1.2 Research Questions

The goal of this work is to explore the opportunities and limitations of using SSL/TLS certificates for address identification on Ethereum. The final artifact should be an implemented prototype that allows on-chain decisions, is open to as many entities as possible, and does not require any actions by stakeholders other than the certificate owner. To guide our process, we pose the following research questions:

1. **How can we enable on-chain decisions on identity using SSL/TLS certificates?**

   The ability to verify SSL/TLS certificates and associated identities deterministically on-chain is the fundamental building block of our proposed system. To find a fitting solution we need to answer the following subquestions:

   a) What are possibilities to provide determinism for the validity decision?

   b) What are the associated costs of the approaches?

   c) How can certificates be revoked on-chain?

   d) What are inherent problems of the SSL/TLS public key infrastructure and how can we mitigate them?

2. **How can we use SSL/TLS certificates to endorse Ethereum addresses on-chain?**

   Once a certificate is considered as valid on-chain, the corresponding private key can be used to endorse an Ethereum address and bind it to the identity described by the certificate. We aim to provide a functional, secure, and cost-efficient endorsement scheme. To find an answer to the question of how this is implemented best, we need to consider the following:

   a) How can already deployed contracts and externally owned accounts be endorsed?

   b) How can identity endorsements be revoked?

   c) What measures can an identity owner take to increase trust in their identity claim?

1.3 Methodology

The work presented in this thesis consists of the following phases:

1. **Background and literature research.** Our work is heavily involved with two flourishing ecosystems, blockchain-based systems and the SSL/TLS public key infrastructure. As our goal is to build a usable system, we need to understand the theoretical foundation and current practices of both areas. Therefore, we perform

thorough research on these topics. Additionally, we read and evaluate literature that deals with questions similar to ours.

2. **Requirements Analysis** Concurrently to background and literature research, we define the requirements for our system. Some requirements stem from the foundational goal of this thesis and influence the literature search, while the literature search leads to the definition of other requirements. We furthermore discuss and validate the requirements with domain experts.

3. **System Design.** Based on the functional and nonfunctional requirements, we develop a method that allows the on-chain binding of web domains to Ethereum addresses. One main consideration is the validation of SSL/TLS certificates. We propose different solutions and discuss their advantages and disadvantages. Moreover, possible structures and policies for endorsement creation and validation are discussed.

4. **Prototype implementation.** The final system design is realized in a proof-of-concept prototype. We present and discuss technical key details, such as data formats and performing cryptographic computations on Ethereum.

5. **Evaluation.** In this phase, we evaluate the advantages and disadvantages of our system design and implementation. We examine the prototype in regards to technical characteristics, such as the execution costs of different actions. We discuss the implementation's security properties. Additionally, we model the bootstrapping of our system with certificates and show that the entry cost for identity owners and the recurring costs for verifiers are low.

1.4 Contribution

The main contribution of this work is the design and prototype implementation of an authentication framework for Ethereum that allows binding account addresses to identities. These should be meaningful for human use. The key differentiator between our and previous work is that we leverage an already fully functional authentication system – the SSL/TLS ecosystem – with its established mapping of names to real-world identities. This eliminates the bootstrapping problem.

Concretely, we contribute

- a cost-efficient system that enables the validation, storage, and maintenance of SSL/TLS certificates on-chain.
- a comprehensive framework for the binding of Ethereum addresses to real-world identities.
- a prototype library for the parsing and validation of SSL/TLS certificates in Solidity.

1.5 Structure

The remainder of this thesis is structured as follows: We summarize relevant background information about Ethereum and the SSL/TLS ecosystem in chapter 2. In chapter 3, we motivate the need for our system with use cases, describe its stakeholders and formulate the requirements for our system. Furthermore, we survey the state of SSL/TLS certificates to adapt the design and implementation of our system to our findings. In chapter 4, we present and discuss research and initiatives that combine blockchain technology and certificates or aim to provide identity solutions for blockchains. We discuss our design choices in chapter 5 before walking the reader through the key elements of our implementation in chapter 6. Finally, we evaluate our system in chapter 7 in regards to cost, functionality, and security. We revise the results of our work and give pointers for future work in chapter 8.

2 Background

In this work, we leverage the existing infrastructure used for identity assertion and trust management in the internet to provide a mechanism that binds Ethereum addresses to real-world identities. To follow this work, it is important to have a fundamental understanding of Ethereum and blockchain-based systems in general, as well as of the SSL/TLS public key infrastructure. This chapter introduces the most relevant concepts of blockchain technology and Ethereum in section 2.1 and the SSL/TLS ecosystem in section 2.2, while defining terms that we use throughout this work and emphasizing properties that we rely on during the design and implementation of our system.

2.1 Ethereum

Ethereum[1] is a public permissionless blockchain that was introduced in 2014 [47]. In contrast to the original Bitcoin blockchain[2], Ethereum does not only track the ownership of coins, but is designed as a general-purpose blockchain. Sections 2.1.1 and 2.1.2 introduce the basic functionality of blockchain technology and Ethereum. We present the programming environment of Ethereum, namely the Ethereum Virtual Machine and smart contracts in sections 2.1.3 and 2.1.4. Sections 2.1.5 and 2.1.6 discuss the difficulties with running programs on-chain that rely on real-world data and with identity management in Ethereum, respectively.

It is important to notice that, while this work focuses on Ethereum, the concepts described and results acquired can be transferred to every blockchain that supports Turing-complete programs.

2.1.1 Blockchain Technology

The concept of blockchains was first introduced in 2008 by Satoshi Nakamoto with Bitcoin, a peer-to-peer version of electronic cash [35]. The proposed blockchain technology has three fundamental properties:

- **No trust required**: The participants in the network do not need to trust each other and no intermediate trusted third party is required. Trust is solely based on cryptographic primitives and consensus protocols.

[1] https://ethereum.org/, accessed 13.05.2020
[2] https://bitcoin.org/, accessed 13.05.2020

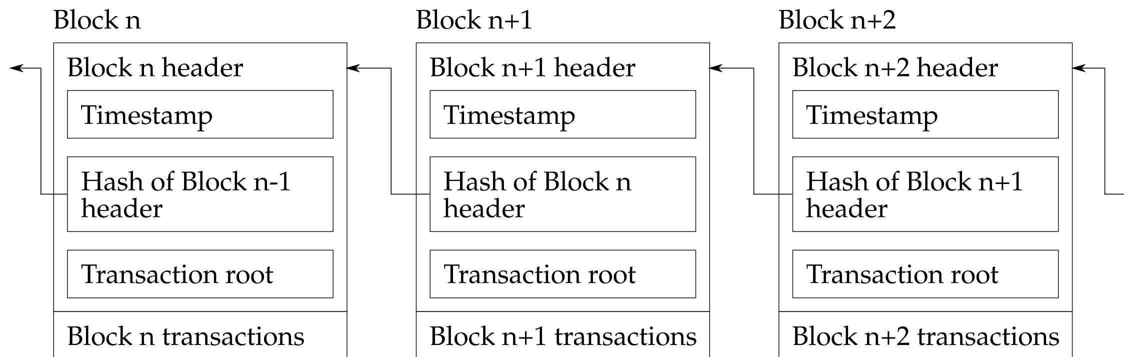


Figure 2.1: The basic structure of a blockchain.

- **Immutability**: Once an action is recorded on the blockchain, it cannot be reversed, reordered, or modified.
- **Transparency**: Every participant has access to all information stored on the chain. While this enables them to validate all actions and the current state, it poses a challenge to the secrecy of sensitive data and the privacy of individuals.

The properties are provided by the design of blockchain as a distributed ledger that stores a sorted list of published *transactions*. In Bitcoin, transactions are used for the transfer of ownership of coins. Transactions are accumulated and stored in blocks. These blocks are then chained together by storing the cryptographic hash of each block header in its succeeding block. This basic structure of a blockchain is depicted in Figure 2.1. The attempt to modify the content in any of the blocks would lead to a different hash value. All participants of the network agree on the most recent block, in Figure 2.1 block $n + 2$, and its hash value through a consensus mechanism in certain time intervals. This makes the blockchain immutable and tamper-proof, as the discrepancy between values can easily be detected.

Anyone can participate in a public permissionless blockchain by creating an *account*. The user creates a public key pair; the public key determines the account address. The account can then be used to receive funds or trigger transactions. To prove that a transaction is authorized, the user signs the transaction with the private key of the account. The user sends the signed transaction to the peer-to-peer network, where miners verify it and put it into a block which is appended to the blockchain. Transactions change the of state of the blockchain.

2.1.2 Ethereum Fundamentals

Ethereum builds on blockchain technology to construct a “world computer” [6]. The concept of a world computer implies that all users share one computer together with its resources and memory, which holds both data and code. In contrast to usual general-

purpose computers the state of the machine is not governed by only one but many entities and the state is distributed globally. Each participant can issue transactions that alter the state of the computer. The Ethereum blockchain is used to track this state over time. The part of Ethereum that handles the state, computations, and state transitions is called "Ethereum Virtual Machine" (EVM). As all computations need to be performed by the participants of the peer-to-peer network, issuing transactions does cost money. Transaction costs are paid in *ether*, the built-in currency of Ethereum.

A world computer allows developers to implement decentralized applications, so-called Dapps. A Dapp is an application that serves some specific purpose but does not rely on the presence or existence of a particular party. Dapps are usually not only comprised of smart contracts (cf. section 2.1.4) for the decentralization of the controlling logic, but also of other aspects of the application such as storage or naming [6]. A Dapp based on a blockchain comes with several advantages over an application that is centralized [6]:

- **Resiliency**: As the code and information required to execute the Dapp is stored on the blockchain, it is available as long as the underlying blockchain is operating. In contrast, a centralized application usually depends on the availability of specific servers.
- **Transparency**: Every participant can inspect the Dapp code to verify its purpose and functionality and monitor all its interactions, as the Dapp is both stored and executed on the blockchain.
- **Censorship resistance**: Once a Dapp is deployed on the network, its code cannot be altered due to the immutability of the blockchain. This allows participants to interact with the Dapp without interference by any entity, as long as they have access to the blockchain.

2.1.3 Ethereum Virtual Machine

The Ethereum Virtual Machine (EVM) handles the deployment and execution of code on the blockchain. The EVM is a quasi-Turing complete state machine, only limited by the amount of computational steps that are allowed. The number of allowed steps is limited to guarantee that a program execution halts as there is no external force that could stop a non-halting computation. The EVM state is comprised of all accounts and the persistent storage.

Ethereum differentiates two different kind of accounts: *Externally owned accounts* (EOAs) and *smart contracts*. An EOA is an account that is created as described in subsection 2.1.1 and therefore controlled by a private key. Any entity that knows the private key has the ability to send ether or invoke smart contract executions from the EOA. An EOA does not contain any code. A *smart contract* is a piece of immutable code that can be executed in the context of the EVM. More detailed information on smart contracts is

given in subsection 2.1.4. All accounts are identified by a unique 20-byte address. For EOAs, the address is derived from its public key. The address of a smart contract is computed based on the account that creates the contract and a nonce. In both cases, the resulting address cannot be forced to take a certain value or pattern with a reasonable amount of effort.

A transaction is a digitally signed data packet used to transfer coins to another account or invoke smart contract execution. As creating transactions requires a private key for the signature, transactions can only originate from EOAs. The recipient of a transaction can either be an EOA or a contract. Each transaction holds information about the amount of ether transferred, the maximum number of computational steps allowed, and the fee that the sender is willing to pay for each computational step. Additionally, an optional data field can be used to invoke smart contract functions. When a contract needs to send funds to another account or invoke a method of another smart contract, it sends a so-called *message*. While messages have a similar structure to transactions, they are not signed and are not explicitly stored on the chain. A contract can never send a message on its own, it can only be invoked by a transaction.

As hinted above, the issuer of a transaction or message needs to pay for every computational step that is made. This is because every step needs to be performed by verifying parties and every transaction increases the size of the blockchain. Smart contracts are represented in EVM byte code, which is similar to assembly. Each opcode costs a predefined fee, measured in gas. The cost of a transaction is determined by the amount of gas that was used multiplied by the gas price that the sender was willing to pay. Additionally, the sender of a transaction specifies how much gas they are willing to pay at most for a transaction. If the transaction requires less gas, the remaining gas is refunded; if it requires more, the transaction fails and all changes made by it are reverted. All transactions are atomic, regardless of how many messages and contract calls they entail.

2.1.4 Smart Contracts

Smart contracts (SCs) are immutable deterministic programs - sets of variables and functions together with persistent storage - that can be executed in the context of the EVM. As smart contracts are accounts, they can receive, store, and send Ether. In addition to that, any other account that knows a smart contract's address and its specification can call its public functions. The creator of a smart contract does not have any privileges at the EVM protocol level. However, the application layer logic can be implemented to support concepts like the ownership of contracts.

The EVM works with smart contracts that are represented in EVM bytecode. For more convenient development, smart contracts are usually written in high-level languages such as Solidity or Vyper and then compiled to EVM bytecode. To deploy a smart contract, a transaction is sent to the special contract creation address 0x0. This transaction carries

the bytecode of the contract in its data field. Subsequently, the contract is assigned a deterministically computed address. From this point on, all accounts can invoke the public methods of the smart contract.

Once a smart contract is deployed on the chain, it is only controlled by its code. As the code is stored on the chain, it is immutable – smart contracts cannot be updated or patched, even by the contract creator. The only modification that is possible is the removal of the contract from the chain, but only if the `selfdestruct()` method is callable.

That smart contracts are accessible to anyone who knows the contract address and public functions requires special attention on security during the development process. This is even aggravated by the fact that even if a vulnerability is detected, it cannot be patched due to the immutability of the smart contract. The immutability also poses another problem: code cannot be updated to comply with new requirements. If functionality is to be added or modifications need to be made due to external factors, a completely new smart contract needs to be deployed. Additionally, as every computing step requires spending some ether, it is important to avoid expensive computations and storing large amounts of data on the chain in order to keep a Dapp attractive to the user.

2.1.5 Bridging the Gap Between Off-chain Data and On-chain Computations

One important aspect of blockchains is that in order to reach consensus about the state of the chain, the execution of all transactions must yield the same result for every participant. This means that all computations must be completely deterministic. To achieve this, all computations must be conducted based only on the shared context of the blockchain and external data can only be introduced to the chain as data payload in transactions. Therefore, there are no mechanisms in Ethereum that allow a contract to establish connections to the outside world to inquire about varying information such as currency exchange rates, weather information or the validity of certificates. If a contract relies on such external data, it must be provided by a trusted party that has access to this information and stores it on the blockchain for the contract to use.

The parties that bridge the gap between the off-chain world and the blockchain context are referred to as oracles. In general, an off-chain oracle service gathers information, for example from a web service, validates, and processes it. It provides this information to its corresponding oracle contract on the chain by issuing a transaction. When contracts need the data for their computations, they can then read the information from the oracle contract.

The main issue with oracles is that the on-chain contract has no means to validate the information provided. It has to use the information trusting the authenticity and the integrity of the oracle service. To mitigate this trust issue, several approaches have been proposed. One of them is to provide cryptographic proofs with information. This

approach is implemented by the Provable[3] blockchain oracle. It relies on TLSNotary[4] proofs, which attest that certain HTTPS traffic has occurred between a client a server, as well as the content of the server response. Another approach uses decentralized oracles to avoid having a single point of failure: information is provided by n independent entities. The information is considered valid if at least m out of the n claims match. This type of oracle is often used for computation oracles when expensive computations that can theoretically performed on-chain are moved off-chain to save costs. Especially when using computation oracles, one has to make careful considerations on the trade-off between saving money and security.

2.1.6 Ethereum Addresses

On the Ethereum protocol level, account addresses determine identity. Ethereum addresses are unique hexadecimal numbers with a length of 20 bytes. For EOAs, an address comprises the last 20 bytes of the Keccak-256 hash of the account's public key. The properties of Keccak-256 as a cryptographic hash imply that the creator of an account cannot influence the resulting address by other means than brute force. The result is that Ethereum addresses appear as randomly generated and are very human-unfriendly.

As the value of Ethereum addresses cannot be controlled, they are not bound in any way to the real-world identity of their owner which requires that addresses need to be distributed and exchanged off-chain. This comes with a risk as it opens the door for man-in-the-middle attacks [22]. For example, the Coindash hack that took place in 2017 was based on such an attack [14]. Coindash hosted an initial coin offering (ICO), where it offered users to exchange ether against tokens for their Dapp. Coindash announced the ICO and the account address that funds should be sent to on their website. However, shortly after the ICO started, an attacker was able to replace Coindash's account address with an account address owned by themselves. Users aiming to obtain tokens subsequently transferred the necessary funds to the attacker's account instead of the legitimate Coindash account.

In addition, Ethereum addresses do not, unlike Bitcoin addresses, natively contain a checksum that protect from typing an erroneous address. EIP-55 suggests backward-compatible checksums, but its implementation is not mandatory and might not be supported by all wallet applications [6][8]. If ether are sent to a mistyped address, they can usually not be recovered, as the accounts private key cannot be computed. At the time of design, checksums for Ethereum addresses were not included, as the idea was that address would be hidden behind abstractions at higher layers [6]. However, the development and adoption of these higher layers have only advanced slowly in reality.

[3]https://provable.xyz/, accessed 13.02.2020
[4]https://tlsnotary.org/, accessed 13.02.2020

The properties of Ethereum addresses highlighted in this section urgently demand for mechanism that binds Ethereum addresses to real-world identities in a way that does not require off-chain address distribution and can be bootstrapped quickly.

2.2 Authentication of Web Servers and Trust Establishment on the Internet

Trust, which [29] defines as "ability of two entities to believe one another at some level, so that they can interact in a secure manner", is a fundamental requirement in human interactions while not trivial for web-based interactions where real-world identities are easily faked. On the internet, users usually connect to web servers based on the server's domain name. The user associates the domain name with a real world identity, for example their bank, which it deems the operator of the server. This association generates the user's trust in the services and information that the responding server provides. However, the user must ensure that the responding web server is actually genuine and belongs to the owner of the requested domain. The web server needs to authenticate itself.

In today's internet, the authentication of servers and the establishment of secure connections is commonly performed with the SSL/TLS[5] protocol. The authentication of the communication partners with TLS is based on X.509 certificates and a public key infrastructure (PKI). In this section, we address the technical and organizational aspects of the TLS PKI that are relevant to our work. In subsection 2.2.1, we describe X.509 certificates, certificate validation, and certificate revocation. We introduce the basic concepts, security flaws and mitigation attempts of the PKI in subsection 2.2.2. We give an overview of the guidelines for good practice defined by the CA/Browser-forum in subsection 2.2.3. Finally, we outline the criteria a verifier uses to decide on the validity of certificates in subsection 2.2.4.

2.2.1 X.509 Certificates and Certification

Certification is the process in which one entity, the issuer, asserts the identity of another entity, the subject, and endorses the assignment of a public key to the subject with a digital signature. A subject may, for example, be a person, organization, or also a web server. The information the issuer has gathered about the subject, the subject's public key, and the digital signature are compiled in a document called *certificate*. This certificate is passed to the user. Subsequently, the subject can use the certificate to attest their identity to another party, the *verifier*, by proofing that they own the private key that corresponds to the subject's public key.

[5]TLS (Transport Layer Security) is the successor of SSL (Secure Socket Layer). While TLS is the technically correct term to describe the protocol currently in use, SSL is still commonly used. Henceforth, we use the term TLS.

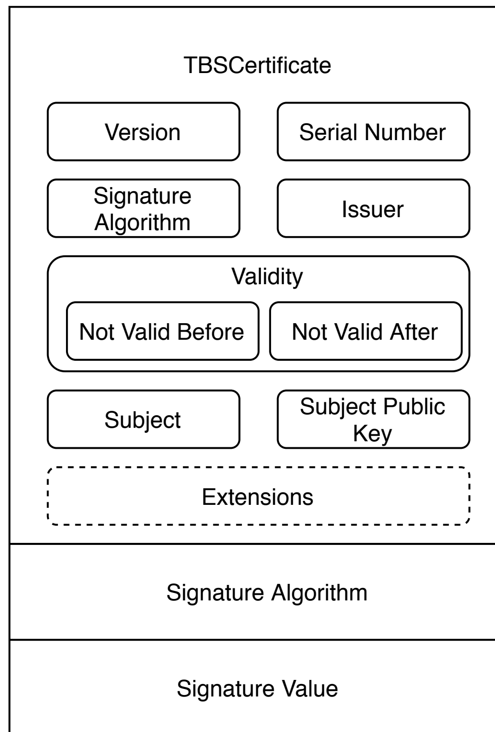


Figure 2.2: Structure of an X.509 certificate.

The certificate format that is commonly used today is specified by the X.509 standard, which is defined as part of the X.500 authentication framework. The current version X.509 v3 is specified in RFC 5280 [16]. In addition to the fundamental components, X.509 certificates contain information about its validity period, the cryptographic algorithms used, and information about the issuer.

