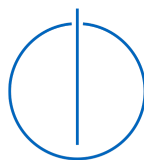
TECHNICAL UNIVERSITY OF MUNICH

DEPARTMENT OF INFORMATICS

Master's Thesis in Information Systems

Leveraging TLS/SSL-based Identity Assertion and Verification Systems for On-chain Authentication and Authorization of Real-world Entities

Jan-Niklas Strugala



TECHNICAL UNIVERSITY OF MUNICH

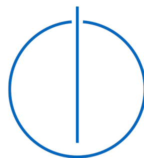
DEPARTMENT OF INFORMATICS

Master's Thesis in Information Systems

Leveraging TLS/SSL-based Identity Assertion and Verification Systems for On-chain Authentication and Authorization of Real-world Entities

Nutzung von TLS/SSL-basierten Identitätsbereitstellungs- und -Verifikationssystemen zur On-Chain Authentisierung und Autorisierung von Echtwelt-Entitäten

Author: Jan-Niklas Strugala
Supervisor: Prof. Dr. rer. nat. Florian Matthes
Advisor: Ulrich Gallersdörfer, M.Sc.
Submission Date: December 15, 2020



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

Munich, December 14, 2020 Jan-Niklas Strugala

Acknowledgments

First and foremost, I would like to thank my advisor Ulrich Gallersdörfer for his continuous support and feedback during my thesis. His input at our weekly meetings provided guidance for my research and significantly influenced its outcome. The positive and friendly working atmosphere allowed me to express and develop my thoughts and ideas freely. Furthermore, I also would like to thank my supervisor Prof. Dr. Florian Matthes for the opportunity to conduct such an interesting research project as my master's thesis at the chair of Software Engineering for Business Information Systems.

During my research I conducted interviews to evaluate my contribution. Hence I would like to thank all interview partners for taking their valuable time to provide great insights that improved my work: J. Beinke, J. Buchwald, D. Burkhardt, R. Knobloch, S. Kurrle, R. Muth, T. Paixão and H. Precht.

I would like to express gratitude to my family and friends for their support whenever it was needed during my studies and this thesis. Furthermore, I deeply appreciate that Dr. Michael Strugala always finds time to read any of my work and together with Elvira Strugala supported me during all of my life. Finally, I would like to thank Franziska Walz for proof-reading and always putting a smile on my face.

Abstract

As the popularity of blockchain systems is continuously growing and advantages of blockchain technology is common knowledge, organizations and users with new business ideas explor blockchain systems for their use-cases. However, many of these use-cases require authentication and access control of real-world entities, as only a limited group of people from a certain organization or with specific attributes should be able to use the functionality of a smart contract. Opposed to traditional systems, authentication and access control on the public blockchain still is in an early stage with few mechanisms and research available. Hence, with this work complement this research by proposing an authentication and access control mechanism at smart contracts for real-world entities.

During our research we define potential use-cases and survey existing research of traditional access control systems. Furthermore, we explore related work of authentication and access control at smart contracts. Considering the insights we design and implement an authentication and attribute based access control (ABAC) system that allows owners of smart contracts to restrict access to trusted accounts of real-world entities that hold certain attributes. As the smart contract that evaluates the access request from a real-world entity needs to trust the authenticity of attributes, we bootstrap the SSL/TLS certificate public key infrastructure to associate attributes and endow trust to accounts of real-world entities. To create this link between SSL/TLS certificates and the accounts of real-world entities we leverage the SSL/TLS-based identity assertion and verification system *On-Chain AuthSC* and create a sub-endorsement framework. Subsequently, we design the ABAC framework such that it can be implemented in any smart contract to protect its functionality. The interview-based evaluation of the system design unveils strong interest in our approach. Thus we implement and integrate the different components into a prototype for the Ethereum blockchain and conduct an analysis to evaluate its performance.