The structure of X.509 v3 certificates is displayed schematically in Figure 2.2. A X.509 certificate consists of three main parts: The TBSCertificate (TBS is short for to-be-signed), the signature algorithm identifier, and the signature. The TBSCertificate contains the information associated with the subject and the certificate. Its fields are explained below in detail. The signature algorithm field contains the object identifier (OID) of the cryptographic algorithm that was used to create the signature. The signature field contains the digital signature of the DER-encoded TBSCertificate that was created with the public key of the issuer.

The fields of the TBSCertificate are, in order:

Version Format of the encoded certificate, for version 3 the value is 2.

Serial number Positive integer that is unique for every certificate issued by one issuer, maximum size of 160 bit.

Signature algorithm The OID of the cryptographic algorithm and parameters used to

create the signature. Must match the signature algorithm field after the TBSCertificate.

Issuer Name of the issuer. Must be of the X.501 type *Name*.

Validity Time interval, specified by two dates, during which the certificate is valid. For this time, the issuer warrants that they will maintain information about the certificate.

Subject Uniquely identifying name of the subject. Must be of the X.501 type *Name*.

Subject public key Info Public key of the subject together with the cryptographic algorithm the key is to be used with.

Extensions Optional fields that carry additional information about the subject, issuer, or certificate.

The length of the validity period is a critical design parameter. A short validity time means that a certificate needs to be replaced often, incurring higher costs. A long validity time increases the chance that the cryptographic algorithms in use are broken or that a certificate might be compromised, and prolongs the time during which the certificate can be misused. Therefore, the validity period has to be defined carefully and based on the type of certificate, its intended application, and the strength of the subject key.

Commonly used extensions are *key usage* and *extended key usage*, which define the operations that the subject private key is allowed to be used for. The *basic constraints* extension describes if and how the subject may act as an issuer for other certificates. The *subject alternative name* extension allows additional identities to be bound to the subject, such as additional domain names or an e-mail address. More extensions not relevant for this work are defined in RFC 5280 [16].

X.509 is a specification that is application agnostic. A TLS-certificate is a X.509 certificate where the *Extended Key Usage* extension field allows server authentication. With the TLS protocol and TLS certificates, one can ensure that the genuine server responds and create a shared secret without previous knowledge of each other over an unencrypted channel. When authentic certificates are in use, malicious actors cannot intercept or interfere with the messages transmitted over the network. As TLS is the most common application of X.509 certificates, the terms TLS certificate and X.509 certificate are often used interchangeably. In the rest of this work, we leverage the existing, commonly used and trusted infrastructure of TLS certificates and refer to them plainly as certificates unless a clear distinction needs to be made.

Certificate Revocation

Sometimes, it is necessary to invalidate a certificate before its validity period expires. This can, for example, be necessary when the subject's private key has been compromised

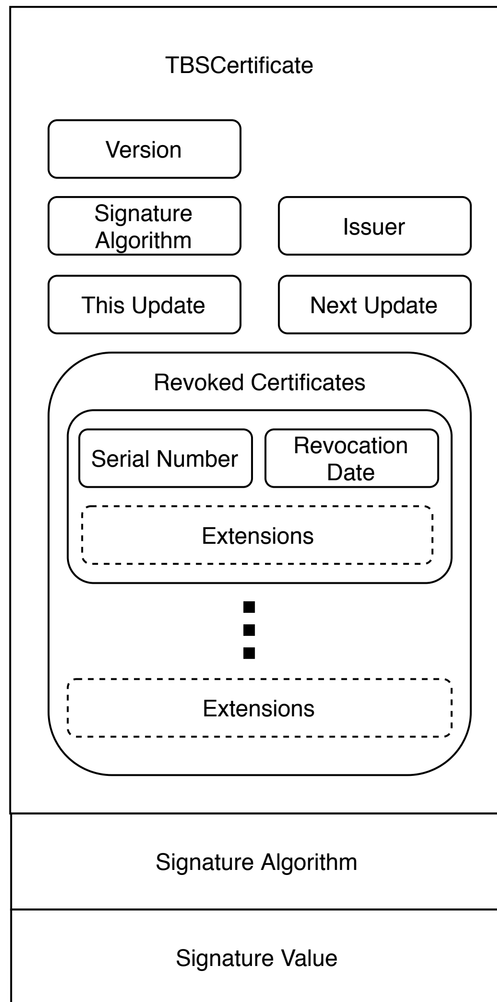


Figure 2.3: Structure of an X.509 certificate revocation list.

or lost. Another instance is when an issuer is discovered to have issued fraudulent certificates, may it be due to being compromised, careless certification procedures, or simply due to acting with a malicious intent [49]. Fraudulent or compromised certificates can have a substantial effect on security, so it is important that trust is maintained by invalidating such certificates [13]. As the certificate persists as valid document, the issuer needs to announce that a certain certificate cannot be trusted anymore. This is called *certificate revocation*.

Certificate Revocation Lists (CRLs) [16] are the initial approach for certificate revocation that was introduced with the X.509 specification. With this approach, issuers maintain a list of the certificates they issued and that needed to be revoked. The structure of X.509 CRL is displayed in Figure 2.3. Certificates contained in CRLs are identified by their serial number. During validation, the verifier inquires the CRL from the issuer. The

verifier considers a certificate revoked and consequently invalid when it is contained in the list.

The main disadvantage of CRLs is their lack of scalability. While the approach might work well for a small number of certificates, the size of a large CRL puts strains on the network through which the CRL file is distributed, on the validation process as the whole list needs to be iterated, and on the memory of the verifier if they keep a copy of the file. To reduce the load on the issuers, CRLs are often updated periodically, for example on a weekly basis, and stored locally by verifiers for this time period. This delays the announcement of the revocation information and opens a window during which the issuer already knows that the certificate is revoked, but the certificate is still evaluated as valid by verifiers.

To remedy these problems, the Online Certificate Status Protocol (OCSP) [41] was proposed. When validating a certificate, the verifier sends an OCSP request about the status of the certificate to the issuer. The OCSP response contains the digitally signed status information of the certificate. The status of a certificate can be either *good*, *revoked*, or *unknown*. In addition to the certificate status, the response contains information about the point of time when the response was created, how long the OCSP response is valid, as well as other optional information in extension fields. The authenticity and integrity of the response is evaluated by checking the digital signature of the response with the private key of the issuer.

While this solution is more scalable than CRLs, it poses a privacy risk as an attacker might observe the verifier's request and can infer that they want to interact with a certain web server. Additionally, this approach only works when the issuer is online. As some applications evaluate a certificate as valid when they do not receive an OCSP response, this puts issuers under the threat of Denial-of-Service attacks. OCSP stapling [38], a variation of OCSP where the certificate subject requests an OCSP response for its certificate and provides it together with the certificate, tries to mitigate these problems. However, as the OCSP response provided is valid for a period of time and only updated on certain time intervals, the freshness of the revocation assertion is problematic again.

2.2.2 Public Key Infrastructure

Certificates act as a recommendation made by the issuer to trust the authenticity of a public key. Recommendations are a vital part for identity management systems as it is impossible to know all parties one has to interact with beforehand [31]. However, these recommendations must still be propagated and public keys must be administered and distributed in a way that allows verifiers to rely on their authenticity. Certificate systems rely on various organizations, policies, and processes for the vetting of identities, and circulation of certificates [25].

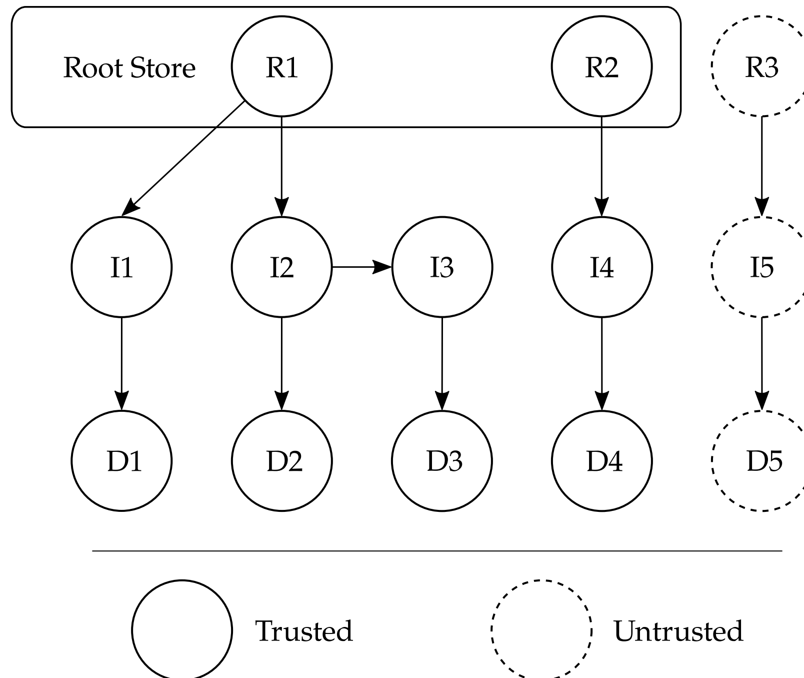


Figure 2.4: Exemplary structure of a hierarchical X.509 PKI.

Public key infrastructure (PKI) is the term that describes systems for the generation, administration, and distribution of public key certificates [18]. PKIs build on the concept of multiple recommendations stringed together: The subject of one certificate can use its private key to sign another certificate. A verifier can trace back the chain of signatures until they encounter an issuer that they know and trust. This issuer acts as a *trust anchor* for the recommendation chain. There exist two types of PKI: In web-of-trust-like PKIs participants are peers and every participant may endorse certificates. This gives users complete control over who they trust, but requires them to proactively assess whom they trust [29]. In contrast, hierarchical PKIs rely on chains of trust. In this model, only *certificate authorities* (CAs) can create and sign new certificates; each certificate can only be signed by exactly one entity.

The X.509 PKI is a hierarchical PKI. The trust anchors are so-called *root certificates*. Root certificates are usually self-signed certificates in which the subject matches the issuer. Self-signed certificates can only be validated against themselves, therefore, verifiers store the set of root certificates that they trust in a root store. For TLS, root stores are usually distributed together with an operating system or browser. Root certificates are used to sign *intermediate certificates* that act as issuer for other intermediate certificates or end certificates.

Certificates that can be used to sign new certificates are called *CA certificates*. To create a CA certificate, the issuer of a certificate can endorse the certificate owner as trusted CA. In X.509, this is indicated by the presence of the *basic constraints* extension with the

CA-flag set to true and the presence of the *key usage* extension with the *keyCertSign*-bit set. In the X.509 PKI, the set of CA certificates is comprised of root certificates and subordinate certificates. Subordinate certificates act as delegate of a root certificate to sign other certificates. We refer to subordinate certificates as *intermediate certificates*. Leaf certificates that are not CA certificates are used for authenticating servers and domains. We refer to them as *domain* or *server certificate*.

In Figure 2.4 an exemplary hierarchical PKI is depicted. Trusted certificates are represented by circles with drawn-through lines, untrusted certificates by dashed lines. An arrow from a certificate A to a certificate B indicates that B was signed by A. In the example, the verifying application's root store does contain the root certificates R1 and R2, but not R3. Therefore, the verifier does only accept the domain certificates D1 to D4, but not D5. Commonly paths in the chains of trust are of length three, as they consist of exactly one root, one intermediate, and one domain certificate. Some cases exist where multiple intermediate certificates are part of the path. In Figure 2.4, this is the case for domain certificate D3.

Problems of the X.509 PKI

In the model of hierarchical PKIs, the whole trust is based on the integrity and benign nature of CAs [27]. In reality, both the quality of intra-CA validation processes and security measures are difficult to assess from the outside [25], which shows in numerous wrongdoings by CAs in the past, may they be either mistakenly or maliciously. As one compromised CA can, at least for some time period, weaken system-wide security [13], such incidents are regarded with great concern.

The compromise of the DigiNotar CA was one of the most notable incidents and had an immense effect on measures taken to ensure integrity of the X.509 PKI [5]. In 2011, DigiNotar issued, unauthorized by Google, a fraudulent certificate for `google.com` and all of its subdomains. Subsequently, the certificate was presented to Iranian users to perform man-in-the middle attacks on them [32]. The underlying flaw that facilitates such an attack is simple: While a domain owner can mandate one CA to issue certificates for their domain, they cannot keep other CAs from issuing fraudulent certificates.

To empower both domain owners and users to monitor CA behavior, different approaches have been proposed. One of them is the attribute-based detection of fraudulent certificates, based on the assumption that certificates for the same domain are issued with the same or similar attributes [9], [44]. For example, if a new certificate for a domain is issued by a CA in a unrelated country, the probability that it is fraudulent is considered higher. While some of these approaches show a good detection rate of maliciously issued certificates, a high number of benign certificates are flagged as fraudulent [3]. A different proposal is the establishment of centralized certificate databases. When a user is presented a certificate by a server, they can ensure that this is the same certificate other users have seen. This idea is implemented for example by the

ICSI Certificate Notary [4]. Another approach is Certification Authority Authorization (CAA) [23], where DNS records limit which CAs are allowed to issue a certificate for a domain. These records can be checked by CAs during the issuing process and by clients during certificate validation. A first study on CAA showed that adoption is slow and error-prone [42].

Certificate Transparency

The most promising and now widely adopted approach is Certificate Transparency (CT) [10]. Google proposed CT in 2013 as a reaction to the DigiNotar compromise. Since 2017, CT is mandatory for all CAs that issue certificates for web domains. The goal of CT is to make it possible to identify mistakenly or maliciously issued certificates and detect CAs that exhibit fraudulent behavior.

CAs conforming to CT present the content of every certificate that they are about to issue, a so-called precertificate, to several public append-only logs. The presented information is stored in the logs, in return, the CA receives a proof of inclusion. This proof is added to the content of the certificate, which is then signed. When a verifier validates the certificate, they check whether the certificate contains a proof of inclusion from a log that they trust, indicating that the creation of the certificate has been made public. It is important to note that precertificates are marked with a so-called *CT poison*. The CT poison indicates that the precertificate must not be used for authentication purposes [28]. Precertificates may only be used for the submission to CT logs. For authentication, a variant of the certificate that does not contain the CT poison must be presented.

CT is a reactive approach; it does not prevent the issuance of fraudulent certificates. Instead, it enables domain owners to monitor the public logs and detect when a unsolicited certificate is issued. In such a case, the responsible CA and their root certificates are considered compromised or unreliable and would be removed from root stores.

2.2.3 CA/Browser Forum Guidelines

The CA/Browser Forum (CAB) is a coalition of CAs and browsers for a uniform definition of technical and organizational certificate issuance standards. Their guidelines are the de facto standard for the TLS certificate ecosystem, as it comprises the four main root store curators - Mozilla, Apple, Google, and Microsoft.

The forum defines *baseline requirements*[6], that CAs and certificates that they issue have to comply with. The requirements deal with technical details, such as which cryptographic algorithms are allowed together with the minimum key length that is to be used. For example, new certificates with signatures based on the SHA-1 hash are no longer

[6] https://cabforum.org/baseline-requirements/

accepted. They are also concerned with certificate policies, such as the maximum validity period. Server certificates may only be valid for up to 825 days. Additionally, no certificates for special-use domains, such as "localhost", may be issued. Since 2017, all certificates complying with the CAB requirements need to be included in multiple CT logs and contain the respective proofs.

The forum's baseline requirements demand CAs to verify *all* subject information contained in a certificate. As certificates are used in scenarios that have varying security requirements, CAB defines different levels of certificates. In general, when a certificate needs to be more asserting, more information needs to be verified by the CA more diligently, resulting in a higher price for the certificate. The different certificate types are, from least asserting to most asserting:

Domain Validation The CA verifies that the entity requesting the certificate controls the domain through a challenge-response protocol. The signed certificate does only contain the domain name in the fields "commonName" and "subjectAltNames". The certificate does not contain any further information about the subject. This is enough for the use of TLS, as the public key can be used to secretly negotiate connection keys, but does not provide authenticity about the real-world identity of the domain owner. With the Automatic Certificate Management Environment protocol, domain validation certificates can be generated automatically without the need for validation by a human.

Organization Validation This type of certificate contains the organization name as subject information. In addition to domain validation, the existence of the organization is verified, and that the certificate sign request originates from this organization.

Extended Validation (EV) Additional guidelines that require the validation of the legal and physical existence of the subject. The certificate ensures the the visited website is controlled by a specific legal entity identified by name, address, jurisdiction of registration, and registration number. In some browsers, this certificate is displayed as being especially trustworthy.

The distinction of different certificates made the adoption of TLS certificates thrive. Websites can now offer web traffic encryption due to the low- to zero-cost of automatically issued certificates. Applications that need a higher level of trust, such as bank web sites, can generate it by presenting more valuable certificates.

2.2.4 Certificate Validation

So far, we have covered three aspects of trust [13]: Trust anchoring, deciding who is trusted to act as CA; transitivity of trust through certification and the issuance of certificates; and the maintenance of trust through the revocation of certificates. These aspects concern CAs and the PKI. The last aspect involves the user and how they interact

with certificate information. Users need to interpret this information to decide whether (i) they accept the certificate as valid and (ii) they trust the certificate.

For certificate validation, the verifier receives the entire chain of certificates from the domain certificate to the root certificate. The validity of the certificate depends on formal factors that can be verified in conformance with strict rules defined by the X.509 specification [16]. To be considered valid, a certificate has to fulfill the following points:

- Intact chain of trust: The verifier follows the chain of trust from the server certificate to the root certificate. Each certificate must be validated with a correct signature produced by its predecessor. As trust anchor, the root certificate must be contained in the verifier's root store. Additionally, all certificates in the chain must be valid as well, i.e. conform with the following requirements.
- Integrity: By verifying the signature with the issuers public key, the verifier can ensure that the TBSCertificate has not been tampered with and that the information contained is genuine.
- Expiration: The certificates validity period must include the time of validation[7].
- Subject information: The information identifying the subject must match the claimed identity. For server certificates, this is the fully qualified domain name. For intermediate certificates, the subject must equal the issuer of its child certificate.
- Constraint processing: An issuer can impose constraints on the actions that the subject public key might be used for. This includes the information whether the subject is trusted to act as issuer for other certificates. The root and intermediate certificates must be CA certificates. The verifier has to ensure that the allowed key usage of the web server corresponds to the application scenario.
- Revocation status (optional): The certificate must not be revoked by the issuer. This step is not mandatory, but highly recommended.

For the certificate to have an actual meaning, the subject needs to prove the possession of the subject private key. Otherwise, anyone could claim an identity by requesting the certificate from a server and subsequently presenting it as their own. How this prove is performed depends on the protocol that is used. In TLS, the subject either signs a challenge or decrypts a message provided by the verifier.

That a certificate is valid according to the X.509 specification does not necessarily mean that the verifier trusts the certificate and accepts the subject as authenticated. This decision depends on the implementation of the verifier application, additional

[7] This holds for TLS certificates and cases where certificates are used for authentication. When certificates are used for endorsing the authenticity of a digital signature, different policies may apply. For example, the German Digital Signature Act considers a signature valid if the signature was created while the certificate was valid. The validity status of the certificate at the time of validation does not matter.

verifier-side policies and the authentication scenarios. For example, all major browsers and operating systems reject certificates that do not comply with the CAB baseline requirements. This means that certificates that use outdated cryptographic algorithms and do not include certificate transparency proofs are rejected. In addition, the verifier might decide to impose further restrictions, for example to only trust EV certificates or chains of trust that do not exceed a certain path length.

This chapter introduces the basic terms and concepts that we rely on for the remainder of this work. Of particular interest are the properties of Ethereum smart contracts (section 2.1.4) and Ethereum's (lacking) identity management (section 2.1.6). For the design of our system, we refer back to methods of certificate revocation (section 2.2). For the implementation the properties of X.509 certificate validation are strongly relevant.

3 Analysis

In this chapter, we motivate the need for our system and describe and analyze the ecosystem that we are working in. For this purpose, we describe use-case scenarios for our system in section 3.1 before describing the involved stakeholders in section 3.2. Based on this information, we derive the requirements for our system in section 3.3.

3.1 Use-Case Scenarios

Binding Ethereum account addresses to real-world identities and offering an on-chain identity assertion and verification system allows users to interact with other accounts without the need that the users manually verify and maintain Ethereum addresses. Below, we highlight in three exemplary use cases when and how users can profit of an authentication framework for Ethereum.

3.1.1 Address-independent Payments

To transfer Ethereum funds from Alice's to Bob's account, Alice needs to provide Bob's account address as an input to the transaction. As the design of Ethereum does not incorporate any identity management, Bob needs to communicate his address to Alice through an off-chain channel. This could, for example, be achieved by meeting Alice in person and physically exchanging the address, by announcing the address on a web site, or by storing the address-identity mapping in a trusted database. In all scenarios, Alice needs to (i) obtain the address, (ii) carefully copy the address, and (iii) provide the address as an input to the transfer transaction.

With an identity management system incorporated in Ethereum, this process could be made more convenient and secure by storing address-identity mappings on-chain. Instead of obtaining the address off-chain, one could query the identity information stored on-chain and use this address for the transaction without the need to manually copy it. For example, when shopping at Bob's website "www.bobs-shop.com", Alice would instruct a designated payment smart contract to transfer 1.5 ether to the account associated with the identity "www.bobs-shop.com". The contract searches for the correct address and transfers the funds on behalf of Alice. During the process, Alice never comes in contact with the Ethereum address itself.