Contents

Acknowledgments **iii**

Abstract **v**

1 Introduction **1**
1.1 Problem Statement 2
1.2 Research Questions 3
1.3 Research Methodology 4
1.4 Contribution 6
1.5 Outline 7

2 Fundamentals **9**
2.1 Identity Management and the Identity Lifecycle 9
2.2 Blockchain Technology and Ethereum 10
2.2.1 Blockchain: A Peer-To-Peer Distributed Ledger 10
2.2.2 Introduction to Ethereum 12
2.2.3 Ethereum Accounts 12
2.2.4 Ethereum Virtual Machine (EVM) 14
2.2.5 Asymmetric Cryptography, Authentication, Authorization in Ethereum 14
2.3 Public Key Infrastructure (PKI) 16
2.3.1 Hierarchical PKI 16
2.3.2 Decentralized PKI 21
2.4 On-Chain AuthSC 22
2.4.1 Architecture Overview 23
2.4.2 Endorsement Framework 24
2.4.3 Certificate Framework 26

3 System Design **29**
3.1 Use-Cases 29
3.1.1 Reference Use-Case 29
3.1.2 Exemplary Use-Cases 30
3.2 Requirements 33
3.2.1 Functional Requirements 33
3.2.2 Non-Functional Requirements 34
3.3 Endorsement Framework of the Smart Contract Access Control System 36
3.3.1 Endorsement - Endorsing the Registry 37
3.3.2 Sub-Endorsement - Sub-Endorsing the User Account 40

3.4 Survey of Access Control Mechanisms . . . 41
3.4.1 Mandatory Access Control (MAC) . . . 42
3.4.2 Discretionary Access Control (DAC) . . . 44
3.4.3 Role-Based Access Control (RBAC) . . . 45
3.4.4 Attribute-Based Access Control (ABAC) . . . 47
3.5 Creating an ABAC System for Smart Contract Access Control . . . 49
3.5.1 Attributes and Policies in the ABAC System for Smart Contracts . . . 50
3.5.2 ABAC Framework of the ABAC System for Smart Contracts . . . 54
3.5.3 Application Lifecycle of the ABAC System for Smart Contracts . . . 56

4 System Implementation **59**
4.1 Endorsement Framework . . . 59
4.1.1 Sub-Endorsement Framework . . . 59
4.1.2 Configuration of the Registry Smart Contract . . . 62
4.2 ABAC Framework . . . 64
4.2.1 ABAC Prototype Architecture . . . 64
4.2.2 Application Smart Contract (PEP_{app}, PAP) . . . 68
4.2.3 PDP Library (PDP_{lib}, PEP_{lib}) . . . 70
4.2.4 Registry Smart Contract (PIP_{reg}) . . . 73
4.2.5 EndorsementDatabase Smart Contract (PIP_{edb}) . . . 74
4.2.6 CertificateStore Smart Contract and X509Parser Library (PDP_{cst}, PIP_{cst}) 75

5 System Evaluation **77**
5.1 Evaluation Approach . . . 77
5.2 Episode 1: Ex Ante Implementation Evaluation . . . 80
5.3 Episode 2: Ex Post Implementation Evaluation . . . 83
5.3.1 Performance Analysis . . . 84
5.3.2 Requirement Analysis . . . 90

6 Related Work **93**
6.1 Authentication . . . 94
6.2 Blockchain Access Control Systems . . . 94
6.2.1 Attribute-Based Access Control on Blockchain . . . 95
6.2.2 Other Access Control on Blockchain . . . 97

7 Discussion **99**
7.1 Conclusion . . . 99
7.2 Future Work . . . 102

List of Figures **105**

List of Tables **107**

Bibliography **109**

1 Introduction

In our everyday life, some resources are not intended to-be-accessed and used by everyone, hence access and further permissions sometimes need to be limited to a certain group of entities. This applies to resources in the real-world (e.g. building, substance) and to digital resources (e.g. document, application). To enforce the protection of a resource against unauthorized access and interaction, mechanisms are needed to determine and check the identity, as well as the permissions of an entity. Meaning, the owner of a to-be-protected resource needs to be able to authenticate and check the authorization (i.e. access control) of an access requesting entity.
Research is extensive for authentication and access control of digital resources in general and multiple mechanisms are available for used, however few research is concerned with protecting digital resources residing on a public blockchain system. As blockchain systems are increasingly popular among end-consumers and business organizations, we want to complement the existing research with a system that allows authentication and access control of real-world entities at digital resources on the public blockchain. More precisely, we elaborate the design and implementation of a system, that is intended for owners of digital resources on public blockchains, that want to authenticate and control access of real-world entities. In order to limit the scope, but also maximize the impact of our contribution and reach the largest possible base of potential users, we design and develop for the popular Ethereum blockchain. The to-be-protected digital resources are the applications on the Ethereum blockchain, called smart contracts.