3.1.2 Authenticity of Provided Information

With it's properties of being tamper-proof and being always online, blockchain technology is a suitable and convenient option for endorsing the authenticity of information. Consider the process of verifying the validity of university diplomas and credentials: Alice applies for a position in Bob's company and supplies the PDF of the university degree that she obtained from Carol's university. As the PDF might be manipulated, Bob wants to ensure its integrity by inquiring about the document's validity and status at Carol's university. Traditional methods, like calling or writing the document's issuer, are labor intensive and come with a processing delay. Additionally, they depend on the availability and continued existence of the institution.

With a solution based on Ethereum, Carol would create a smart contract and maintain a list that contains the cryptographic hashes of all diplomas issued by her university and a list of revoked diplomas. Subsequently, Alice could provide the smart contract address together with the PDF to Bob. Bob can hash the certificate, and verify that the contract contains the document in the list of issued credentials and not in the list of revoked credentials. However, there is still one issue: Bob needs to ensure that the information was actually provided by Carol. Carol can give the information she provides authenticity by signing the contract address and storing the signature in the contract, thereby binding the contract to her identity and warranting the credibility of the information stored. Bob can retrieve this signature and verify that it was indeed created with the private key of the TLS certificate for "www.carols-university.edu". In case the website is offline or the institution does no longer exist, the corresponding TLS certificate should be obtainable on-chain.

3.1.3 Automated Authentication and Authorization of Accounts

In some cases, the operator of a smart contract might want to restrict the access to the service they offer. The authorization could be based on different criteria such as a client's country, the client's status as a registered organization or business, or a set of specifically whitelisted entities. Without a system for identity assertion and verification for Ethereum, approaches for general criteria would place the burden of verifying identities on the smart contract operator, while the whitelisting approach would require manual labor to match identities to account addresses and is infeasible for a large amount of authorized accounts.

An automated system, where smart contracts can authenticate and authorize other Ethereum accounts without external intervention, would work in the following way: Contract A wants to use the service provided by contract B. The operator of contract B wants to offer its service only to a defined set of entities, for example owners of web domains ending on ".gov". Instead of whitelisting individual account addresses, the owner would whitelist identities, i.e. domain names. This way, the operator does not need to keep track of the accounts that authorized entities own. Instead, the owner

of contract A endorses the address with its private key and stores the signature in the contract, linking the contract to themselves. When contract A requests a service from contract B, contract B can retrieve the identity information, verify it with the on-chain identity assertion system, and authorize it based on policies defined by the owner of B.

3.2 Stakeholders

In this section, we list the stakeholders of an identity management system for Ethereum, define their roles and assign terms that we use throughout this work.

Certificate issuers are, as described in section 2.2.1, entities that are responsible for checking identities and creating corresponding certificates. In addition, they maintain information about the revocation status of the certificate during its validity period. We also refer to this role as *issuer*.

Certificate subjects are entities whose identity is described in a certificate. In our system, the type of an identity is a web domain name as used in TLS certificates. The *subject* is the entity that owns the certificate and the private key and operates the web domain's server. The entity taking the role of the subject is normally also the entity taking the role of address owner and endorsement creator.

Address owners either control the private key of an EOA or have control over a smart contract on application level. This stakeholder is also referred to as *account owner* and, when the account in question is a smart contract, *smart contract operator*.

Endorsement creator is the term used to describe an entity that creates a cryptographic binding between an Ethereum address and a web domain name. In order to create an endorsement, one should be an address owner and must be in possession of a valid certificate and its private key. This role is also referred to as *endorser*.

Verifiers are parties that are interested in knowing the identity of an account owner. For this purpose, they retrieve and validate the certificate pointed to by the endorser, validate the endorsement, and make a decision whether they believe the identity claim based on their policies.

Trusted third parties (TTPs) are entities that are trusted by at least one of the other stakeholders to perform certain actions or attest certain information. For example, a TTP can perform validation of endorsements on behalf of a verifier or attest the point of time at which an endorsement was created.

In general, stakeholders can be divided into three categories: Certificate issuers, identity owners, and verifiers. Identity owner comprises the roles of certificate subject, address owner, and endorsement creator. In a benign setting, these roles are always taken by the same entity, or at least by entities from the same organizational unit. Conceptually,

trusted third parties are verifiers that do not validate claims for their own purpose, but as a proxy on behalf of other verifiers.

3.3 Requirements Analysis

In this section, we derive, explain, and discuss the requirements for our targeted system. These requirements are the foundation for the design (chapter 5) and implementation (chapter 6) of our system. The final requirements are summarized at the end of this section.

As pointed out in our problem statement, there is a gap between the determinism required for routines running on Ethereum and the subjective perception of truth concerning X.509 certificate validation. For an Ethereum authentication framework, it is of utmost importance that it supports on-chain decisions about identities and trust (R1). This means that our system needs to provide all information that is required for deciding on the validity of a certificate, while the verifier needs to define policies and make them available. Another essential property for an identity system is that an identity can only be claimed by the real identity owner. For our system this means that (i) an endorsement must be produced with the secret private key of a certificate and that (ii) it must not be possible to reuse an endorsement or modify it in a way that allows linking the identity to another address. We demand this property in requirement R2. Furthermore, our system must enable the issuer of an endorsement to revoke the endorsement independently from the certificate or other endorsements (R3).

Another crucial design decision when building an identity system is whether there is one maintained state of truth or different perceptions of truth are possible. The first scenario requires for some sort of consensus – may it be by having a designated authority or through democratic processes – to decide on the set of trust anchors. With the second scenario, anyone would be to add trust anchors and decide on their set of trusted roots. As we leverage the TLS ecosystem, we want to adhere to its design principle and require open participation (R4), i.e. enable the second scenario. The final essential requirement for our system is availability (R5): The successful operation of our system must not rely on infrastructure external to the Ethereum context. The system should be self-sufficient and shall not rely on the availability and state of external servers or information sources.

Finally, we define four additional requirements with the purpose to make the system attractive, usable, and versatile that should be implemented in a way that does not contradict requirements R1 to R5. Requirement R6 demands compatibility: Numerous smart contracts have already been deployed and do not support the functionality of our system. However, it should still be possible to endorse them on-chain. In addition, the system that we implement should simply be an enabler for other applications. Consequently, the design of our system should be flexible (R7), i.e. should not mandate

exactly on which criteria trust decisions are made and allow constructs such as multiple endorsements for one address. One of the driving motivations of our work is to provide a practical identity system for Ethereum that allows fast adoption (R8). This entails two things: Infrastructure that already exists should be used to minimize bootstrapping and actors wishing to participate should not rely on third parties. In particular, this means that owners of domain certificates should be able to participate without requiring action from certificate issuers. Lastly, we have to consider the on-chain costs that are associated with our system. Considering that the validation of endorsements in combination with the validation of certificates are the most commonly performed actions, the system should be cost-optimized for them (R9).

In summary, the requirements for our system are:

R1 Support of on-chain decisions

R2 Unambiguous endorsements

R3 Individual revocation of endorsements

R4 Open participation

R5 Availability

R6 Compatibility

R7 Flexible design

R8 Enable independent and fast adoption

R9 Cost-efficiency for the verification of endorsements

3.4 Survey of the TLS Certificate Landscape

With our system, we leverage the existing infrastructure of the TLS ecosystem. We introduce the theoretical background on this topic in section 2.2. In this section, we present numbers that describe the current state of the ecosystem in practice. This includes the cryptographic algorithms in use and the relationships between CA and domain certificates.

For our analysis, we use data provided by Censys [17]. Censys performs internet scans and monitors certificate transparency logs to compile the encountered TLS certificates in data sets. These data sets also include certificates that are invalid, expired, or not trusted by common root stores. However, for a certificate to be relevant for our analysis, it needs to be

- commonly trusted. For this criterion, certificates must have a valid signature and a valid path to a root certificate part of a commonly trusted root store. We use

the Mozilla NSS root store[1] as a reference point. The information about which certificates are in the NSS root store are provided directly by Censys.

- not expired.
- not a precertificate. Certificates that are submitted to certificate transparency logs contain the critical "CT poison" field. This field indicates that the certificate must not be used for authentication (cf. section 2.2.2).

In the following, we will only consider the set of certificates with these properties. Furthermore, as the data set is dynamic with new certificates being added, we only consider certificates added to the data set before a cut-off date and expiring after the cut-off date. For the data sets in section 3.4.1 the cut-off date is the 21st of April 2020 at 23:59:59 GMT; for the data set in section 3.4.2 the date is 27th of April 2020 at 23:59:59 GMT.

3.4.1 Distribution of Certificate Types

In total, 204,166,070 certificates fulfill the described requirements on the 21st of April. This includes root, intermediate, and domain certificates. We refer to this set as *set 1*. As this set of certificates also contains a lot of certificates that are not intended for general use in the world wide web but for more ubiquitous applications, we also consider a second data set: The certificates of websites that were in the Alexa list[2] of the top million websites by visits on the 21st of April and fulfill the same requirements as above. For this data set, we identify 418,956 unique domain certificates. This number is much smaller than one million due to several reasons: Different (sub-)domains sharing one certificate, web sites serving expired certificates, web sites serving certificates that are not trusted, or web sites not serving certificates at all. We refer to this set as *set 2*.

Out of the 204,166,070 certificates in set 1, 3,345 are CA certificates, 204,162,724 are domain certificates, and one certificate is of version X.509 v1 and does therefore not include this information. We define a level-x certificate as a certificate where the shortest trusted path to the root certificate contains x certificates. In set 1, 153 certificates are level-1 certificates (meaning that 153 certificates are in the root store), 2,387 are level-2 certificates, 203,838,127 are level-3-certificates, 325,196 are level-4 certificates, and a negligible number of 207 are level-5 certificates. This means that the most common structure for chains of trust is "domain certificate – intermediate certificate – root certificate". There are no certificates that are level 6 or higher.

The numbers above show that it can be expected that each CA certificate is responsible for issuing and maintaining a significant amount of certificates. To find out whether there are differences regarding the number of certificates depending on one CA certificate or

[1] https://www.mozilla.org/en-US/about/governance/policies/security-group/certs/, accessed 09/05/2020

[2] https://www.alexa.com/topsites, accessed 09/05/2020

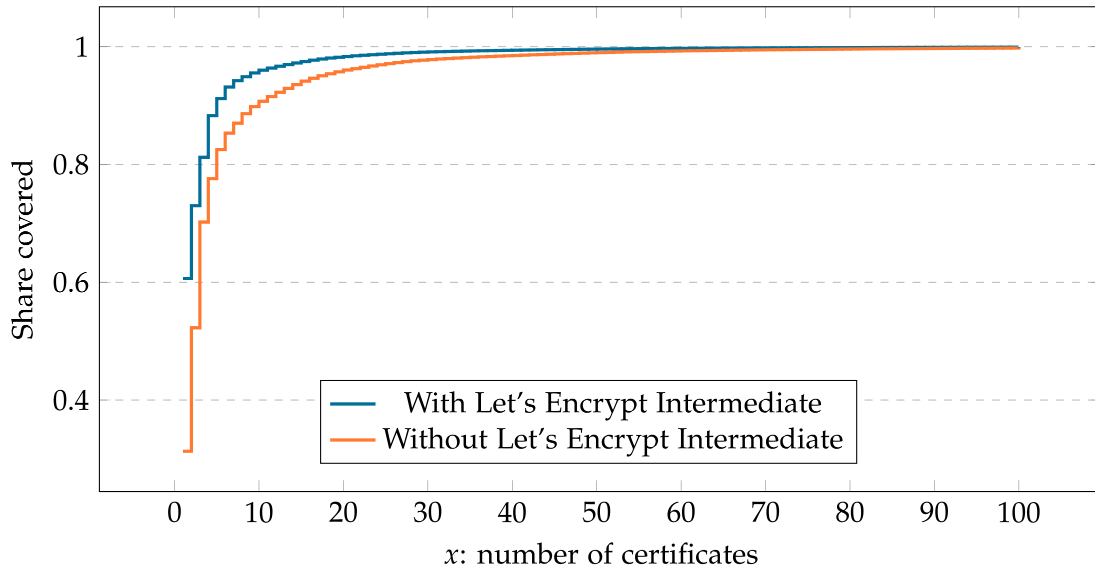


Figure 3.1: Maximum share of certificates covered by aggregation of the top x intermediate certificates. Graph is truncated after $x = 100$.

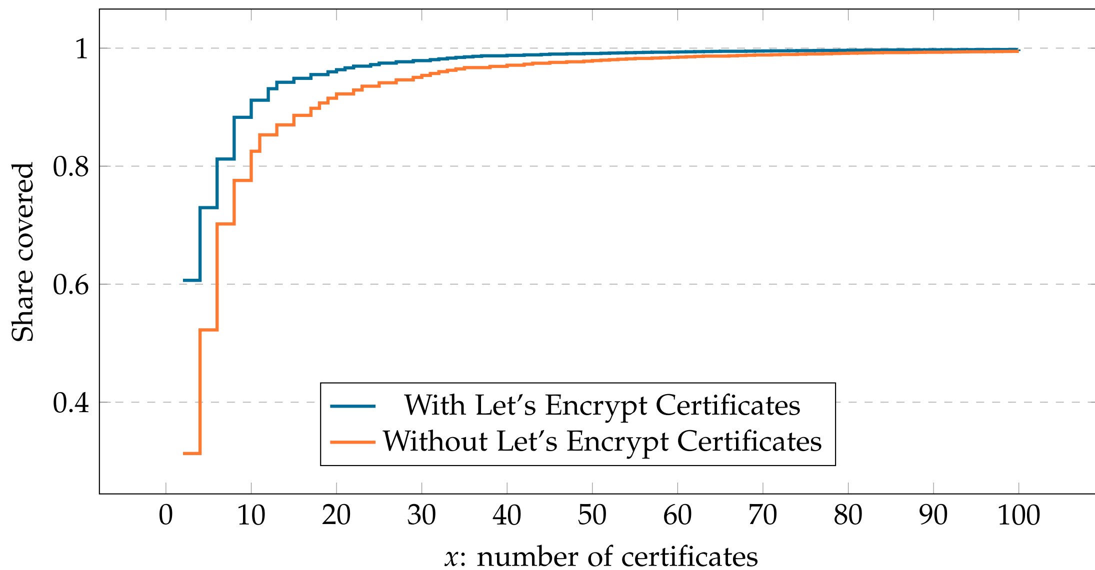


Figure 3.2: Tight lower bound on achievable share of certificate domain when choosing x CA certificates. Graph is truncated after $x = 100$.

whether numbers are distributed equally, we examine set 1 in a bottom-up approach: We group domain certificates by their issuer and count the number of certificates in each group. It is important to note that an issuer is determined by the issuer name and the identifier of its public key, not by its certificate fingerprint[3]. From the cardinality of the groups, we can derive the number of intermediate and ultimately the number of root certificates required to cover a certain percentage of domain certificates.

At first, we take a look at intermediate certificates that issue domain certificates and order them by how many valid certificates they issued. The by far most prevalent issuer of domain certificates is "Let's Encrypt Authority X3", the currently active intermediate for Let's Encrypt with 123,826,849 issued certificates, a share of over 60%. The top five intermediates together cover over 91% of domain certificates, eight intermediates are required for 95% and 26 for 99%. These numbers are visualized in Figure 3.1. We also consider these numbers without the Let's Encrypt intermediate, which might be relevant for applications in which the no-cost automated issuance of certificates provided by Let's Encrypt is not trustworthy enough. Then, the top issuer is "CloudFlare Inc ECC-CA-2" with a share of over 30%. The top five intermediates cover over 83%, 16 intermediates are required for 95% and 43 for 99%.

Of course, these numbers do not represent the total numbers of CA certificates required to cover the domain certificates as we must take root certificates and, in case of chains containing more than three certificates, additional intermediate certificates in account. A first look at the data shows that root certificates do not scale quite as well as the intermediate certificates: The top six intermediate certificates are all signed by unique roots. This means that in total, $2 \times 6 = 12$ CA certificates are required to cover 93% of certificates. Calculating the total numbers of certificates to cover certain shares is complex due to the convoluted nature of the TLS ecosystem: features like cross-signing make the bottom-up approach infeasible and the top-down approach is obstructed by the fact that some of the top roots maintain only one intermediate certificate, while others maintain dozens. Therefore, the numbers presented in Figure 3.2 are a tight lower bound. Concretely, this means that by our heuristics, we chose x CA certificates and were able to cover the certificate chain for a certain share of domain certificates with it. It might be possible to pick a more optimal combination of CA certificates and cover a larger share. For example, the share of 98% of certificates can be covered by at most 37 CA certificates, divided in 24 level-2 (intermediate) certificates and 13 level-1 (root) certificates. In a situation without Let's Encrypt, 35 CA certificates (23 intermediate, 12 root) cover over 96% percent of domain certificates.

Finally, to get a well-rounded perspective on the distribution of certificates, we take a look at data set 2. Domain certificates in the Alexa list are covered by just 256

[3]The only information about its issuer that a certificate contains is the issuer name, not the issuer's certificate fingerprint. Additionally, it is important with which issuer key the certificate was signed, as issuers may own several different keys. The certificate can be validated with any certificate that contains the right issuer name and public key.

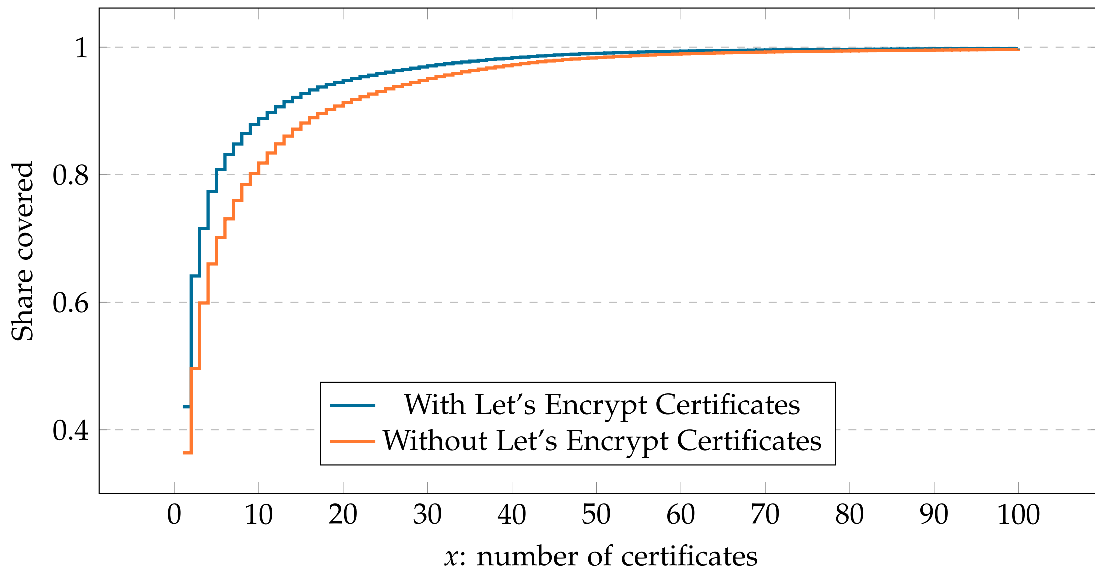


Figure 3.3: Maximum share of certificates of domains in the Alexa list covered by aggregation of the top x intermediate certificates. Graph is truncated after $x = 100$.

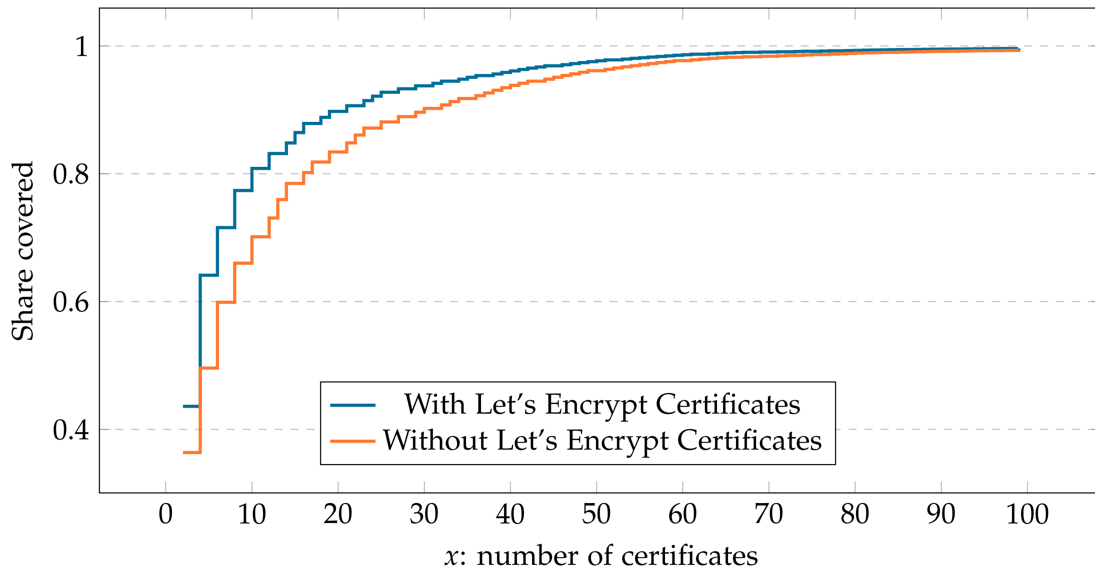


Figure 3.4: Tight lower bound on achievable share of covered certificates of domains in the Alexa list when choosing x CA certificates. Graph is truncated after $x = 100$.

intermediate certificates, with "Let's Encrypt Intermediate X3" again taking a lead with covering 43%. However, domain certificates seem to be distributed slightly more evenly across intermediate certificates, as 21 intermediates are required for covering 95% and 49 are required for 99%. Without the Let's Encrypt certificate, 30 and 69 are required, respectively. These numbers are presented in Figure 3.3.

When taking root intermediates in account, the numbers in set 2 scale worse that in set 1, as the graphs in Figure 3.4 show. To reach a coverage of 95%, 35 CA certificates are required in total, 67 CA certificates are required for 99%. Without the Let's Encrypt certificates, 45 are required for 95% and 85 for 99%.

In conclusion, the numbers presented in this section show that it is possible to validate the vast majority of certificates even when only a small subset of root and intermediate certificates are available. While adding CA certificates does not scale as well for Alexa 1 M domains as it does for all TLS certificates, in both cases a very high coverage of server certificates can be achieved with a low number of CA certificates. In addition, one root or intermediate certificate maintains is the issuer to very large number of server certificates. Therefore, centralizing the validation and storage of certificates – in contrast to validating the whole chain for each server certificate anew – should have a huge potential in reducing costs in an authentication framework based on TLS certificates.

3.4.2 Cryptographic Algorithms in Use

Another metric important for the implementation of our system is which cryptographic algorithms are used for the public key pairs and signatures of the certificates and how often they are present. For the analysis in this section, the set of certificates that fulfills the requirements described above was retrieved on the 27th of April and comprises 208,431,988 certificates.

In general, there are five key types defined for X.509 certificates: RSA, DSA, Diffie-Hellman Key Exchange, KEA, and ECDSA/ECDH [39]. However, only RSA and ECDSA are commonly in use today: Out of the certificates in the data set, nearly 85% specify an RSA public key, the rest specifies an ECDSA key. Table 3.1 lists the number of public key types further divided by the public key length for RSA keys and the curve used for ECDSA keys.

For creating and validating the signature of a certificate, the encryption algorithm is usually combined with a cryptographic hash function that is used to create a digest of the certificate. We refer to the combination of both algorithms as signature algorithm. The number of absolute occurrences of each signature algorithm is summarized in Table 3.2.

The hash functions we observe in our data set are SHA-1[4], SHA-256, SHA-384, and

[4]SHA-1 is not considered secure anymore and certificates should not be issued using this hash function

RSA key length	Number	Share	ECDSA curve	Number	Share
2048	156,746,476	0.75	P-256	28,908,431	0.14
3072	1,010,383	$4 \cdot 10^{-3}$	P-384	2,446,235	$1 \cdot 10^{-2}$
4096	19,317,221	$9 \cdot 10^{-2}$	P-521	283	$1 \cdot 10^{-6}$
8192	1,753	$8 \cdot 10^{-6}$			
other	1,206	$6 \cdot 10^{-6}$			
Total	177,077,039	0.85	Total	31,354,949	0.15

Table 3.1: Number of public key types observed.

Signature Algorithm	OID	Total Number
sha1WithRSAEncryption	1.3.14.3.2.29	11,988
rsassa-pss	1.2.840.113549.1.1.10	2
sha256WithRSAEncryption	1.2.840.113549.1.1.11	182,247,401
sha384WithRSAEncryption	1.2.840.113549.1.1.12	1,501
sha512WithRSAEncryption	1.2.840.113549.1.1.13	10,678
ecdsa-with-SHA256	1.2.840.10045.4.3.2	26,158,204
ecdsa-with-SHA384	1.2.840.10045.4.3.3	2,205
ecdsa-with-SHA512	1.2.840.10045.4.3.4	9

Table 3.2: Number of certificates that are signed with different signature algorithms.

SHA-512. We also observe the signature algorithm rsa-pss, but as it is only used in two certificates we do not consider it further.

In Table 3.3, we organize the numbers presented in Table 3.2 by hash algorithm and encryption algorithm. From this table, we can deduct that RSA is used for approximately 87% of signatures, ECDSA for 13%. In terms of hash functions, SHA-256 is clearly leading with being used in over 99% of signatures. While the other three hash algorithms cover minimal relative shares, the importance of SHA-1 should not be underestimated. 809 CA certificates are signed using SHA-1, out of which 67 are root certificates. Therefore, while the number of certificates that are signed with SHA-1 is low, SHA-1 still plays an important role for the validation of many certificate chains.

	SHA-1	SHA-256	SHA-384	SHA-512	Total
RSA	11,988	182,247,401	1,501	10,678	182,271,568
ECDSA	0	26,158,204	2,205	9	26,160,418
Total	11,988	208,405,605	3,706	10,687	208,431,986

Table 3.3: Number of signature algorithms occurring grouped by hash and encryption algorithms used.

anymore. However, especially root and intermediate certificates that were issued a long time ago and are not expired yet are signed using SHA-1.

With the data presented in this section, we gain a basic understanding of the current situation of the TLS ecosystem. We use the information that a small number of CA certificates issues the vast majority of domain certificates during the design (section 5.2.1) and evaluation (section 7.2) of our system. Furthermore, understanding which cryptographic algorithms are commonly in use is important for the implementation of our system (section 6.2.2).

4 Related Work

In this chapter we introduce previous work and ongoing efforts with goals or approaches similar to ours. In section 4.1 we briefly describe several proposals that aim to improve certain properties of PKIs by relying on blockchain technology. We discuss the Ethereum Name Service in section section 4.2. Then, we introduce two approaches that leverage existing TLS infrastructure for blockchains: did:web (section 4.3) and AuthSC (section 4.4).

4.1 Blockchain-based PKI Solutions

There exist numerous proposals to integrate blockchains and existing PKI infrastructure. However, the focus of these approaches is not to provide identity solutions for blockchain applications, but to leverage the blockchain for improving the properties of (the TLS) PKI. Giving an overview of all research that has been done in this field is out of the scope of this work, so we focus on approaches that target Ethereum or Ethereum-like blockchains and include CAs for issuing certificates. Various other approaches ([2], [7], [20], [24], [37], [43], [46]) do not include CAs in their design and introduce web-of-trust like solutions instead, which means the incompatibility with existing protocols does not solve the inherent bootstrapping problem, or develop new blockchains, which means that the certificate information cannot be used for Ethereum.

Khieu and Moh propose CBPKI, a cloud blockchain-based public key infrastructure [26]. CBPKI uses a combination of a CA hosted in the cloud as a stateless web service and using smart contracts to store associated information such as a certificate's revocation status. The idea is that the CA's security can profit from security measures that have already been put in place by the cloud provider, while denial of service attacks that focused on obstructing access to revocation information would now have to target the blockchain network itself. However, the approach does not fit our requirements as only the certificate hash is stored on-chain, but not relevant information such as the subject name or the public key. Consequently, certificates cannot be used for on-chain authentication. In addition, the approach relies on CAs adapting to it and issuing a new type of certificate that contains the address of the smart contract that contains it information.

Chen et al. suggest CertChain as a decentralized and tamper-proof tool for auditing certificates. In this model [11], CAs need to generate and sign certificate operations,

such a registration, update, or revocation, which they then broadcast to the blockchain. Certificate operations are stored in a newly defined data structure called *CertOper*, which is used for storing and traversing the certificate information. This proposal requires a new certificate format, an adapted implementation of Ethereum, and a new type of CAs that also act as miners in the blockchain network, which are all properties that contradict our requirements.

Kubilay et al. introduce CertLedger, a PKI system with the intention of shifting trust from CAs to the blockchain and providing certificate and revocation transparency [27]. On CertLedger, all TLS certificates, their revocation status, the revocation process, and CA management are handled. Once a certificate is submitted to CertLedger, it is validated and only added if valid. This means that client can simply refer to CertLedger instead of validating a certificate themselves when they are authenticating a public key. This also means that the client does not need to store and maintain a set of trusted roots anymore. In addition, CertLedger provides a transparent revocation system and allows owners of certificates – not just the issuers – to revoke them. While this proposal fulfills many of our requirements, it does not allow open participation: The set of trusted CAs is defined by CertLedger board and all validation decisions are made depending on it. This means that (i) the CertLedger board needs to be fully trusted by clients, (ii) clients cannot distrust individual CAs, and (iii) clients cannot add root certificates for specific applications.

Matsumoto and Reischuk describe Instant Karma PKI (IKP), an incentivization platform aiming to prevent fraudulent issuing of TLS certificates [30]. With IKP domains can define and report CA misbehavior, and CAs can sell insurance against misbehavior. CAs and domains implement their policies in smart contracts, which allows for an automated process to trigger financial transactions to the affected domain and the reporter if a CA is observed to issue an unauthorized certificate. The amount that is payed out is defined by a reaction policy published on-chain by the CA. This information is public, allowing clients to assess how likely it is that a CA maintains high security standards. This proposal focuses strongly on improving the security of the TLS ecosystem, but does not align with our goals and requirements: certificates are not presented to the blockchain unless they are fraudulent and CAs have to take significant action to make the system work.

Yakubov et al. propose a blockchain-based PKI management framework that supports the issuing, validation, and revocation of certificates [48]. In their system, participating CAs create smart contracts for their certificates; these contracts contain the hash of all issued certificates together with their revocation information. This information may only be altered by the corresponding CA. The authors extend the X.509 format with a custom extension that contains among other things the smart contract address of the issuing CA. When a verifier receives a certificate, they refer to the smart contract and verify that the hash is contained, that the certificate is not revoked, and that the chain of trust is valid. Just as the approaches before, this proposal relies on proactive CAs. Additionally, a new

certificate format is required and only the certificate hash is stored on-chain, which is not sufficient for an on-chain authentication framework.

The aforementioned studies on blockchain-based PKIs all focus on improving the security and transparency of TLS certificate management. This entails that they leverage blockchain technology to improve the TLS ecosystem. In this work, we aim to do the reverse: leverage the TLS ecosystem to provide an authentication framework for Ethereum.

4.2 Ethereum Name Service

Ethereum Name Service (ENS) was launched in 2017 and aims to provide a decentralized way to address blockchain resources in a human-friendly way [19]. Similar to the Domain Name System (DNS) which maps web domains to IP addresses, it enables users to resolve human-readable names to Ethereum addresses. ENS is curated by the Ethereum Foundation and is described in three Ethereum Improvement Proposals: EIP-137, EIP-162, and EIP-181 [6]. ENS names are dot-separated hierarchical names called domains; currently, the only supported top-level domain (TLD) is ".eth". TLDs are owned by smart contracts called registrars. The owner of a domain can create subdomains and transfer the ownership of the subdomains to other parties. For example, Bob can acquire the ownership of the address "bob.eth" and configure rules for the creation and ownership transfer of its subdomains.

The ENS architecture consists of two central components: Registries and resolvers. As noted above, registries are authoritative over one TLS. As ".eth" is currently the only supported TLD, there exists one registry. The ".eth".registry is currently controlled by a 4-of-7 multisig. It is planned to transfer control to a decentralized account in the future [6]. A registry contains a list of all its subdomains and its respective owners, resolvers, and cache expiration. All Ethereum accounts that support the relevant standards can be the owner of a domain. Resolvers are responsible for translating the domain to an actual Ethereum address. To translate a domain to the address, the enquirer first obtains the resolver address from the registry and then retrieves the address and optional associated information from the resolver.

The big advantage of ENS is that it was developed with Ethereum in mind and is optimized for its properties. For example, instead of working internally with human-readable names that are inefficient to implement in Ethereum, 32-byte namehashes are used. The namehash algorithm is a recursive algorithm that allows to compute the hash of a new subdomain without requiring knowledge of the human-readable form of the parent domain. Once ENS is established, it is a cost-efficient and decentralized system providing human-readable identities to Ethereum addresses. One problem, however, remains: Domain ownership can be acquired through auctions in which anyone can participate and the highest bidder wins. This means that ENS domain names cannot be

intuitively mapped to real-world identities. Furthermore, there is no judicial system in place which would allow the seizing of individual domains, for example in the case of copyright disputes.

4.3 did:web

The authors of did:web point to the fact that current decentralized identifiers (DIDs) methods for blockchains lack of enough trusted data for successful bootstrapping. Therefore, they propose to leverage the TLS system and to use web domain names as identifiers. In their system, a DID document is created that contains the web domain name and a public key, for example an Ethereum account address. This document is stored at the web domain under a well-known path. A verifier can then connect to the web server through a TLS connection and retrieve this information. As the TLS connection is authenticated, the public key contained in the DID document is as well.

While the foundational idea of using TLS mechanisms for a blockchain-authentication framework is the same, this approach does not align with the properties that we want to achieve. It depends on the availability of the server and the question of how to migrate the information safely from the web to the blockchain remains.

4.4 AuthSC

In section 2.1.6 we discuss the security problems that can arise from Ethereum addresses being independent from real-world identities: When addresses are, for example, announced on web sites, it is the burden of the user to verify the correctness and authenticity of this address. Proposed approaches such as account address images, vanity addresses, or announcing the address through different channels merely mitigate and do not solve this issue. With this problem in mind, Gallersdörfer proposes AuthSC [21].

The basic idea of AuthSC is that domain owners sign the Ethereum address that they control with their private key and store the signature together with the domain name in the smart contract. When a user is interested in interacting with the contract, they retrieve the information and check that the domain name matches their expectation. Then, they connect to the web host serving the domain through a TLS connection in order to obtain the certificate and its chain. They ensure the validity of the certificate in accordance with the usual criteria (cf. section 2.2.4). Finally, if the certificate is valid, the user verifies the signature that was stored in the smart contract with the public key contained in the certificate. If the signed data corresponds to the address of the Ethereum account that they interact with, the account is deemed authenticated and linked to the real-world identity of the web domain operator. Consequently, the user can

proceed interacting with the account. To address privacy concerns, AuthSC also offers the possibility to store the signature without the domain information in the contract and distribute the domain name off-chain instead. In this case, it is the discretion of the identity owner to share the signature with entities and disclose which Ethereum accounts they operate. The verifier can still check the binding of the address to the identity by performing the steps described above.

This approach solves the bootstrapping problem of identity systems by relying on the already established TLS ecosystem. Users are already able to link web domain names to real identities. When an Ethereum addresses is bound to a web domain, it is ultimately linked to the real identity. However, this approach does not support the on-chain authentication of accounts as the certificate retrieval, the certificate validation, and the signature validation are performed off-chain. Furthermore, the approach relies on the availability of the web server and that it provides the same certificate that was used for creating the signature. In this thesis, we take the foundational idea of AuthSC and use TLS certificates for an Ethereum authentication framework but design our system in a way that allows on-chain authentication and resolves the availability issue.

5 Design

The goal of this work is to design a system that enables binding Ethereum account addresses to real-word identities, namely web domain addresses. The system design aims to comply with the requirements presented in section 3.3. In this chapter, we present the conceptual design of our system, which includes the endorsement scheme (section 5.1) and the on-chain assertion of identities based on TLS certificates (section 5.2). In addition to our final design, we discuss alternative solution approaches and their advantages and drawbacks. In section 5.3, we summarize how our design complies with the requirements.

5.1 Endorsement of Account Addresses

We define the endorsement of an Ethereum address as the signature of the address value together with optional associated data, such as the web domain or corresponding TLS certificate, produced with a private cryptographic key. An endorsement indicates that the endorser claims to own the address, i.e. that they receive ingoing funds, control outgoing funds, vouch for data associated with the address, and are the originator of outgoing transactions. Endorsements need to present some kind of liability and make only sense in scenarios where an adversary cannot gain advantage by signing an address they do not control. For example, signing a contract or EOA address to prove that one owns the ether contained in this account does not make sense, as anyone can produce a signature for an address.

5.1.1 Using TLS Certificates for Endorsements

To endorse an account, any private cryptographic key can be used. However, if the public key cannot be mapped to an identity, this endorsement is meaningless. We rely on certificates that bind an identity to the public key, which in turn binds an identity to the endorsement and account. In theory, any type and kind of certificate could be used. However, as we point out in requirement R8, our aim is to build a system that is compatible with widely-used standards to support fast adoption. Therefore, we chose to rely on X.509 certificates.

In general, it is possible to design a scheme where CAs sign the public key or address of an account or smart contract, respectively, and issue a certificate that binds the account

directly to a real world identity. However, this requires that CAs actively engage in such a new scheme, which contradicts requirement R8. Consequently, we decide to work with TLS certificates, a certificate type that is already routinely issued. In addition, TLS certificates are trusted by most internet users and provide a vast and well-studied ecosystem. For our system, we use root and intermediate certificates as intended for trust propagation. Instead of being presented during the TLS handshake, server certificates are used to create endorsements.

While using TLS certificates reduces the types of identity that can be asserted to web domains in the short run, the broad deployment and sheer number of existing TLS certificates is an advantage that encourages fast adoption of our system. In the long run, our vision is to open our system up to other certificate types. This would enable the assertion of identities other than web domains, permit entities not in ownership of a web domain to prove their identity, and empower users to build application-specific PKIs for the identity management on Ethereum. This work, however, focuses on the opportunities and limitations of using TLS certificates.

5.1.2 Endorsement Content

The format and content of endorsements needs to be specified such that misuse of endorsements is avoided. This means the endorsements must be unambiguous (R2). In particular, endorsements need to meet the following requirements:

- An endorsement issued for one Ethereum address may not be reused for another Ethereum address.
- It must be clear to which web domain an address is linked. The situation when this is unclear might arise when a certificate is issued for multiple web domains.
- It must be possible to identify the domain certificate with which the endorsement was created in order to retrieve the public key and to check the validity and revocation status of the certificate.
- It should be possible for the issuer to specify an expiration date of the endorsement.

An endorsement comprises a signature and associated data. The signature is computed over the hash of the concatenation of the address account $addr$, the web domain ID_{domain}, the unique certificate identifier ID_{cert}, and the optional expiration date $date_{exp}$. An endorsement E is characterized informally by

$$E = \{sign(hash(addr|ID_{domain}|ID_{cert}|date_{exp}), key_{priv}), addr, ID_{domain}, ID_{cert}, date_{exp}\}$$

where $sign(m, k)$ produces the cryptographic signature of the message m using the key k, key_{priv} is the endorsement creator's private key, and $hash(x)$ computes the cryptographic hash of x. While not strictly necessary for endorsing, we decide to include ID_{cert} in the signature, as it enables the individual revocation of endorsements (R3). In case of

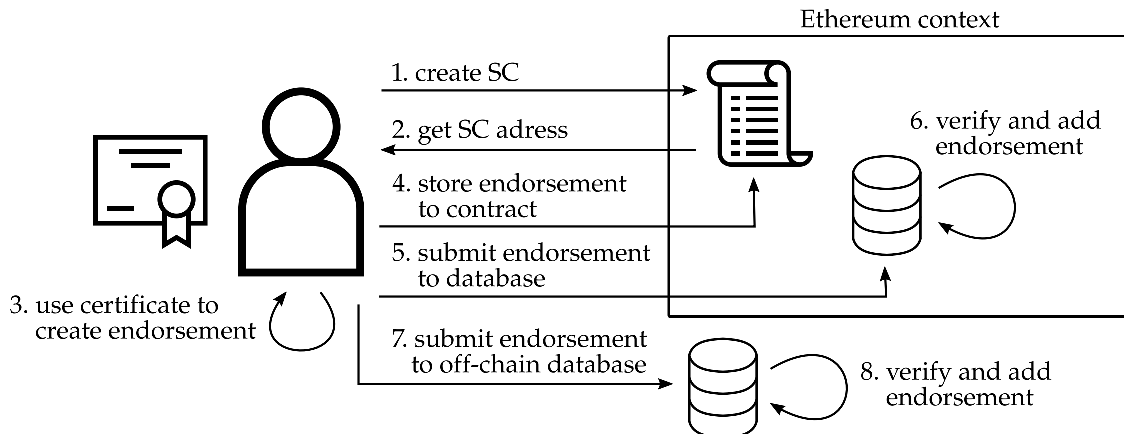


Figure 5.1: Different ways of endorsing a smart contract: internally (step 4), externally on-chain (steps 5 and 6), and externally off-chain (steps 7 and 8). The endorsed contract and the endorsement are created in steps 1 to 3.

internal endorsements (cf. section 5.1.3), *addr* does not need to be explicitly provided with the associated data but should be inferred from the smart contract address.

Another important information concerning an endorsement is its creation date. However, if the creation date is provided by the endorsement creator, it is not trustworthy as a malicious party can simply provide an earlier date. Therefore, this requires a TTP that attests the creation date or the submission date, respectively. Fortunately, Ethereum is such a TTP and can be used to track when an endorsement was added.