Authentication and access control is based on the evaluation of characteristics. Hence, an access requesting real-world entity needs to show desired characteristics and provide a proof to the entity that evaluates an access request. In the real-world a common proof is a photo ID that specifies characteristics as name and age, while in digital systems often certificates are used. The trust in the authenticity of a proof is crucial, as otherwise no trust is endowed in the characteristics of a real-world entity. Therefore, such proofs are issued by generally trusted, central entities as a government or a certificate authority. However, as a public blockchain is a decentralized system, central trust providing entities are not available. Furthermore, although recent research is actively elaborating a decentralized trust providing infrastructure, no such system has been successfully established yet. Therefore, to enable authentication and access control at smart contracts on the public blockchain, we first need to create an endorsement framework that endows accounts of real-world entities with trust.

Opposed to most existing research for public blockchain systems we decide against creating a new trust infrastructure on the blockchain, but to bootstrap an existing one. This allows

us to facilitate fast adaption of our system, as we may leverage a large and proven trust infrastructure. However, as no such infrastructure exists on the Ethereum blockchain, we bootstrap the blockchain external SSL/TLS certificate public key infrastructure.
It comprises multiple tree-like hierarchies of SSL/TLS certificates, that are linked with public key cryptography. The certificates at the root of each hierarchy act as trust anchors issued by trusted central certificate authorities, while the certificates at the leaves of the hierarchy each link a public key to the Fully Qualified Domain Name (FQDN) of a website. Furthermore, SSL/TLS certificates usually contain further characteristics (i.e. attributes) that describe the organization that operates the website.
Our research leverages the SSL/TLS public key infrastructure to endow trust and associate these attributes to accounts of real-world entities on the blockchain for access control at smart contracts. To do so, we use *On-Chain AuthSC*, a SSL/TLS-based identity assertion and verification system for the Ethereum blockchain developed by Groschupp et al. in [33]. However, as *On-Chain AuthSC* only supports the endorsement of real-world entities that are owners of SSL/TLS certificates, we complement the system with a sub-endorsement framework that allows to associate the trust and attributes to every real-world entity that owns an account on the blockchain. More specifically, we introduce the concept of decentralized Registries that are endorsed with their owners SSL/TLS certificate by *On-Chain AuthSC*. At these Registries the owner of the endorsing SSL/TLS certificates can create sub-endorsements for accounts of real-world entities. These accounts can then leverage the attributes and the endowed trust of the SSL/TLS certificate for authentication and access request evaluation at a smart contract that uses the access control framework. As we bootstrap the attributes of SSL/TLS, our access control needs to evaluate access requests based on attributes. Hence, after careful evaluation we design and develop an Attribute-Based Access Control (ABAC) mechanism. An ABAC framework, that decides whether a sub-endorsed account of a real-world entity may access the functionality of a smart contract, based on the evaluation of trust and attributes of the endorsing SSL/TLS certificate.

1.1 Problem Statement

The general goal of our research is to create a system that enables authentication and access control at smart contracts. We want to allow owners of smart contracts to restrict access to a selected number of accounts from real-world entities, that are indirectly endorsed by a SSL/TLS certificate with specific attributes. To achieve our goal we have to overcome the following problems:

- **No source of trust for real-world entities**:
  Currently there is no widely trusted entity or a comparable solution that vouches for the authenticity of attributes associated to a real-world entities on a public blockchain. In order to enable authentication and access control at smart contracts we need to create trust.

- **Exclusion of real-world entities without a SSL/TLS certificate**:

On-Chain AuthSC only supports the endorsement of real-world entities that are owners of SSL/TLS certificates. To increase participation in our system, we need to extend the endorsement framework to also support endorsements for real-world entities that do not own a SSL/TLS certificate.

- **Integration of *On-Chain AuthSC***:
  As we use the standalone system *On-Chain AuthSC* within our system, we have to find a way to overcome potential integration challenges.
- **Attributes of SSL/TLS certificates are not designed for authentication and access control on the blockchain**:
  SSL/TLS certificate are not designed as a trust source and an attribute repository for real-world entities on the blockchain. We have to explore how to best access and retrieve attributes from the certificates, link them to real-world entities and evaluate them in the access control mechanism.

1.2 Research Questions

To provide guidance for our research, we define the following research questions (RQ) which we intend to answer during this work:

RQ1: Which are the major access control practices and technologies?

For a successful design and the development of a system that enforces access control at smart contracts on the blockchain the comprehension of fundamental access control mechanisms is needed. It is important to identify the most prominent mechanisms and determine if and how these can be applied on the blockchain. Especially, in due consideration of SSL/TLS certificates, as trust providing entities for the real-world entities. Furthermore, in order for our contribution to be relevant for research in blockchain, we also need to identify related work that already applies access control on the public blockchain. Hence, we pose the following two sub-questions:

1.1 Which access control practices and technologies are predominant in the literature?

1.2 Which access control practices and technologies are relevant in blockchain?

RQ2: How can a SSL/TLS-based identity assertion and verification system contribute trust to authentication and access control on the blockchain?

In our research we leverage *On-Chain AuthSC*, a SSL/TLS-based identity assertion and verification system, to expand trust to accounts of real-world entities on the blockchain. Given an endowment of trust from such a SSL/TLS certificate, accounts shall be authenticated and authorized to use services at smart contracts. In order to successfully expand trust and endow accounts, we need to understand SSL/TLS certificates and SSL/TLS-based identity assertion and verification systems. Hence, we pose the following two sub-questions:

2.1 Which of its properties endow a SSL/TLS certificate with an increased level of trust?

2.2 What are challenges of bootstrapping a SSL/TLS-based identity assertion and verification system?

RQ3: How can we achieve on-chain authentication and access control of real-world identities at smart contracts considering the constraints of blockchain?

The final goal of our work is to elaborate a system design and prototype implementation of a blockchain-based authentication and access control solution for smart contracts. In order to create a valuable design, we need to discuss and evaluate possible design choices and consider limitations of blockchain. Moreover, we need to define an application life-cycle, as well as discuss and evaluate possible choices during its implementation. Hence, we pose the following five sub-questions:

3.1 Which are the constraints of blockchain that affect the development of our solution?

3.2 What are potential system designs for a blockchain-based authentication and access control solution for smart contracts?

3.3 What are the advantages and disadvantages of the different system designs?

3.4 What is the application life-cycle of a blockchain-based authentication and access control solution for smart contracts?

3.5 How can a blockchain-based authentication and access control solution for smart contracts be implemented?

1.3 Research Methodology

To successfully design a system that meets the specific (business) needs of potential users, our research adheres to the Design Science Research methodology, a well established "[...] problem solving process" [36, pg.82] in Information Systems.
The design science research methodology aims to create an artifact, in our case an instantiation, i.e. a software implementation. It is created in a cycle of development and evaluation considering and contributing to the current knowledge base to solve a complex problem related to a business need. Since the introduction of the first specific design science research model by [36], multiple models have been discussed by researchers.

We conduct our research according to the design science research model proposed by [53] depicted in Figure 1.1. The model is structured as a six step sequential process, but does not require researchers to walk through the whole process in that specific order. Different possible entry points allow researchers to start at the process step which matches the origin that triggered their research. This could be either an identified problem, an objective, an already existing object or already a solution. Furthermore, the process is not necessarily linear.

It may contain multiple cycles that allow to repeat activities to revise the work based on input received during the Evaluation and Communication steps. In the following we traverse through the model from the entry point induced by our research origin and describe the different process steps, i.e. activities in the context of our work.