5.1.3 Internal and External Endorsements

The endorsement of an address can be stored in- or outside the corresponding account. We refer to endorsements that are stored inside a smart contract as internal endorsement. In contrast, an external endorsement is stored by another entity than the account, which may be on- or off-chain. An address can simultaneously be endorsed in- and externally. Figure 5.1 depicts how all three types of endorsement can be deployed.

An *internal endorsement* contract can be created by any owner of a valid TLS certificate (called subject below) in the following way:

1. The subject creates a smart contract with the main contract functionality and support for the standardized endorsement interface.
2. The subject deploys the contract on-chain and retrieves the address of the deployed contract (steps 1 and 2 in Figure 5.1).
3. The subject uses their private key to create the endorsement. This action takes place off-chain (step 3 in Figure 5.1).

4. Through functions offered by the endorsement interface, the subject stores the endorsement in the contract (step 4 in Figure 5.1).

Subsequently, any other entity that interacts with the contract can retrieve the endorsement and verify it by fetching its corresponding certificate, validating it, and obtaining the certificate's public key. The validation of the signature is always performed by the verifying party.

An internal endorsement has the advantage that the information that an account is endorsed is immediately available. A verifier does not need to act proactively and inquire directory services to find out if and by whom a contract is endorsed and an attacker does not have the chance to keep this information from the verifier. However, internal endorsements do not work for EOAs and for smart contracts that have been deployed without the endorsement functionality, a functionality that is demanded by requirement R6.

An *external endorsement* can be created in the following way:

1. The subject retrieves the address of the EOA or contract they want to endorse (step 2 in Figure 5.1).
2. The subject creates the endorsement off-chain (step 3 in Figure 5.1).
3. The subject publishes the endorsement together with relevant information. The publication can take place at on- or off-chain endorsement registrars, web sites, or also in private messages to specific verifiers (steps 5 to 8 in Figure 5.1).

In this case, a verifier has to act proactively. When they want to interact with a contract, they need to rely on third-party resources to gather endorsement information. The fundamental advantage of external endorsements is, in compliance with requirement R6, that they can be created for all accounts including EOAs and contracts that do not support endorsement functionality. Additionally, the cost of signature validation can be partly borne by the subject, when they submit the signature to a trusted party that performs the validation and offers this information to verifiers.

As both approaches can be applied simultaneously, we decide to offer both possibilities in our system to combine the advantages. We provide an interface for internal endorsements and endorsement registrars and discovery services for external endorsements. Users can then chose what the best applicable option – either internal, external, or combined – for their application is. This is also in compliance with requirement R7, which asks for a system that is flexible and adaptable for different use-case scenarios.

5.1.4 Storage and Distribution of Endorsements

In this section, we define how the structures and policies with which endorsements are maintained in our system. The operations differ between internal and external

endorsements. While defining the CRUD operations, we have to keep requirements R6 – enable signing contracts that do not support the endorsement interface – and R7 – flexible design that does not limit applications – in mind. Furthermore, the revocation of individual endorsements must be supported (R3).

Internal-Endorsement Interface

To endorse a contract internally, the contract needs to support a clearly specified interface. The interface should ensure on application level that only authorized entities – the owners of a contract – are able to add endorsement information. This mitigates that authentic endorsements can be replaced by arbitrary endorsements, which would allow Denial-of-Service attacks. In contrast, it should be possible for anyone in possession of the corresponding private key to submit the revocation of an endorsement. This is to allow the revocation of an endorsement in which the initial owner has lost control over the contract.

The implementation of the internal-endorsement interface needs to offer the following functionality:

Create The creation of an internal endorsement is outlined in section 5.1.3.

Read The endorsement information can be accessed through getter functions integrated in the interface. The endorsement is not validated on submission, so it needs to be checked on retrieval. As the address is not explicitly stored, the address must be obtained through functions offered by the EVM.

Update Endorsements themselves are immutable. If new information is required, a new endorsement needs to be created. The only associated information that can be updated is the revocation status. The endorsement creator can submit a signed message stating that an endorsements is revoked. The endorsement interface must require that the signature is validated before the revocation status is updated.

Delete It should be possible to remove internal endorsements before their expiration. However, it must be ensured an endorsement's revocation status persists even on deletion. Otherwise it would be possible for an attacker who has gained control over the contract to remove the revoked endorsement information and resubmit the unrevoked endorsement.

External-Endorsement Database

In addition to the internal-endorsement interface, we propose a central database for storing endorsements. Providing a central database empowers verifiers to proactively and conveniently search for endorsements. Such a database query can have two distinct goals: The verifier might either be interested whether and by whom a specific Ethereum address was endorsed or whether there exist endorsements for a specific web domain.

In addition, a central data base facilitates the revocation of endorsements. Another advantage is that the endorsement can be validated upon submission, subsequent parties interested in the endorsement do not need to perform the validation again.

The implementation of the external-endorsement database should provide the following functionality:

Create An endorsement E, as defined in section 5.1.2, is submitted to the database. The validation procedure retrieves the certificate with the certificate ID ID_{cert}, checks that the certificate is issued for the web domain ID_{domain}, and obtains the public key key_{pub}. If the endorsement's signature is valid and not expired, the endorsement it stored in the database.

Read Endorsements can be retrieved with $addr$ or ID_{domain} as key. As multiple endorsements per account or web domain may exist, the query returns a set of endorsements. The querying party is responsible for checking the endorsements for one that is signed by a certificate whose root certificate they trust.

Update Endorsements themselves are immutable information. The only associated information that may change is the revocation status. If the original issuer of an endorsement wants to revoke it, they sign the respective information and store it with the endorsement.

Delete Unexpired endorsements may not be deleted from the database. Some applications might also accept expired endorsements, therefore, expired endorsements should not be deleted while it is allowed to do so. However, if an endorsement was revoked, the revocation information should persist.

It is important to note that external endorsements may also exist that are not stored in the central database. Submitting endorsements to the database is a voluntary action with the intent to facilitate the discovery of endorsements.

5.2 Enabling On-chain Decisions

To make a decision on trust based on an account endorsement, the verifier relies on two components: The validity of the signature and the signer certificate and the policies defined by the verifier. The policies depend on the use-case scenario and can be implemented in the application's functionality. Determining the validity of a signature poses a greater challenge as several factors may influence the decision:

- Time of validation: A verifier validating a certificate during the validity period of the certificate comes to a different conclusion than one validating it after the expiration date.
- Mismatch of certificates: A subject might own several certificates that are issued for different keys. If a verifier retrieves the wrong certificate, the signature validation inevitably fails.

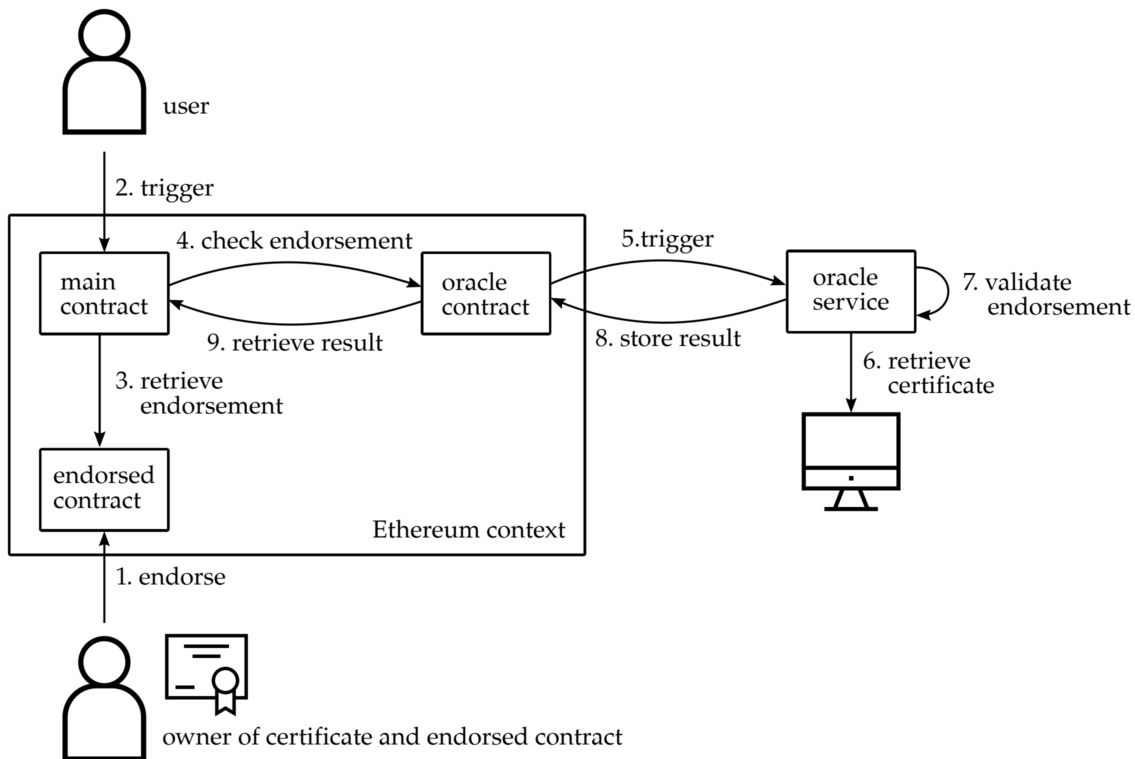


Figure 5.2: Depiction of the workflow when using the oracle approach for certificate validation.

- Different trusted roots: Ultimately, the decision on trusting a certificate depends if the verifier trust the root of the certificate chain. The set of trusted root may differ for different verifiers.

There is no absolute truth on the validity of certificates and we cannot expect that Ethereum miners validate certificates based on the off-chain TLS PKI and come to a unanimous decision. Therefore, we need to design a mechanism that allows TLS certificate validation solely based in the context of the Ethereum blockchain. Only if this is possible, endorsements can be validated on-chain as required by R1.

5.2.1 Using Oracle Services Versus Migrating the PKI On-chain

We have investigated two approaches to solve this problem and enable the on-chain usage of TLS certificates: Relying on off-chain oracles and migrating part of the PKI on-chain. In this section we discuss both approaches and stress out why we decided to implement the second one.

Using oracles for certificate validation is based on the idea of "outsourcing" the main tasks of identity assertion, namely the storage and validation of certificate information,

to the off-chain world. Figure 5.2 depicts the involved steps when using this approach. Identity owners embed the endorsement information in their contract (step 1). Once a smart contract is triggered (step 2) that needs to authenticate the contract, it retrieves the endorsement (step 3) and requests a trusted oracle contract to verify the endorsement (step 4). The off-chain oracle service is triggered (step 5) to obtain and validate the certificate chain belonging to the endorsement's web address (step 6), verify the endorsement signature with the respective server certificate's public key (step 7), and report its conclusion about the validity of the endorsement to the oracle contract (step 8). The main application contract can then retrieve the validation result from the oracle contract (step 9). Alternatively, the oracle contract can also only provide information whether the certificate is valid together with the public key and the original contract performs the validation of the signature. Obtaining the certificate chain in step 6 is done by performing a TLS handshake with the web domain server; the set of trust anchors used for the chain validation is determined by the oracle service. If the contract is externally endorsed, the identity owner submits the endorsement to a database in step 1 instead to the contract and the verifier retrieves it from the database in step 3 instead.

There exist different possibilities for the design of the oracle service. One possibility is that clients self-host an oracle, which could use a combination of already existing tools for certificate retrieval and validation, like OpenSSL, and a small routine that is developed specifically for endorsement verification. If a user is not able to host their own oracle, they could rely on technologies like TLSNotary[1], which produces cryptographic proofs of TLS traffic content, to query the status of the certificate from online certificate validators as offered by Digicert[2], SSLShopper[3], Geocert[4], or SSLChecker[5]. In this case, the validation of the endorsement would need to be performed by a computation oracle or on-chain by the requesting contract itself.

Using oracles has several advantages: It is cheaper as certificate information is neither stored nor validated on-chain. The oracle services can be patched and upgraded, which makes them adaptable in case of changes of the X.509 or TLS certificate specification. And as the most recent information is available, the revocation information of the certificate is up to date. However, the trust and security issues that arise with oracles are not the only problem of this approach. As it depends on off-chain server responses, it violates the availability requirement R5. Additionally, there might be a mismatch between the certificate that was used to create the endorsement and the certificate that is provided by the server. Considering that server certificates have a significantly shorter life time than CA certificates, this problem might be mitigated – not solved – by storing the server certificate with the endorsement on-chain. Off-chain, only the CA certificates

[1] https://tlsnotary.org/, accessed 13.05.2020
[2] https://www.digicert.com/help/, accessed 13.05.2020
[3] https://www.sslshopper.com/ssl-checker.html, accessed 13.05.2020
[4] https://www.geocerts.com/ssl-checker, accessed 13.05.2020
[5] https://www.sslchecker.com/sslchecker, accessed 13.05.2020

of the chain need to be obtained. However, this would diminish the cost advantage of the oracle approach.

To satisfy the availability requirement R5 and to solve the mismatch problem, the information that is required for certification validation needs to reside on-chain. We call this approach migrating (a part of) the TLS PKI to Ethereum. The information that is required to validate a certificate is the whole certificate chain from server to root certificate, the set of trust anchors as defined by the verifier, and the validation procedures.

The validation routine for X.509 certificates can be implemented and offered on-chain as Ethereum library. It is a security-critical component and needs to be carefully implemented and the source code needs to be openly available to be trusted by users. As R4 requires that the set of trust anchors is specific to the verifier, each contract that acts as verifier needs to declare their own set. There are two different options for storing the certificate chains: a decentralized and a centralized one. With the decentralized approach, each endorsement is stored with the corresponding certificate chain. The endorsement and the certificate are then validated each time a verifier wants to assert the identity against the set of trusted roots of the verifier. With the centralized approach, the certificate chain is not stored with the endorsement, but in a central database. The obvious advantage of this approach is that intermediate and server certificates need to be stored only once and can be shared by the server certificates. Additionally, if the validity of a certificate and its chain is asserted by the database when it is submitted and only valid certificates are accepted, the validation of the certificate needs to be performed only once. When verifiers assert an identity, they only need to verify that the certificate is present in the database, that the root certificate is part of the trusted roots, that the validity period has not expired, and that the certificate has not been revoked. These operations are significantly cheaper than performing the full validation for a certificate chain.

A difficulty with the migration approach is the assertion of the revocation status of certificates. In the oracle approach, the revocation status can be checked through further interactions to obtain a CRL or OCSP response. As the goal is to be independent from external servers, this is not possible for the migration approach. Luckily, both CRLs and OCSP responses are documents that are valid for a certain time period and that are commonly signed with the issuer private key and can consequently be verified on-chain. The idea is that the database entries of certificates can be updated with the current corresponding revocation information. This needs to be repeated while the certificate is valid and the validity period of the revocation status information expires. If a certificate is revoked, this information is final and does not need to be updated.

Table 5.1 presents an overview of the degree that the different approaches fulfill the requirements discussed in section 3.3 and that are relevant for the certificate framework. All options enable on-chain decisions and open participation: The oracle approach as

	Oracles	Decentralized migration	Centralized migration
R1 - Support of on-chain decisions	✓	✓	✓
R4 - Open participation	✓	✓	✓
R5 - Availability	✗	✓	✓
R7 - Cost-efficiency	✓	✗	∼

Table 5.1: Fulfillment of relevant requirements of the different approaches to enable on-chain validation of certificates and endorsements.

everyone can choose their trusted oracle and the set of trust anchors, the migration approach as anybody can define their own set of trust anchors on-chain. The oracle approach does not guarantee availability, as it relies on external servers, in contrast to the migration solutions, which operate based solely on information available on-chain. In terms of on-chain cost (R9), the oracle solution is the least expensive as most computations are moved off-chain. The centralized migration approach performs all necessary steps on-chain, but the number of computations and the storage size required are notably less than with the decentralized migration approach.

Overall, migrating the TLS PKI on-chain in a centralized manner satisfies our requirements to the greatest degree. We decide to proceed with this approach; the design of the central certificate database is discussed in section 5.2.2. Independent from our system and use case, this solution might also serve as a trusted proxy certificate validator and storage for resource-constrained devices. Instead of performing the validation of certificates by themselves, they can retrieve certificate status information from the blockchain.

5.2.2 Central Certificate Database

For enabling on-chain decisions on the validity of certificates, we rely on a central database that stores all valid certificates that have been submitted to the system. In the following, we will describe the CRUD (create, read, update, delete) operations of the database.

Create Certificates are submitted to the database one-by-one. Anyone can submit certificates. Before a certificate is stored, it is confirmed that it is valid. This check is performed in accordance with the criteria presented in section 2.2.4. As the signature of the certificate needs to be verified, the certificate must either be a self-signed certificate or the certificate's issuer's certificate must already be stored in the database. The validity period of the certificate must not be expired. If the certificate validation is successful, the relevant information is retrieved from the certificate and stored in the database. This includes a pointer to the entry of the issuer certificate; in the case of self-signed certificates, it is the certificate itself. The

revocation status information is set to *unknown*. If the certificate validation is not successful, the certificate is rejected.

Any self-signed certificate with valid format and content can be added to the database and subsequently act as trust anchor. This enables anyone to create and maintain their own application-specific PKI.

Read Certificate information can be retrieved from the database with a unique certificate identifier. The certificate chain can be retrieved thanks to the pointers that refer to the issuer of each certificate.

Update The only information that can be updated is the revocation status of certificates. For this purpose, either the CRL or the OCSP response corresponding to a certificate can be submitted. The submitted information is only used to update the revocation status information if it is valid and signed by the certificate's issuer. For the CRL, the certificate status is considered as *not revoked* when its serial number is not contained in the CRL and considered as *revoked* when it is contained. For OCSP responses, the certificate status is updated to the status that is contained in the response. In both cases, information about the time of the last update and the expiry date are stored. Once a certificate is marked as *revoked* in the database, the state cannot be reversed to *unknown* or *not revoked*.

Other certificate attributes cannot be updated in the database as all information reflects the information of the submitted certificate. If altered information is required, a newly issued certificate must be submitted with a new unique certificate identifier.

Delete Once submitted, certificates cannot be deleted. This is because other certificates and endorsements may rely on this certificate and their validity and revocation status cannot be verified sufficiently if certificates and their chain of trust are missing.

5.3 Evaluation of the Design

The design of our system is comprised of an interface for internal endorsements, a database for external endorsements, a database for certificates, and a library for parsing and validating TLS certificates. Table 5.2 summarizes how our design complies with the requirements that we identify in section 3.3.

By storing certificates on-chain and providing functionality for certificate and signature validation, we enable on-chain decisions on identity and trust (R1). With our endorsement scheme, we ensure that endorsements are unambiguous (R2) and can be individually revoked (R3). Our system does not enforce one absolute truth; everyone can use the system and define their own trust anchors and decision policies (R4). As all information required in terms of certificate and endorsement validation is stored on the

R1	Support of on-chain decisions	✓
R2	Unambiguous endorsements	✓
R3	Revocation of endorsements	✓
R4	Open participation	✓
R5	Availability	✓
R6	Compatibility	✓
R7	Flexible design	✓
R8	Fast adoption	✓
R9	Cost-efficiency	$\sim$

Table 5.2: Compliance with requirements of our design.

Ethereum blockchain, our system does not depend on the availability and state of web servers (R5). By introducing external endorsements, it is possible to bind identities to EOAs and already deployed, non-conforming smart contracts (R6). Nevertheless, we leverage the advantages of internal endorsements by providing an interface for internal endorsements that can be used for new contracts. It is possible to endorse a contract with a deliberate combination of internal and multiple external endorsements, serving a wide range of use-cases and application scenarios (R7). As we utilize TLS certificates to create endorsements, we do neither require CAs to take any action for our system to work nor do we rely on new certification processes. Together with leveraging the large amount of readily-available trusted data, we facilitate the fast adoption of our system (R8). While we have not chosen the cheapest approach in terms of on-chain computation, consolidating the maintenance of certificates helps to reduce the cost for identity verification and abolishes the need for off-chain resources that act as oracles (R9).

6 Implementation

In this section, we present the structure and key technical details of our implementation. Our prototype includes functionality for parsing DER documents (the data format in which X.509 certificates are stored), parsing and validating X.509 certificates and related artifacts, storing validated certificates in a central database, and enabling the endorsement of Ethereum addresses. We introduce the general structure of our prototype in section 6.1, before discussing its two parts in more detail: The certificate framework in section 6.2 and the endorsement framework in section 6.3. The smart contracts presented below are implemented in Solidity. We implemented and tested the system with support of the Truffle framework and deployed it on a local test blockchain operated with Ganache.

6.1 Prototype Structure

Our prototype includes reference implementations for verifiers as well as functionality for the creation, storage, and validation of endorsements and the submission, storage, and validation of certificates. The prototype has a layered and modular structure, meaning that subsystems like the validation routine for X.509 certificates can also be used independently from our application. In total, our prototype implementation is comprised of 14 contracts and libraries: Three contracts for the endorsement functionality, three contracts concerning the validation and storage of certificates, four contracts provisioning cryptographic algorithms, and four libraries with helper methods.