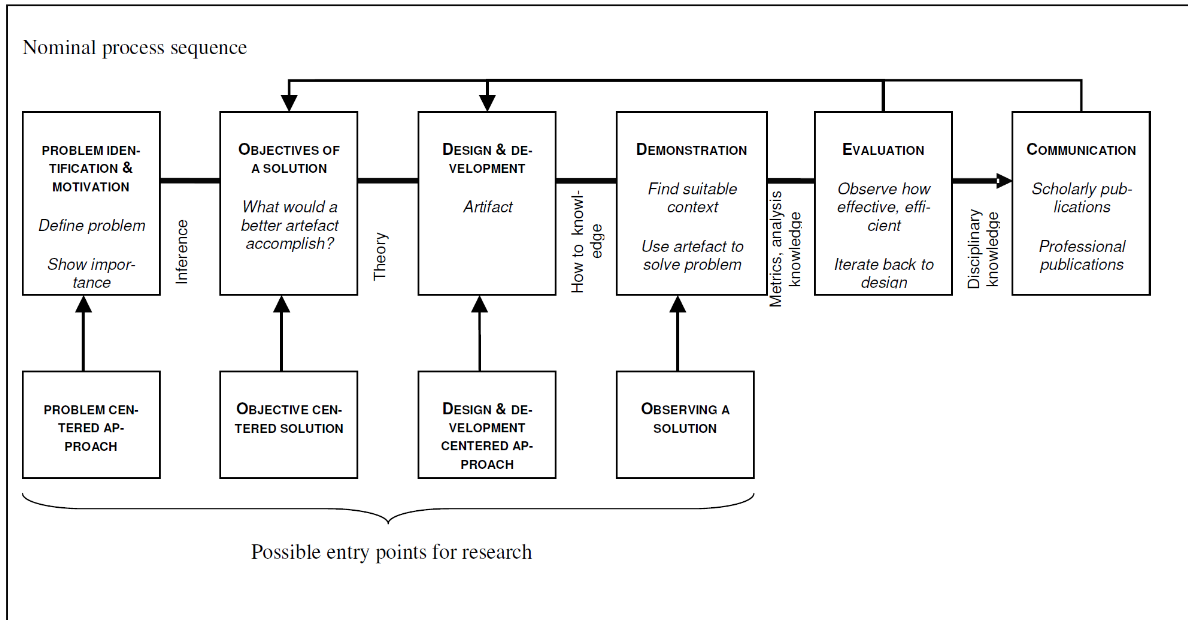


Figure 1.1: Design Science Research Process from [53, pg.93]

In their research Gallersdörfer [32] and Groschupp [33] designed the SSL/TLS-based on-chain identity assertion system *On-Chain AuthSC* and suggested future research to explore opportunities in the context of authentication and access control based on SSL/TLS certificate attributes. Since our research aims to explore these opportunities and tries to find a solution for the limited availability of access control mechanism in blockchain, we pursue the problem centered approach.

The first activity is the **Problem Identification & Motivation**, where the research problem has to be defined, the motivation be depicted and the contributions value to be justified [53]. We elaborate this step extensively in previous sections.
In the next step **Objective of a Solution** we determine the objectives of our solution, by first investigating the problem statement and conducting an analysis of *On-Chain AuthSC* and the related work in authentication and access control mechanisms for blockchain. The literature review revealed a research gap in on-chain access control mechanisms for smart contracts and on-chain identity federation. These insights allow us to define our solutions objectives among others as development of a system for ABAC at smart contracts for real-world identities, which bootstraps trust from an established trust infrastructure and is consistent with existing Ethereum authentication, access control and identity management design patterns.

Next in the the **Design & Development** step of the model the artifact is created by defining the requirements and architecture as groundwork for its development [53]. In the first iteration of the process we want to only define the requirements and a preliminary architecture of the system to evaluate it before implementation.
Therefore, in the **Demonstration** step, where "[...] the efficacy of the artifact to solve the problem" [53, pg.90] is tested, we create and conduct interviews with experts to determine how to improve the preliminary design that it can be a better foundation for the development of our artifact.
In the first iteration of the process during the **Evaluation** we only evaluate the design of our solution based on the expert interviews, since we do not have a prototype implementation yet. The insights from the interviews provide valuable feedback for our solution design.