The simplified basic structure of our system is displayed in a UML-like diagram in Figure 6.1. Helper contracts and contracts for crypto algorithms are not displayed. The `EndorsementDatabase` is the interface to the authentication framework for verifiers. It stores valid external endorsements and root stores. In order to validate endorsements and confirm the link of an endorsement to a web domain name, `EndorsementDatabase` refers to `CertificateStore`. This entity validates and stores certificates and certificate chains. For the validation of X.509 documents such as certificates, CRLs, and OCSP responses, it utilizes the library `X509Parser`, which in turn relies on `ASN1Parser` for parsing the documents. The functionality of `Verifier`, `EndorsementDatabase`, and `X509Parser` requires the validation of signatures, which is provided by `SignatureValidation` and different crypto libraries used by it. The `InternallyEndorsed` contract is a structurally independent contract.

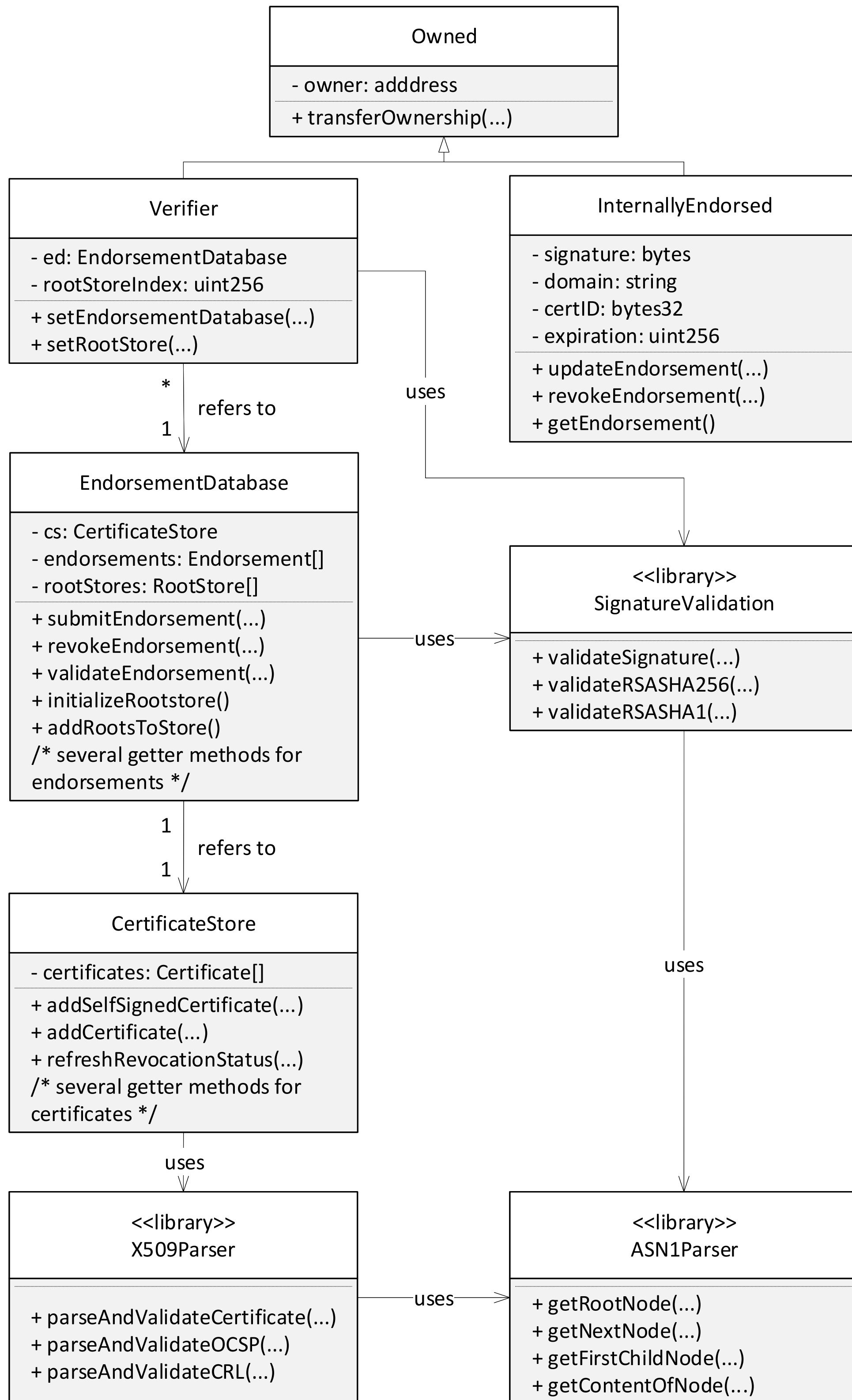


Figure 6.1: Structure of the authentication framework.

```
1  pragma solidity ^0.5.12;
2  import "./X509Parser.sol";
3
4  contract CertStore {
5  event CertificateAdded(bytes32 fingerprint);
6  event SelfSignedCertificateAdded(bytes32 fingerprint, uint256 rootIndex);
7  event CertificateUpdated(bytes32 certID, uint256 expiration);
8  event CertificateRevoked(bytes32 certID);
9
10 mapping (bytes32 => X509Parser.X509certificate) private certificates;
11
12 function addSelfsignedCertificate(bytes memory der) public {...}
13 function addCertificate(bytes memory der, bytes32 issuerID) public {...}
14 function refreshOCSP(bytes32 certID, bytes memory ocsp) public {...}
15 function refreshCRL(bytes32 certID, bytes memory crl) public {...}
16
17 /* Various getter functions are omitted for the sake of brevity. */
18 }
```

Listing 6.1: Extract from the code of the central certificate database `CertificateStore`.

6.2 Certificate Framework

We denote the part of our system that provides functionality for the storage, maintenance, retrieval, and validation of X.509 certificates as *certificate framework*. In the following, we discuss the parts it is comprised of: the certificate database, the routines for validating and parsing X.509 documents, and the format for the internal processing of certificates.

6.2.1 Certificate Database

The central component of the certificate framework is the `CertificateStore` contract, which serves as certificate database and is point of reference for all actions concerning certificates. In accordance with our design described in section 5.2.2, anyone can submit root, intermediate, and server certificates to be added, validated, stored, and maintained. The shortened code of `CertificateStore` which contains the most relevant function and field definitions are shown in Listing 6.1.

All certificates that have been submitted are stored in the mapping `certificates` (line 10), which maps a certificate's unique identifier to the certificate information. We decided to use a certificate's SHA-256 hash as its unique identifier, as the SHA-256 hash is natively supported by Ethereum and its value cannot be controlled by an attacker to forge a duplicate certificate due to its nature as cryptographic hash. The certificate information is stored in structs of type `X509Certificate`, which is explained below. The

type of a certificate is not explicitly stored but can be inferred from their properties: Root certificates are their own issuers and they are a CA certificate, intermediate certificates have a different from themselves and their CA flag is set, and server certificates have a different issuer and their CA flag is *not* set.

Adding Certificates

To add a certificate to the database, the user provides its DER-encoded representation in bytes to the respective method. To add a root certificate, the function `addSelfSignedCertificate()` (line 12) is invoked. This function passes the DER data to the validation function in `X509Parser`, which retrieves the certificate's relevant information, confirms that the certificate is not expired, and returns it in a `X509Certificate` struct. `CertificateStore` stores this struct in its `certificates` mapping. For the other two certificate types, the process is slightly different: Together with the DER, the user needs to provide the issuer's unique identifier, i.e. SHA-256 fingerprint. `CertificateStore` retrieves the issuer from the `certificates` mapping and passes it along with the DER to the validating function to verify the signature, content, and expiration of the certificate.

The processing of a certificate is aborted and the transaction is reverted when

- the referenced issuer is not available under the specified ID,
- the submitted DER document does not comply with the X.509 ASN.1 specification,
- or when the certificate is invalid.

Consequently, only valid certificates are stored by `CertificateStore`. Aside from the validity period and the revocation status, clients do not need to re-validate the certificate upon usage. When a new certificate is added to `CertificateStore`, the event `CertificateAdded` or `SelfSignedCertificateAdded` (lines 5 and 6) is emitted. `CertificateStore` offers various public functions to retrieve stored certificate information.

Refreshing Revocation Status of a Certificate

Once a certificate is submitted, its properties like the subject name or the expiration date cannot be modified as they were retrieved from the original document and the certificate cannot be deleted as other certificates or endorsements may depend on it. The only associated property of a certificate that can be updated is its revocation status. This can be done by any user, not just the certificate owner. As first step, the user obtains the current OCSP response or CRL from the responsible certificate authority off-chain. This is a common action that verifiers of certificates perform when opening TLS connections. Then, the user passes the obtained document to `CertificateStore` along with the certificate identifier by invoking either `refreshOCSP()` (line 14) or `refreshCRL()` (line

```
-----BEGIN CERTIFICATE-----
MIIEvzCCA6egAwIBAgIRAMT1T7RK2JtGCAAAAAA4ymcwDQYJKoZIhvcNAQELBQAw
QjELMAkGA1UEBhMCVVMxHjAcBgNVBAoTFUdvb2dsZSBUcnVzdCBTZXJ2aWNlczET
/* 22 lines omitted */
w/Dk7ZlTIOoL+CN8yIUHLvOvPpKTFBxm4O+jN27DahhJETbraQxVGJtNjO+/LJ1N
gQ+PGC2FmXOFgKgYKxPW7kCfwg==
-----END CERTIFICATE-----
```

Listing 6.2: Shortend example of the X.509 certificate representation in PEM format.

15). `CertificateStore` then retrieves the stored certificate and the issuing certificate and passes them to `X509Parser`. `X509Parser` validates that the response is authoritative over the certificate and that it was signed by the certificate issuer. In case of OCSP, it returns the status as contained in the response; in the case of CRL, it returns *revoked* if the certificate is contained in the list and *not revoked* otherwise. It is recommended to use OCSP whenever possible, as CRLs might be quite large and incur a high processing fee. However, root and intermediate certificates are usually only revoked through CRL, making this the only option. As OCSP responses and CRLs do have an expiration date, this information is updated together with the status. When the revocation status of a certificate is updated, a corresponding event (lines 7 and 8) is emitted.

6.2.2 Certificate Parsing and Validation

The largest part of our implementation – both in terms of lines of code and in total computing time – concerns itself with parsing and validating X.509 certificates. This functionality is provided by our `X509ParserLibrary`, which can also be used for other applications. In our implementation the parsing and validation of certificates happen simultaneously. This means that the certificate information is validated already while we transform the provided certificate representation into our internal representation. While this means that certificates cannot be parsed or validated independently from the other step, it reduces the processing cost for our application.

In the remainder of this section we present the common certificate format that is the input to our routine, the representation of certificates in our implementation, and the cryptographic algorithms and certificate extensions that the implementation supports

X.509 Certificate Format

X.509 certificates and other artifacts are usually stored and distributed in the ".pem" format. These documents contain the Base64-encoded X.509 artifacts. Listing 6.2 displays the representation of a X.509 certificate in such a document. The first step of processing a

```
1 SubjectPublicKeyInfo ::= SEQUENCE {
2     algorithm AlgorithmIdentifier,
3     subjectPublicKey BIT STRING }
4
5 AlgorithmIdentifier ::= SEQUENCE {
6     algorithm OBJECT IDENTIFIER,
7     parameters ANY DEFINED BY algorithm OPTIONAL }
```

Listing 6.3: Definition of the structure of a public key in ASN.1.

certificate is converting the Base64 encoding to the hexadecimal representation, which is the so-called DER encoding. To save costs, this step is performed off-chain, i.e. certificates are submitted in their hexadecimal format and not Base64 encoded. All following steps, however, need to be performed on-chain as it would otherwise be impossible to verify the signature and assert the integrity of the certificate. Unfortunately, this means that it is not possible to pre-process the certificate off-chain, for example to remove irrelevant information and relieve the blockchain from unnecessary computation.

The next step is the parsing of the ASN.1 structure of the DER encoding. ASN.1 is a formal language used to specify type and format of exchanged information [36]. The structure and content of all X.509 artifacts, such as certificates, CRLs, OCSP messages, public keys, are defined using ASN.1 in the respective RFC [16][41]. As an example, Listing 6.3 displays the ASN.1 definition for the structure of public keys as it is used in X.509 certificates: Public key information is defined as a sequence of two components: The cryptographic algorithm that the key is for (line 2) and the public key itself (line 3) . The algorithm is specified with the `AlgorithmIdentifier` type, which is a sequence of the algorithm OID and optional algorithm parameters (lines 5-7). In our implementation, `X509Parser` implements the parsing logic for X.509 artifacts relying on `ASN1Parser`, which implements general functionality for traversing ASN.1 documents.

`X509Parser` first divides the certificate in the TBSCertificate, the signature information, and the signature (cf. section 2.2.1) and validates the signature. If valid, the TBSCertificate is traversed from top to bottom, starting from the version field and ending with the extensions. All relevant fields are copied into a new `X509Certificate` struct, which is described below. Furthermore, the routine validates that the issuer field matches the subject field of the issuing certificate, that the certificate is currently valid, and does not contain any extension that is marked as critical and of a unknown type. If any of these test is not passed, the transaction is reverted with an error message and consequently, the certificate cannot be added to the database. The parsing and validation of OCSP and CRLs is performed very similarly as described here for certificates.

Cryptographic Algorithms Supported

One of the most compute-intensive tasks during certificate validation is the verification of cryptographic signatures. In section 3.4.2 we evaluate which cryptographic encryption and hash algorithms are currently in use for TLS certificates. For our prototype we implement a subset of these algorithms that is already available in Ethereum or have been implemented by third parties.

Ethereum provides a precompiled contract at address 0x02 that computes the SHA-256 hash of a byte array. For SHA-1, we use a third party library[1]. Since the Byzantium fork, the precompiled contract at address 0x05 performs modular exponentiation which is the foundation of RSA algorithms. Furthermore, as Ethereum signatures are created using elliptic curve cryptography, it provides a cost-efficient function for verifying ECDSA signatures. Unfortunately, Ethereum uses the secp256k1 curve, which is not used for X.509 certificates. Other curves are not natively supported by Ethereum. Consequently, we can validate RSA-SHA1 and RSA-SHA256 signatures which amount to over 84% of signatures that we observed in our analysis in section 3.4.2. For a nearly full coverage of certificate signatures, it is advisable to integrate the relevant ECDSA curves and SHA-384 to the implementation in the future.

Supported Extensions

To keep the cost of verifying certificates small, we decided to only parse the most relevant extensions. Other extensions are either ignored or, when marked as critical, lead to the rejection of the certificate. Among other things, this keeps our system from accepting precertificates created for certificate transparency, as they contain the critical "CT poison" extension. The risk of supporting only few extensions is that some certificates might be rejected even if they would normally be considered valid. However, our evaluation in section 7.1 shows that this is not a problem for the certificates of popular websites that we test.

Our prototype supports the following extensions:

- Key usage: Indicates actions that the certificate key can be used for. Important for our implementation to decide whether a certificate is a CA certificate.
- Basic constraints: Indicates whether certificate is a CA certificate or and may define the maximum path length of a certificate chain.
- Subject alternative name: In combination with the "name" field, this extension contains all domains that the certificate is for. Processing this field is important as endorsements may be issued for any of the domains that the certificate contains.

In the future, it might be necessary to assess if there exist other types of extensions whose value when adding them to our system justifies the additional processing and

[1] https://github.com/ensdomains/solsha1, accessed 01.05.2020

storage cost. One extension type that we deem a sensible candidate is the "certificate policies" extension, which can be used to infer whether the certificate is an EV certificate together with some other factors.

Internal Certificate Representation

Internally, certificates are not processed and stored in their original format, but in the struct `X509Certificate` that is defined in the `X509Parser` library. This allows for a faster processing of the stored certificates, allows that irrelevant data is discarded and which would otherwise needed to be stored in the persistent Ethereum storage, and facilitates adding additional associated data which is not stored in certificates, such as the certificates revocation status.

The fields of the `X509Certificate` struct are listed in Table 6.1. The first part of the table are fields that must be specified by every certificate and are, apart from the *issuer* field, directly extracted from the submitted certificate. The issuer information is verified on validation, but not stored with the subject certificate. Instead, the unique identifier of the issuer certificate is stored by which the issuer certificate can be retrieved from the database at any time. The times defining the validity period are converted from the `GeneralizedTime` or `UTCTime` formats to a timestamp, which can then be easily compared to the current block timestamp. Subject name and public key are stored exactly as in the DER-encoded certificate and not further converted. This allows for simple data structures while preserving all information. It is the responsibility of the processing party to interpret the information correctly.

The second part of the table are the values of extensions fields that are relevant to our system. As these values are not necessarily contained by a certificate, they are initialized with appropriate default values and updated if specified by the certificate. In the future, this part may be extended to other extension types to support more diverse applications. Now, however, we resort to only supporting the extensions that are essential for the functioning of our system: The "Subject Alternative Name", short *san*, is important as it often contains many additional web domains the certificate is valid for. The *cA* and *pathLength* fields are important to distinguish CA certificates and for checking that a newly submitted certificate was indeed issued by a CA certificate within a valid path length. The fields in the third part of the table determine the OCSP and CRL revocation status of the certificate. Upon the submission of the certificate, the status for both types is set to *unknown*. Subsequently, the status can be changed to *not revoked* or *revoked* upon presentation of a valid OCSP response or CRL, respectively. However, once a status is set to *revoked*, it cannot be modified anymore.

Name	Type	Description
version	uint8	Version of certificate, currently only version 3 certificates are accepted.
serialNumber	bytes	Serial number of certificate, stored for validating the revocation status.
issuer	bytes32	Unique identifier of the issuer certificate. Does not store the issuer name, as this information is available in the issuer certificate.
notValidBefore	uint256	Start of certificate validity represented as timestamp.
notValidAfter	uint256	Expiration date of the certificate.
subjectName	bytes	Name of the subject, represented as it is in the original certificate as DER-encoded ASN.1 structure.
subjectPublicKey	bytes	Public key of the subject, represented as it is in the original certificate as DER-encoded ASN.1 structure. This also includes the information about the algorithmic public key type.
san	bytes	Subject Alternative Name, represented as it is in the original certificate as DER-encoded ASN.1 structure. Set when the respective extension is present, otherwise an empty byte array.
cA	bool	Indicates whether the certificate is a CA certificate. Default value *false*.
pathLength	uint	Indicates the maximal length of the certificate chain starting from this certificate, may only be set when cA is true.
ocsp_status	uint8	OCSP revocation status of the certificate. 0 := *not revoked*, 1 := *revoked*, 2 := *unknown*. Initial value is 2.
ocsp_lastUpdate	uint256	`lastUpdate` field of the last seen OCSP response, represented as timestamp.
ocsp_nextUpdate	uint256	`nextUpdate` field of the last seen OCSP response, represented as timestamp.
crl_status	uint8	CRL revocation status of the certificate. 0 := *not revoked*, 1 := *revoked*, 2 := *unknown*. Initial value is 2.
crl_lastUpdate	uint256	`lastUpdate` field of the last seen CRL response, represented as timestamp.
crl_nextUpdate	uint256	`nextUpdate` field of the last seen CRL, represented as timestamp.

Table 6.1: Fields of the implemented X509Certificate structure.

6.3 Endorsement Framework

We use the term "endorsement framework" to describe the functionality that enables the creation, submission, and validation of internal and external endorsements. While certificates can be submitted by anyone, an endorsement can only be created by the certificate owner as it requires knowledge of the private key. This knowledge is proved by presenting the valid signature that is part of the endorsement as defined in section 5.1.2. Furthermore, an endorsement can only be created for domain names that are specified in the referenced certificate. This is ensured by the verifying party, who confirms that the claimed domain name is part of the certificate. This restriction does not hold for account addresses: With an external endorsement, any certificate owner can create an endorsement for any account address. For internal endorsements, this can be resolved by only allowing the owner of a certificate to add an endorsement. However, as verifiers might not be able to duly verify the correct implementation of this restriction, endorsements cannot be trusted in situations where the endorsement creator could profit from creating a fraudulent endorsement. This applies, for example, when the endorser wants to prove that they control a certain amount of funds contained by an Ethereum account. Currently, our system only supports endorsements created with RSA-SHA-256.

Aside from a reference implementation for internal endorsements and verifiers, our framework includes an endorsement database. This database serves several purposes: It allows to store external endorsements, enables the search for endorsement by domain or account address, and offers a validation functionality for internal endorsements that verifiers can use. This database is implemented in the `EndorsementDatabase` contract and relies on an instance of `CertificateStore` for certificate information. In the following, we briefly summarize our implementation and features of the contracts `EndorsementDatabase` and `InternallyEndorsed`.