Therefore, we iterate back to the **Design & Development** step to adjust the design accordingly and subsequently implement the artifact based on the revised requirements and architecture. This time during the **Demonstration** step of the process we deploy our prototype on the Truffle test network to see how well the access control performs under different conditions.
Next in the **Evaluation** step the goal is to determine the degree to which the artifact solves the research problem [53]. Hence, we conduct a functional and a performance analysis [36], evaluating among others the cost-efficiency of our artifact, based on data collected from the previously described deployment on the Truffle test network. Furthermore, we evaluate if the requirements which we defined in the Design & Development step are satisfied by our implementation.

The final step of the Design Science Research Model is the **Communication**, including the appropriate communication of the research problem, its solution, and the value of the contribution. Communication can be either in the context of a professional or scholarly publication [53].The communication of our work is provided through this master's thesis, the final result of the design science research for the development of a blockchain access control mechanism.

1.4 Contribution

The primary contribution of our research is a system design and a prototype implementation of a blockchain-based authentication and access control system for smart contracts that leverages trust from SSL/TLS certificates in order to authenticate and authorize blockchain accounts of real-world entities. More specifically, the accounts of real-world entities are linked to SSL/TLS certificates so that they can leverage trust and attributes of the SSL/TLS certificate to authenticate and authorize at smart contracts. In the prototype implementation of the system we provide a library, as well as reference smart contracts developed in Solidity for an easy application of our ABAC system on the Ethereum blockchain.
By developing such a novel type of ABAC system we contribute to the still relatively young blockchain authentication and access control research landscape. Opposed to existing work,

which either creates a new or does not use any source of trust, we explore trust endowment by an existing trust infrastructure from outside the blockchain. Furthermore, our system is one of the first implementations of ABAC on the Ethereum blockchain and the first implementation that leverages the *On-Chain AuthSC* identity assertion and verification system proposed by [33].

1.5 Outline

Subsequent to the **Introduction** we explain the theory that is fundamental for this research in chapter (2) **Fundamentals**. We first introduce identity management, blockchain technology and Ethereum, before we provide an overview of Public Key Infrastructure and *On-Chain AuthSC*. In chapter (3) **System Design** we present use-cases, define system requirements and conduct a survey of access control mechanisms. Furthermore, we design the architecture of our system, as well as specify its application lifecycle. The prototype implementation of our system in Solidity is specified in chapter (4) **System Implementation**. Here we explain the key components, the architecture and discuss implementation choices. In chapter (5) **System Evaluation** we assess our system based on expert interviews and a performance analysis, before we present related work for blockchain-based authentication and access control in chapter (6) **Related Work**. Finally, in chapter (7) **Discussion** we discuss our research and provide suggestions for future work.

2 Fundamentals

In this chapter we introduce the foundational concepts and theories that readers need to understand to follow our research and the development of our system. Our goal is to improve access control at smart contracts of the Ethereum blockchain, by developping an authentication and access control system for real-world entities that leverages SSL/TLS certificates attributes. Since our research is about authentication and access control of digital identities, we start with shortly describing the fundamentals of identity and the identity lifecycle in section 2.1. Moreover in this section we also provides the groundwork for the deeper analysis of access control in section 3.4. Next, we briefly introduce the concept of blockchain in section 2.2, before we explain the Ethereum blockchain and its central components as smart contracts and Externally Owned Accounts (EOAs) in more detail. Subsequently in section 2.3, we explain centralized and decentralized Public Key Infrastructure (PKI), a central component of the certificate based identity assertion system *On-Chain AuthSC*, which is the basis of our development and hence the topic of section 2.4.

2.1 Identity Management and the Identity Lifecycle

Every person is endowed with multiple characteristics, attributes and a specific personality. The identity is a representation of that person "[...] in a particular context and consists of identifiers and credentials [...]" [16, pg.287]. An identifier is a collection of one or more attributes and/or characteristics that uniquely identify that single entity [10]. This could be the first name, last name and birth date if this combination would uniquely identify every person. Credentials are used to show proof that one is allowed to use that identity. A popular example for a credential is a passport or a birth certificate. In some cases it is also possible that identifier and credential are the same item, as for example in the case of biometric characteristics.