6.3.1 External Endorsement Database

As shown in Figure 6.1, the main interface to our system for verifiers is the contract `EndorsementDatabase`. Certificate owners can provide endorsement information while verifiers can refer to this contract to verify endorsements, search for endorsed contracts of a domain, or inquire whether a certain address is endorsed externally. The shortened code of the `EndorsementDatabase` implementation is presented in Listing 6.4. The endorsement database does not perform the maintenance and validation of certificates by itself, but does rely on an instance of `CertificateStore` for this purpose. The address of the `CertificateStore` instance is passed as argument in the constructor and is immutably stored in the `cs` field.

```
1  pragma solidity ^0.5.12;
2  import "./X509Parser.sol";
3
4  contract EndorsementDatabase {
5  event EndorsementAdded(bytes20 account, string domain, bytes32 certID,
       uint256 expiration);
6  event EndorsementRevoked(bytes account, string domain, bytes32 certID,
       uint256 expiration);
7  event rootAddedtoStore(uint256 storeIndex, bytes32 fingerprint);
8
9  struct EndorsementStore {
10     mapping (uint256 => uint256) endorsements;
11     uint256 count;
12 }
13 struct Endorsement {
14     bytes20 account; string domain; bytes32 certID;
15     bytes32 rootID; uint256 addedAt; uint256 expiration;
16 }
17 struct RootStore {
18     mapping (uint256 => bytes32) roots;
19     mapping (bytes32 => bool) contained;
20     uint256 count;  address owner;
21 }
22
23 CertStore private cs;
24 mapping (uint256 => Endorsement) private endorsements;
25 uint256 private endorsementCounter = 0;
26 mapping (uint256 => RootStore) private rootStores;
27 uint256 private rootStoreCount = 0;
28 mapping (string => EndorsementStore) private endorsementsByDomain;
29 mapping (bytes20 => EndorsementStore) private endorsementsByAddress;
30
31 constructor (address certStore) public{
32     cs = CertStore(certStore);
33 }
34
35 function submitEndorsement(bytes20 account, string memory domain, bytes32
       certID, uint256 expiration, bytes memory signature) public {...}
36 function revokeEndorsement(bytes20 account, string memory domain, bytes32
       certID, uint256 expiration, bytes memory signature) public {...}
37 function verifyInternalEndorsement(uint256 rootStore, bytes20 account,
       string memory domain, bytes32 certID, uint256 expiration, bytes memory
       signature) public view returns (bool){...}
38 function initializeRootStore() public returns (uint256) {...}
39 function addRootsToStore(uint256 storeIndex, uint256 startIndex, uint256
       endIndex) public {...}
40 function removeRootFromStore(uint256 storeIndex, uint256 rootIndex) public
       {...}
41 /* private and getter methods omitted */
42 }
```

Listing 6.4: Extract from the EndorsementDatabase implementation.

Submission of Endorsements

An external endorsement can be submitted through the public `submitEndorsement()` (line 35) function. This function takes all fields that are part of an endorsement as input. The routine retrieves the public key and the root identifier of the certificate identified by `certID` from `cs`. If the certificate is not present in `cs`, the submission is rejected. The public key is then used to verify the provided endorsement signature. If the signature is valid, a new `Endorsement` struct is created that contains the endorsement information. In addition to the fields demanded in section 5.1.2, we also store the root identifier to ease the retrieval of endorsements as described below. We do not store the signature, as it already has been used to verify the integrity of the provided information and does not serve any further purpose.

All endorsements are stored in the `endorsements` mapping (line 24) indexed by the order that they have been submitted. Furthermore, there exists an `EndorsementStore` for every domain and every account address, which are referenced in the mappings `endorsementsByDomain` (line 28) and `endorsementsByAccount` (line 29), respectively. Each of these stores contains references to all endorsements for the respective domain. For example, the mapping "`endorsementsByDomain["example.com"].endorsements`" enumerates all endorsements that have been submitted for the domain "example.com". This feature allows users to search for endorsements for specific domains or accounts.

Retrieval of Endorsements

The on-chain retrieval of endorsements is a more complex matter as it may seem to be on first look. Several endorsements may exist for one domain or one account, but not all of them might stem from a certificate chain anchored in a root trusted by the requesting user. For implementing a "look-up service", we have two options:

1. The look-up routine returns all endorsements, or its position in `endorsements`, to the requesting party. This party then goes through all endorsements and decides whether it trusts one of them.
2. The requesting user passes their set of trusted roots to the look-up service. The look-up service then goes through all relevant endorsements and returns either a trusted endorsement or plainly the information that a trusted endorsements exists.

Both approaches are limited by the type and number of parameters that can serve as input to Solidity functions and can be returned: For public functions, it is not possible to pass structs or dynamically sized arrays. Therefore, we decided to implement a variant of option 2.

We introduce so-called `RootStores` (line 17-21). As their name suggests, these are structs that contain the identifiers of trusted root certificates. The trusted certificates are stored in two mappings: `roots` (line 18), which serves the purpose of being able to quickly

identify all roots contained, and `contained` (line 19), which serves the purpose to quickly find out whether a certain root is part of the root store or not. Anybody can initialize a `RootStore` and is subsequently its owner (line 38). The owner of a `RootStore` can both add and remove roots (lines 39 and 40). When a user wants to retrieve an endorsement, they specify the identifier of a root store and the look-up routine can efficiently check whether an endorsement's `rootID` is contained in it. This approach has also another advantage: Users can use root stores that are curated by other entities and do not need to put effort in maintaining their own. For example, the Mozilla Foundation could claim a root store, publicly announce its index and keep it in sync with the Mozilla NSS root store. Malicious actions can be detected by entities monitoring the events that are emitted when a root is added to a root store.

Revocation of Endorsements

External endorsements can be revoked by updating their revocation flag. For this purpose, the corresponding certificate owner can create a "revocation signature" which has the following format:

$$Rev = sign(hash(addr|ID_{domain}|date_{exp}|0xFFFFFFFFFFFFFFFF), key_{priv})$$

This revocation information is submitted to `EndorsementStore` through the `revoke-Endorsement()` function. This function verifies the correctness of the provided signature and, if the signature is valid, marks the endorsement as revoked.

Utility Functions for Verifiers

`EndorsementDatabse` provides several utility functions for verifiers. This includes the `verifyInternalEndorsement()` function (line 37) which can be used to verify internal endorsements that are not present in the database. Furthermore, `EndorsementDatabase` provides numerous getter functions with which endorsements and meta-information can be retrieved.

6.3.2 Internal Endorsement Contract

In contrast to the contracts presented above, the `InternallyEndorsed` contract specification is to be deployed multiple times and not only once. Every contract that is supposed to be internally endorsed should inherit from this contract. The shortened code of the contract specification is shown in Listing 6.5.

Every contract that supports the specification contains fields for the endorsement fields, including the signature and excluding the account address (lines 8 -11). This is because the account address can be retrieved though functions offered by the EVM and the signature is only verified when a verifier retrieves the endorsement – in contrast to

```
1  pragma solidity ^0.5.12;
2
3  import "./Owned.sol";
4  import "./EndorsementStore.sol";
5
6  contract InternallyEndorsed is Owned {
7
8  bytes private signature;
9  string private domain;
10 bytes32 private certID;
11 uint256 private expiration;
12 mapping (bytes32 => bool) private allCertIDs;
13 mapping (string => mapping(bytes32 => mapping (uint256 => bool))) private
       revoked;
14 EndorsementDatabase private es;
15
16 constructor (address endorsementStore) public{
17     es = EndorsementDatabase(endorsementStore);
18 }
19
20 function updateEndorsement(bytes memory _signature, string memory _domain,
       bytes32 _certID, uint256 _expiration) public onlyOwner{...}
21
22 function revokeEndorsement(bytes memory _signature, string memory _domain,
       bytes32 _certID, uint256 _expiration) public {
23    if (allCertIDs[certID]){
24       if (es.revokeInternalEndorsement(toBytes(address(this)), _domain,
25             _certID, _expiration, _signature)){
26          revoked[_domain][_certID][_expiration] = true;}}}
27 }
```

Listing 6.5: Shortened code of the `InternallyEndorsed` contract specification.

external endorsements, where the signature is verified upon signature submission. The `updateEndorsement()` function (line 20) can only be called by the contract owner.

In contrast to this, the revocation of an endorsement can be submitted by anyone that knows the relevant private key. This is to cover scenarios in which an entity looses control of a contract. To avoid spam, we put the following protection mechanism into place: Every internally endorsed contract contains the mapping `allCertIDs` (line 12). This mapping is true for all certificate identifiers that were contained in endorsements submitted to this contract. The first step of the function `revokeEndorsement()` (line 22) is to ensure that the revocation signature was created with a certificate that was previously used to create an endorsement, i.e. that was owned by a previous or current owner of the contract. If this is the case, the `EndorsementStore` instance is called to verify the signature. If the signature is valid, the multidimensional mapping `revoked` (line 13) is set to true for the combination of domain name, certificate identifier, and expiration date. This mechanism ensures that only a very small number of entities can update the revocation status of a certificate.

7 Evaluation

In this chapter we evaluate the design and implementation of our system. We test our Solidity validation routine for certificates and measure the cost that is associated with different actions in our system in sections 7.1 and 7.2. Furthermore, we discuss some security concerns in section 7.3. Finally, we formulate concise answers to our research questions listed in section 1.2 with the knowledge obtained throughout the duration of this work in section 7.4.

7.1 Compatibility

To test the compatibility of our system and to measure its performance with real and commonly used certificates, we create a test data set that contains the certificates of the 1,000 most-visited domains on the 30th of April, 2020. Out of the 1000 domains, 869 domains serve certificates that are valid in regards with the criteria that we define in section 3.4.1. As some domains share one certificate, the total amount of valid certificates amounts to 779 certificates. Out of this set, we remove certificates whose certificate chain contains signatures that are using unsupported algorithms such as ECDSA or SHA-384. Our final testing set is comprised of 576 certificates that serve 660 different domains, in addition to 47 intermediate and 21 root certificates that are required for valid trust chains. This means that our testing set contains 644 certificates in total. We refer to this set as *reference data set*.

We create a fresh instance of `CertificateStore` on our test blockchain and consecutively add all root, intermediate, and domain certificates. All certificates are accepted as valid and added to the database. This complies with the desired behavior, as we have only included valid certificates in this test data set. Furthermore, this hints that our worries discussed in section 6.2.2 are unnecessary: Not one of the certificates contains a critical extension that our validation routine does not support. Considering the nature of our data set, this is a good indicator that special critical extensions are uncommon for TLS certificates and that our implementation is compatible with most certificates.

In addition to our reference dataset we also tested our implementation with a variety of invalid certificates. This test included expired, wrongly signed, tampered, and formally incorrect certificates and such with mismatching issuer information and inconsistent signature information. The submission attempt of all invalid test certificates led to a rejection of the certificate. It is important to note that due to the prototype nature of the

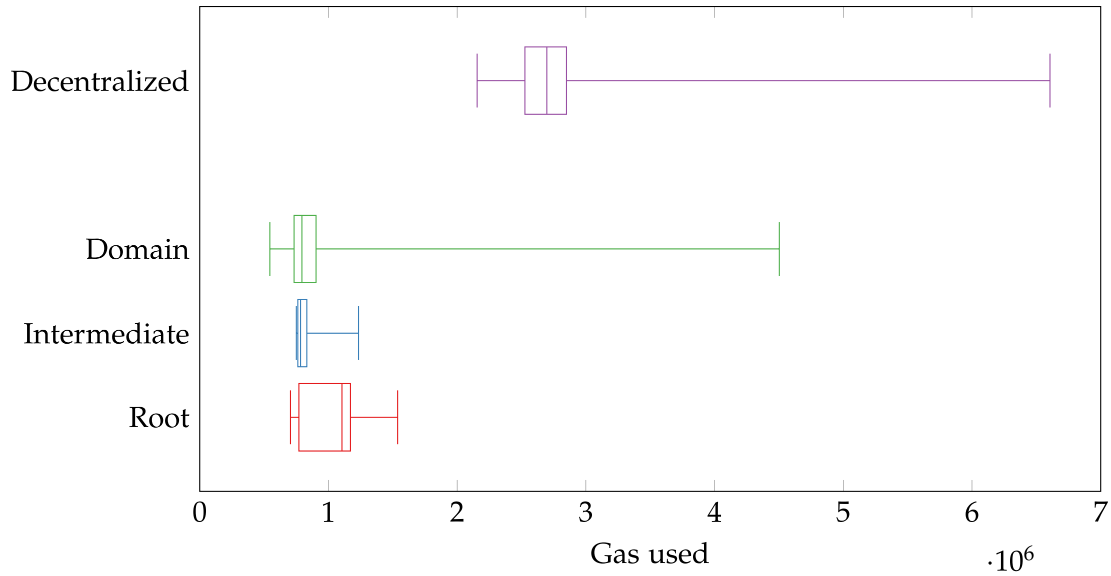


Figure 7.1: Amount of gas used for the submission of root, intermediate, and domain certificates in the reference data set. The graph displays the minimum, median, and maximum value and the first and third quartile of each set. For comparison, the cost for adding domain certificates in a decentralized scenario is shown.

implementation and the limited time scope of this work, these tests were not exhaustive. Due to the security-critical role of the certificate validation, further testing is necessary before deployment.

7.2 Costs and Performance

The performance of our Ethereum application is a fundamental part, as using our system does not only require time and electricity, but requires paying actual funds in the form of gas. We acknowledge that the usage cost is an important factor to the success and viability of our system in requirement R9 and demand cost-efficiency especially for the verifier. To gain a perspective on the cost to be expected, we once again consider the reference dataset.

Certificates

We submit all certificates in this set with one certificate per transaction. Figure 7.1 displays the observed gas usage by transaction, grouped by root, intermediate, and domain certificates. Concrete numbers characterizing the distribution are also presented in Table 7.1. We encounter the following situation: The median value of the root

certificates is the highest. We conclude that this is the case because the majority of root certificate is self-signed using SHA-1, which costs significantly more than SHA-256. The cost for intermediate certificates is quite homogeneous, with some outliers that are signed using SHA-1. For domain certificates the submission cost differs significantly. As domain certificates are commonly issued using the up-to-date SHA-256, the choice of algorithm is not the source of this circumstance. Instead, the reason of this occurrence is the size of domain certificates, especially the number of subject alternative names it specifies. The larger a certificate, the more it costs to parse and validate it, and the larger the SAN field, the more gas is payed for writing it to storage. The most costly certificate specifies 225 subject alternative names.

To show the value of having a centralized database with which root and intermediate certificates can be reused, we calculate the cost of a decentralized approach. We model the situation as follows: For every submitted domain certificate, the root and intermediate certificate need to be submitted and validated anew. This corresponds to summarizing the cost of adding the root, the intermediate, and the domain certificate as we observed them for our reference dataset. We plot the distribution of the obtained costs for comparison in Figure 7.1.

To get an idea of the average savings per certificate, we compute the total cost of both approaches and divide them by the number of domain certificates. The total gas used for approach 1 amounts to 597,912,649. Divided by 576 domain certificates, this corresponds to 1,038,042 gas per certificate. For approach two, the values are 1,590,936,703 and 2,762,042, respectively. The cost per certificate is 2.5 times higher for the decentralized approach than for the centralized approach. Asymptotically, the costs for the centralized approach approximates the average cost of adding a domain certificate, while the decentralized cost approximates the average cost of adding a root, an intermediate, and a domain certificate. Considering our observations from section 3.4.1, we expect that the cost of both approaches diverges even more when a larger number of domain certificates is submitted.

Finally, we set the gas usage in relation to the amount of ether and US dollar used. For this purpose, we assume a gas fee of 11.1 Gwei and a conversion rate of 206 US dollar per ether, as observed on the 30th of April 2020 [15]. We convert the minimum, median, and maximum value, as well as the first and third quartile for the certificate groups that we evaluated. We present the results in Table 7.1.

In section 3.4.1, we found out that by adding 13 root and 24 intermediate certificates, we can cover 98% of all certificates; when excluding Let's Encrypt certificates, it would be 12 root certificates and 23 intermediate certificates for covering 96% of domain certificates. Calculating with an average gas usage of 1,041,580 for submitting a root certificate and 825,926 for submitting an intermediate certificate, an initial investment of $(1,041,580 \cdot 13) + (825,926 \cdot 24) = 33,362,764$ gas (equivalent to 75.60 $) would mean that 98% of certificates can be added with only incurring cost for the domain certificate

	Root certificate			Intermediate certificate			Domain certificate		
	gas	ether	$	gas	ether	$	gas	ether	$
min	705,035	0.0078	1.60	750,584	0.0083	1.70	544,777	0.0060	1.23
1st	770,455	0.0086	1.77	762,129	0.0085	1.75	733,073	0.0081	1.66
med	1,105,114	0.0123	2.53	783,324	0.0087	1.79	793,954	0.0088	1.81
3rd	1,170,981	0.0130	2.67	832,031	0.0092	1.89	903,813	0.0100	2.06
max	1,537,513	0.0171	3.52	1,233,724	0.0137	2.82	4,503,213	0.0500	10.3

Table 7.1: Cost of certificate submission in gas usage, ether, and US dollar.

submission. Without Let's Encrypt, it would be $(1,041,580 \cdot 12) + (825,926 \cdot 23) = 31,495,258$ gas (equivalent to 71.37$) for 96% of certificates.

Another point that needs consideration is the cost for updating the revocation status of a certificate. Unfortunately, OCSP responders seem to use SHA-1 for hashing the issuer identifier. As this hash needs to be performed multiple times while processing an OCSP response, the cost for updating the revocation status are, in comparison to the costs for the initial submission, quite high. With our sample test cases, we observed a gas usage of approximately 733,100, equivalent to a cost of 1.67 $.

Endorsements

The cost of adding an endorsement does not fluctuate as much as for certificates as the only one signature algorithm is used (RSA-SHA256) and endorsements are constant in size except for the length of the domain name. For submitting an endorsement to the external database, we measure a cost of around 577,219 gas (1.32 $). The gas usage of retrieving the endorsement information depends on the method which is used:

- Endorsement by index for all endorsements: 32372 gas (0.07 $)
- Endorsement by address and index in specific endorsement store: 35441 gas (0.08$)
- Endorsement by domain and index in specific endorsement store: 35373 gas (0.08$)
- Address by domain and root store: 30550 gas (0.07 $)
- Domain by address and root store: 30546 gas (0.07 $)
- Endorsement by domain, address, and root store: 43364 gas (0.10 $)

Overall, with the external endorsement database, we minimize the cost of the recurring action, which is authenticating an account by obtaining its endorsement.

For internal endorsements, the absolute cost is higher. For adding the endorsement to a contract we observe a cost of approximately 446,900 gas (1.02$). At this point, the signature is not validated. The validation needs to be performed by the verifier. The retrieval and validation of an internal endorsement comes at a cost of approximately

446,800 gas (1.02$). Considering this cost occurs upon every retrieval of an endorsement, it is quite high. Therefore, we propose to adapt the internal endorsement scheme in the future, such that the signature does not need to be stored anymore and the signature validation needs to be performed only once. One way to achieve this would be to store also internal endorsements in the endorsement database and merely embed their index in the endorsed certificate. As shown above, the retrieval of an endorsement by its index is significantly cheaper, and the only additional cost the verifier bears is matching the domain name and the account address.

7.3 Security Considerations

The security of our system relies on three pillars: (i) the implementation of the certificate validation routine and the databases, (ii) the integrity of the TLS system and its certificate authorities, and (iii) the ability of users to map domain names to real-world identities. We briefly discuss these three aspects in this section.

7.3.1 Security of the Certificate and Endorsement Frameworks

We purposefully designed and implemented our system in a way that does not give one or a number of entities privileges for the system. Once the system is deployed, it is an immutable piece of code. On the one side, this means that our system cannot be subject to any kind of censorship and or can be influenced by an authorized party. On the other side, this means that errors and vulnerabilities cannot be patched. Therefore, the system must be crafted cautiously.

In the past, the validation of TLS certificates has been a troublesome topic: Many TLS certificate verifier applications have been shown to have critcal flaws that lead to invalid certificates being accepted. We aim to minimize the possibility of such critical flaws with two methods. Firstly, as described in section 6.2.2 we keep the capability of our validation routine purposefully small and support only the most important extension types. Less functionality means less surface for errors and attack vectors. Secondly, we research errors made in past verifier applications so that we can avoid traps that are easy to fall for. Some of them are accepting X.509 version 1 certificates uniformly with CA capabilities, not checking the allowed path length for CA certificates, mishandling the matching of domain names especially in the case of wildcard certificates, and accepting certificates with unknown critical extensions [1] [12] [33] . We made sure that our implementation does not repeat these well-known mistakes. However, this is no guarantee for correctness and code audits and further testing should be performed before the system is deployed.

For the case that such a critical flaw is detected, we propose the following approach: An updated and patched contract is created and deployed to Ethereum. Verifiers are

advised to refer to the new database when verifying certificates and endorsements. New endorsements and certificates are also to be submitted to the new contract. Instead of resubmitting old endorsements and contracts however, the patched contract refers to the vulnerable contract and considers all information that was submitted before a certain cut-off date. This approach can also be taken when there are significant changes to the X.509 specification or new essential functionality is to be added to the system.