A digital identity is a specific type of identity, representing an entity (e.g. person, IoT decive) in a communication or computer system to allow virtual interaction with other entities. Possible identifiers are an IP address or an account address in the context of blockchain. Credentials are username and password or the private key to an account address. As communication technology adds another layer of abstraction, a system and framework is needed to manage digital identity [16]. This framework is called identity management and includes lifecycle management, authentication and authorization of digital identities [16].

The identity lifecycle describes the different actions taken during the digital identities existence. At first the digital identity is created by an entity or provisioned by a system administrator for the entity. It is continuously managed and maintained either by a central entity or by the

entity itself. The extent of how free the entity is in managing its identity depends on the type and implementation of the system. One aspect of identity management is the authorization of identities. To limit access, owner of resources as documents or services can implement access control mechanisms, which require an identity to be authorized before using it. To access a resource, identities then first have to be authenticated. This process can be defined as "[...] the act of a system establishing confidence in an identity through the checking of credentials [...]" [10, pg.28]. In the access control step the mechanism then decides whether to grant or deny access to the resource based on an evaluation of the identities authorization [41]. We elaborate access control and authorization in a much deeper analysis later in a survey of access control in chapter 3.
The final step of the identity lifecycle is the de-provisioning or destruction of an identity. It is executed if the entity represented by the identity or the central authority which provisioned the identity decides that it should no longer be an active representation of the entity in the computer or communication system [16].

2.2 Blockchain Technology and Ethereum

The term blockchain is coined by Satoshi Nakamotos in his well known paper "Bitcoin: A Peer-To-Peer Electronic Cash System" [50]. Although he did not specifically define the term blockchain, he did describe the concept which has become increasingly popular under the term blockchain: a cryptographically-secured, immutable, append-only chain of blocks distributed in and maintained by a peer-to-peer network [50]. While with Bitcoin the scope of blockchain is mainly limited to financial transactions, other projects amplified the scope of blockchain to leverage its potential beyond financial applications. The most popular one is the Ethereum project proposed in 2013 by Vitalik Buterin in the Ethereum Whitepaper [13]. In contrast to Bitcoin, which satisfies the very specific use-case of exchanging coins among participants, Ethereum aims to be the generic blockchain platform on which an application for any use-case can be deployed.
During our research we develop our prototype on the Ethereum platform and thus focus on the illustration of Ethereum in section 2.2.2 to 2.2.5. Nevertheless, to be comprehensive and comprehensible, we also provide a short introduction of the fundamental concepts of blockchain and distributed ledger technology in section 2.2.1.

2.2.1 Blockchain: A Peer-To-Peer Distributed Ledger

A blockchain is a distributed system, consisting of an append-only, immutable ledger which is distributed among participants of a peer-to-peer network [7]. Members of the network are identified by a unique account address and can read, update and add to the state of the ledger by submitting transactions. The information stored in a transaction is depending on the blockchain system. For example, while the content of a Bitcoin transaction is nearly only limited to information regarding ownership and transfer of tokens, transactions in Ethereum may contain the code of a complete distributed application, i.e. smart contract. In a