Another concern is that the open trust model, where anybody can submit a root anchor, might lead to the system being spammed with "useless" certificates and endorsements. Useless means that they do not stem from a trust anchor that is included in an actively used root store and that their purpose is not to enable authentication, but to keep others from using the system. An attacker can add its own trust anchor, and produce a large number of valid domain certificates end endorsements that are valid in regards to it. We are not concerned about the total number of submissions – with 2^{256} open positions for both certificate and endorsements, it is unrealistic that the storage will ever run out of space. However, an attacker might want to hinder a certain certificate from being added by occupying its position. As we chose to use a certificate's SHA-256 hash as position index and due to its properties as cryptographic hash, a computationally-bounded attacker will not be able to produce a different valid certificate with the same position index. A more worrying concern is that an attacker might be able to populate the data structures that are in place for the fast retrieval of endorsements with fraudulent endorsements. As currently endorsements are traversed one-by-one until an endorsement stemming from a trusted root is found, this could increase the cost significantly for the on-chain retrieval of external endorsements. To circumvent this, we propose to either rely on a combination of off-chain search and on-chain retrieval, or limit the number of endorsements originating from the same root per domain and per address. To make the second approach feasible, the cost of adding a root certificate would need to be elevated. Overall, the cost of actions must be balanced in a way that incentivizes benign users to participate and that deincentivizes attackers to spam the system.

7.3.2 Security of the TLS Ecosystem

In the past, the TLS PKI has been under criticism as all trust is transferred to CAs, which makes them a single point of failure, and CA misbehavior has not been unobserved in the past. However, as the TLS system is widely adopted and "too big to fail", in the past a lot of considerations have been made to improve its security. For example, with the introduction of CT [10], a large step has been made towards the transparency of the TLS PKI and the issuance processes of CAs. It is no longer possible for a CA to issue a fraudulent certificate undetected. Furthermore, due to its wide deployment, the TLS is thoroughly investigated by security researchers in the past and in the present. A system that is set-up newly does not profit from these efforts but still requires trust anchors for bootstrapping and endorsing identity information. Furthermore, users have access to a

wide array of publicly available information. For example, they do not have to assess the trustworthiness of certificate issuers themselves, but can rely on judgments made by organizations such as the CAB forum.

A major advantage of our system in comparison to the internet ecosystem is that all certificates must be publicly submitted and are stored for anyone to see. This way, rogue certificates can be quickly identified. On the internet, the rogue certificate might only be presented to one victim, which is not able to identify it as fraudulent. This property of our system is very similar to the certificate transparency method that has been introduced.

Another important point that needs to be considered is the security of private keys or, to be more precise, the actions that are performed with them. It would be fatal for our endorsement scheme if the owner can be tricked into unknowingly signing an endorsement during a regular TLS handshake. Fortunately, by design, such handshake protocols do not require the key owner to sign information provided by the other party. Instead the private key is either used for decrypting information by provided by the other party or by signing values chosen by the private key owner [40].

7.3.3 Mapping Domain Names to Real-world Identities

The foundational assumption of using TLS certificates for an authentication framework is that domain names can be linked reliably to real-world identities. This assumes that users have the ability and knowledge to connect a domain name to an organization or person and vice versa. Usually, this is the case as users have experience with using domain names on the internet and as domain names are constructed to be human-friendly, for example by consisting of the company name.

One threat to this approach is *typosquatting*, the intentional registration of slight misspelling of well-known domain names [45]. While these domains are often used to display advertisements on the web [34], they pose a risk to our system. An attacker might use a typosquatting domain and trick users into using their similar domain or count on users accidentally misspelling a domain. For an automated authentication decision, this can have disastrous consequences. However, we deem the chance of mistyping an Ethereum address higher and the use of domain names as identifying information more reliable. In any case, users can rely on the features of higher grade – organizational validation and extended validation – certificates for more security relevant applications. These types of certificates are not merely based on the existence of a domain name, but also ensure that a respective organization exists.

7.4 Research Questions

Finally, we review the research questions that we phrased at the beginning of our work and that guided our process. In this section, we will formulate answers to this question. These answers represent a distillation of the main concepts and results that are the outcome of this theses.

1. **How can we enable on-chain decisions on identity using SSL/TLS certificates?**

   a) What are possibilities to provide determinism for the validity decision?

      We identified two main approaches to provide determinism: either relying on oracle services that retrieve and provide information as it is needed, or by storing all required information on-chain. The first approach provides determinism as an external party decides on the validity and announces the result, but it does not guarantee availability, as it depends on two off-chain components: the oracle service and the web server. As all information is available on-chain with the second approach, it enables deterministic, policy-based decisions while also guaranteeing availability.

   b) What are the associated costs of the approaches?

      With the oracle approach, the cost for on-chain computations is low. However, it incurs off-chain costs by either operating an oracle service or by paying fees to use one. The migration approach incurs on-chain cost for validating and storing certificates.

   c) How can certificates be revoked on-chain?

      The revocation of certificates with the on-chain approach is straight-forward: The oracle service checks the revocation status of the certificate when retrieving it from the web server. With the migration approach, we leverage that both OCSP messages and CRLs are signed with the private key of the certificate issuer. Consequently, we can submit such a document to the chain, where it is verified and subsequently used to update the certificate's status information.

   d) What are inherent problems of the SSL/TLS public key infrastructure and how can we mitigate them?

      Looking at the TLS PKI, we see two major flaws: All trust is provided by certificate authorities and the specifications are extensive and diverse, which leads to a large attack surface. While it does not mitigate the first problem explicitly, a central database that can be audited by anyone allows the detection of rogue certificates. This hinders split-world attacks, where only the victim is presented with the fraudulent certificate. To approach the second issue, we decide to implement a system that supports the minimal necessary

functionality. This might lead to the rejection of some valid certificates, but reduces the probability of accepting an invalid certificate.

Considering all points, we find that the centralized migration approach is the solution that is the most fitting for our problem. We propose a central database to which any valid certificate can be submitted. This allows the deterministic validation of certificates. Even though all computations are performed on-chain, the cost is reasonable.

2. **How can we use TLS certificates to endorse Ethereum addresses on-chain?**

   a) How can already deployed contracts and externally owned accounts be endorsed?

      As deployed smart contracts are immutable and EOAs do not support any code, the endorsement information cannot be embedded inside them. For these contracts, we propose external endorsements that can be submitted to a central database. Verifiers that want to authenticate a certain domain or account can obtain the endorsement there.

   b) How can identity endorsements be revoked?

      One possibility of revoking an endorsement is to revoke the certificate that signed it. This, however, entails that all endorsements created with this certificate are revoked and that the certificate cannot be used for other purposes anymore. Therefore, we implement a revocation mechanism that works both for internal and external endorsements. The owner of a private key can sign a revocation message that details that the endorsement characterized by a specific account address, domain name, certificate identifier, and expiration date is revoked. This information is then submitted to the endorsement database or the internally endorsed contract, as appropriate.

   c) What measures can an identity owner take to increase trust in their identity claim?

      To boost the credibility of an address-identity binding, the identity owner should pay particular attention to the quality of the information of their TLS certificate. In particular, this means that higher grade certificates such as extended validation certificates are considered more trustworthy. Additionally, the identity owner can reinforce the validity of a submitted certificate by periodically updating its revocation status.

By storing certificate information on-chain, offering support for internal and external endorsements, and allowing the individual revocation of endorsements, we provide a comprehensive framework for binding Ethereum addresses to domain names. As users can decide on a set of trusted root certificates, we enable anyone to participate with their perception of truth. The costs for setting an endorsement

up are reasonable; the verification of an endorsement is especially low-priced when using external endorsements.

8 Conclusion

In this work, we present the conceptual idea, design, and implementation of a TLS-certificate-based authentication framework for Ethereum. Such an authentication framework enables new Ethereum applications that require that at least one of the interaction partners is authenticated. In our framework, identities can be asserted and verified based on TLS certificates that are submitted to and validated by a central database. Identity owners that want to link their identity to an Ethereum account can create endorsements. An endorsement links information about the account address and the domain name, and contains a signature that was created with the certificate's private key and confirms the identity binding. Subsequently, users can obtain this endorsement to authenticate Ethereum accounts they aim to interact with.

The evaluation of the TLS ecosystem and our implementation shows that our system strongly profits from centralizing the validation, storage, and maintenance of certificates. The centralization lowers the cost for identity providers that submit certificates, as they only need to provide the domain certificate instead of the whole certificate chain. It also lowers the cost for verifiers, as the validation of certificates – which is costly due to the necessary cryptographic algorithms – is performed once when the certificate is submitted and does not need to be performed again.

The great strength of our system is that it overcomes the bootstrapping problem: Any identity owner can submit their certificate and endorsement without depending on other stakeholders. Under the assumption that certificate authorities are trusted, we can leverage a massive amount of verifiable/verified identity information that is readily available. However, we also acknowledge that our system comes with drawbacks: The TLS system is considered fragmented and not secure enough by some researchers, our system enables authentication only for certificate owners, the on-chain validation of TLS certificates is costly, and storing certificate information increases the size of the Ethereum blockchain. However, we believe that solutions or mitigations can be found to lower the negative impact of these drawbacks. Overall, our framework serves as a pragmatic and feasible approach to establish a system for the identity assertion and verification on Ethereum in a timely manner.

Future Work

One main goal of future work should be to investigate whether a TLS-certificate-based authentication framework can be used in combination with an identity management

system or naming service developed specifically for Ethereum. A combination of the approaches could utilize the strengths of both: The certificate-based approach can boost the bootstrapping phase of the system. The information acquired in the bootstrapping phase can then be used to populate the system with further, certificate-independent information. The aim is to make the system gradually independent from the TLS ecosystem, thereby improving the security of the framework. Combining the approaches would also allow the system to profit from the potential strengths of an identity system crafted for a blockchain: more efficient validation routines, more appropriate trust structures, privacy-preserving mechanisms, and the possibility to participate independent of certificate-ownership.

Apart from this, attention should be placed on the improvement and testing of the current design and implementation. This includes the addition of cryptographic algorithms like ECDSA curves P-256 and P-384, as well as the hash algorithms SHA-1 and SHA-256. Furthermore, it might be advisable to include more TLS extension types to better support applications. Since the current mechanism for internal endorsements incurs relatively high costs for authenticating an account, it is advisable to restructure the current scheme and centralize the validation of internal endorsements.

Future work should also develop a more elaborate endorsement framework which could provide two major functionalities. First, such a framework could mark endorsements with a special type. For example, if an account is endorsed with the attribute "payable", the endorsement creator expresses that funds should be sent to this account and not to other accounts that are linked to the same identity. Secondly, such a framework could support chains of endorsements, which would, for example, enable organizations or companies to endorse their members or employees on-chain. Such a feature could also be a solution approach for supporting identity types other than domain names.

List of Figures

2.1 The basic structure of a blockchain. 8
2.2 Structure of an X.509 certificate. 14
2.3 Structure of an X.509 certificate revocation list. 16
2.4 Exemplary structure of a hierarchical X.509 PKI. 18

3.1 Maximum share of certificates covered by aggregating the top x intermediate certificates. 31
3.2 Tight lower bound on achievable share of covered domain certificates when choosing x CA certificates. 31
3.3 Maximum share of certificates of domains in the Alexa list covered by aggregation of the top x intermediate certificates. 33
3.4 Tight lower bound on achievable share of covered certificates of domains in the Alexa list when choosing x CA certificates. 33

5.1 Different ways of internally or externally endorsing a smart contract. . . 45
5.2 Depiction of the workflow when using the oracle approach for certificate validation. 49

6.1 Structure of the authentication framework. 56

7.1 Amount of gas used for the submission of root, intermediate, and domain certificates in the reference data set. 72

List of Tables

3.1 Number of public key types observed. 35
3.2 Number of certificates that are signed with different signature algorithms. 35
3.3 Number of signature algorithms occurring grouped by hash and encryption algorithms used. 35

5.1 Fulfillment of relevant requirements of the different approaches to enable on-chain validation of certificates and endorsements. 52
5.2 Compliance with requirements of our design. 54

6.1 Fields of the implemented X509Certificate structure. 63

7.1 Cost of certificate submission in gas usage, ether, and US dollar. 74

Bibliography

[1] D. Akhawe, B. Amann, M. Vallentin, and R. Sommer. "Here's my cert, so trust me, maybe? Understanding TLS errors on the web." In: *Proceedings of the 22nd international conference on World Wide Web*. 2013, pp. 59–70.

[2] M. Ali, J. Nelson, R. Shea, and M. J. Freedman. "Blockstack: A global naming and storage system secured by blockchains." In: *USENIX Annual Technical Conference*. 2016, pp. 181–194.

[3] B. Amann, R. Sommer, M. Vallentin, and S. Hall. "No attack necessary: The surprising dynamics of SSL trust relationships." In: *Proceedings of the 29th annual computer security applications conference*. 2013, pp. 179–188.

[4] B. Amann, M. Vallentin, S. Hall, and R. Sommer. *Extracting certificates from live traffic: A near real-time SSL notary service*. Tech. rep. Citeseer, 2012.

[5] J. Amann, O. Gasser, Q. Scheitle, L. Brent, G. Carle, and R. Holz. "Mission accomplished? HTTPS security after DigiNotar." In: *Proceedings of the 2017 Internet Measurement Conference*. 2017, pp. 325–340.

[6] A. M. Antonopoulos and G. Wood. *Mastering ethereum: building smart contracts and dapps*. O'reilly Media, 2018.

[7] M. Al-Bassam. "SCPKI: A smart contract-based PKI and identity system." In: *Proceedings of the ACM Workshop on Blockchain, Cryptocurrencies and Contracts*. 2017, pp. 35–40.

[8] V. Buterin and A. Van de Sande. *EIP-55*. `https://github.com/ethereum/EIPs/blob/master/EIPS/eip-55.md`. 2019 (accessed May 12, 2020).

[9] *Certificate Patrol*. `http://patrol.psyced.org/`. (accessed November 26, 2019).

[10] *Certificate Transparency*. `http://www.certificate-transparency.org`. 2020 (accessed May 12, 2020).

[11] J. Chen, S. Yao, Q. Yuan, K. He, S. Ji, and R. Du. "Certchain: Public and efficient certificate audit based on blockchain for TLS connections." In: *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE. 2018, pp. 2060–2068.

[12] Y. Chen and Z. Su. "Guided differential testing of certificate validation in SSL/TLS implementations." In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 2015, pp. 793–804.

[13] J. Clark and P. C. Van Oorschot. "SoK: SSL and HTTPS: Revisiting past challenges and evaluating certificate trust model enhancements." In: *2013 IEEE Symposium on Security and Privacy*. IEEE. 2013, pp. 511–525.

[14] CoinDesk. *$7 Million Lost in CoinDash ICO Hack*. `https://www.coindesk.com/7-million-ico-hack-results-coindash-refund-offer`. 2017 (accessed May 12, 2020).

[15] CoinMarketCap. *Historical data for Ethereum*. `https://coinmarketcap.com/currencies/ethereum/historical-data/`. 2020 (accessed May 12, 2020).

[16] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, W. T. Polk, et al. *Internet X. 509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile.* RFC 5280. RFC Editor, May 2008.

[17] Z. Durumeric, D. Adrian, A. Mirian, M. Bailey, and J. A. Halderman. "A Search Engine Backed by Internet-Wide Scanning." In: *22nd ACM Conference on Computer and Communications Security*. Oct. 2015.

[18] C. Eckert. *IT-Sicherheit: Konzepte-Verfahren-Protokolle*. Walter de Gruyter, 2018.

[19] *Ethereum Name Service*. `https://ens.domains/`. 2020 (accessed May 12, 2020).

[20] C. Fromknecht, D. Velicanu, and S. Yakoubov. "A Decentralized Public Key Infrastructure with Identity Retention." In: *IACR Cryptology ePrint Archive* (2014).

[21] U. Gallersdörfer and F. Matthes. "AuthSC: Mind the Gap between Web and Smart Contracts." In: *arXiv preprint 2004.14033* (2020).

[22] A. Garba, Z. Guan, A. Li, and Z. Chen. "Analysis of Man-In-The-Middle of Attack on Bitcoin Address." In: *ICETE*. 2018.

[23] P. Hallam-Baker, R. Stradling, and J. Hoffman-Andrews. *DNS certification authority authorization (CAA) resource record*. RFC 6844. RFC Editor, Nov. 2019.

[24] M. T. Hammi, P. Bellot, and A. Serhrouchni. "BCTrust: A decentralized authentication blockchain-based mechanism." In: *2018 IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE. 2018, pp. 1–6.

[25] R. Holz, L. Braun, N. Kammenhuber, and G. Carle. "The SSL landscape: a thorough analysis of the x. 509 PKI using active and passive measurements." In: *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*. 2011, pp. 427–444.

[26] B. Khieu and M. Moh. "CBPKI: Cloud Blockchain-based Public Key Infrastructure." In: *Proceedings of the 2019 ACM Southeast Conference*. 2019, pp. 58–63.

[27] M. Y. Kubilay, M. S. Kiraz, and H. A. Mantar. "Certledger: A new PKI model with certificate transparency based on blockchain." In: *Computers & Security* 85 (2019), pp. 333–352.

[28] B. Laurie, A. Langley, and E. Kasper. *Certificate Transparency*. RFC 6962. RFC Editor, June 2013.

[29] A. K. Malik, A. Anjum, and B. Raza. *Innovative solutions for access control management*. Advances in Information Security, Privacy, and Ethics. IGI Global, 2016.

[30] S. Matsumoto and R. M. Reischuk. "IKP: Turning a PKI around with decentralized automated incentives." In: *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2017, pp. 410–426.

[31] U. Maurer. "Modelling a public-key infrastructure." In: *European Symposium on Research in Computer Security*. Springer. 1996, pp. 325–350.

[32] N. van der Meulen. "DigiNotar: Dissecting the first dutch digital disaster." In: *Journal of Strategic Security* 6.2 (2013), pp. 46–58.

[33] C. Meyer and J. Schwenk. "SoK: Lessons learned from SSL/TLS attacks." In: *International Workshop on Information Security Applications*. Springer. 2013, pp. 189–209.

[34] T. Moore and B. Edelman. "Measuring the perpetrators and funders of typosquatting." In: *International Conference on Financial Cryptography and Data Security*. Springer. 2010, pp. 175–191.

[35] S. Nakamoto. *Bitcoin: A peer-to-peer electronic cash system.* Tech. rep. 2008.

[36] G. Neufeld and S. Vuong. "An overview of ASN. 1." In: *Computer Networks and ISDN Systems* 23.5 (1992), pp. 393–415.

[37] C. Patsonakis, K. Samari, M. Roussopoulos, and A. Kiayias. "Towards a smart contract-based, decentralized, public-key infrastructure." In: *International Conference on Cryptology and Network Security*. Springer. 2017, pp. 299–321.

[38] Y. Pettersen. *The Transport Layer Security (TLS) Multiple Certificate Status Request Extension*. RFC 6961. RFC Editor, June 2013.

[39] W. Polk, R. Housley, and L. Bassham. *Algorithms and identifiers for the Internet X. 509 public key infrastructure certificate and certificate revocation list (CRL) profile*. RFC 3279. RFC Editor, Apr. 2002.

[40] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. RFC Editor, Aug. 2018.

[41] S. Santesson, M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams. *X. 509 Internet Public Key Infrastructure Online Certificate Status Protocol-OCSP.* RFC 6960. RFC Editor, June 2013.

[42] Q. Scheitle, T. Chung, J. Hiller, O. Gasser, J. Naab, R. van Rijswijk-Deij, O. Hohlfeld, R. Holz, D. Choffnes, A. Mislove, et al. "A first look at certification authority authorization (CAA)." In: *ACM SIGCOMM Computer Communication Review* 48.2 (2018), pp. 10–23.

[43] A. Singla and E. Bertino. "Blockchain-based PKI solutions for IoT." In: *2018 IEEE 4th International Conference on Collaboration and Internet Computing (CIC)*. IEEE. 2018, pp. 9–15.

[44] C. Soghoian and S. Stamm. "Certified lies: Detecting and defeating government interception attacks against SSL." In: *Proceedings of ACM Symposium on Operating Systems Principles*. 2010, pp. 1–18.

[45] J. Spaulding, S. Upadhyaya, and A. Mohaisen. "The landscape of domain name typosquatting: Techniques and countermeasures." In: *2016 11th International Conference on Availability, Reliability and Security (ARES)*. IEEE. 2016, pp. 284–289.

[46] Z. Wang, J. Lin, Q. Cai, Q. Wang, D. Zha, and J. Jing. "Blockchain-based certificate transparency and revocation transparency." In: *IEEE Transactions on Dependable and Secure Computing* (2020).

[47] G. Wood et al. "Ethereum: A secure decentralised generalised transaction ledger." In: *Ethereum project yellow paper* (2014).

[48] A. Yakubov, W. M. Shbair, A. Wallbom, D. Sanda, and R. State. "A blockchain-based PKI management framework." In: *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*. 2018.

[49] S. Yilek, E. Rescorla, H. Shacham, B. Enright, and S. Savage. "When private keys are public: Results from the 2008 Debian OpenSSL vulnerability." In: *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement*. 2009, pp. 15–27.