blockchain, as depicted in Figure 2.1, multiple transactions are bundled in the body of a block. The header of the block contains further data which is needed for the administration of the ledger. Given a successful selection and verification of a block in accordance with a consensus mechanism defined by the community, each new block is chained to the previously added block of the chain, by adding a hash from the header of the previously appended block (n+1) to the header of the new block (n+2).
Due to the chaining of blocks, a blockchain is immutable and append-only [7]. No data previously added can be modified or reordered without invalidating the blockchain from the modified block until the most recent one. If the data of a transaction in block n is changed, the hash of the transactions in the header of block n changes and hence its header changes. Since the hash of the header of block n is included in the header of block n+1, a change of block n also changes block n+1 and every succeeding one.
Another central characteristic of a blockchain is the transparency of information stored on the ledger [7]. Every data stored on the blockchain is available to any interested party, because before new transactions are bundled and appended, each transaction first needs to be verified by the community. Thus due to the high availability of the data, transparently stored on the distributed ledger, the privacy and secrecy of information is not ensured. However, due its transparency, the communities trust in the consensus protocol and the blockchain's cryptographic properties, trustless interaction between participants is enabled [7]. Hence, no intermediary, as a bank or payment service provider, is required when two participants are exchanging funds. Given they have already created an account and a cryptographic key pair, users only need to create and sign a transaction and subsequently send it to the peer-to-peer network. Then Miners validate the transaction, include it in a block and append it to the blockchain. Once the block with the transaction is appended and accepted by the community, the state is changed, e.g. the funds are exchanged.

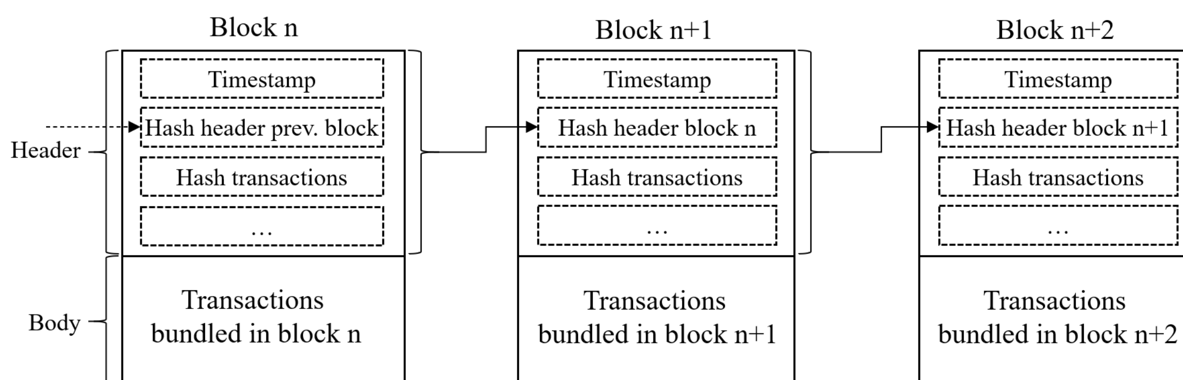


Figure 2.1: Structure of a blockchain

A variety of blockchain systems is available, because different requirements demand different manifestations of blockchain characteristics. Most systems differ in ownership and accessibility. While in permission-less and public blockchains as Bitcoin anybody can participate and

no central entity is the owner of the blockchain, private permissioned blockchains as Hyperledger are owned by a central organization and can only be accessed by already registered entities. Another frequent difference is the choice of the consensus mechanism - the policy of how the community agrees on the state of the ledger [7].
In the following, we introduce the specific characteristics of Ethereum, as it is the system of choice for our development and thus highly relevant for our readers.

2.2.2 Introduction to Ethereum

In an abstract way, Ethereum can be summarized "[...] as a transaction-based state machine [...]" [68, pg.2]. Peers can modify a homogeneous, distributed state tracked by the Ethereum blockchain by creating and submitting transactions to a peer-to-peer network. Executing the transactions requires the use of distributed computation resources. Miners are incentivized by the submitter of the transaction to provide these resources for executing and mining transactions into a new block, by receiving a fee paid in Ether, the currency of the Ethereum network. The payload of the transaction is code, that might invoke state transitions. It is executed by the miner on a local instance of the Ethereum Virtual Machine (EVM), an abstraction layer that " [...] defines the rules for computing a new valid state from block to block" [27].
The goal of Ethereum is to leverage its described technical characteristics and architecture to create a shared world computer with a shared state, shared computation resources and shared memory, that stores data and runs Distributed Application (DApps) for any use-case. Opposed to regular applications, which run on single devices, DApps are stored and executed on multiple devices of a distributed system simultaneously. Therefore, availability of the DApp is not dependent on a single entity. The logic of DApps is implemented in one or multiple smart contracts in the Turing-complete, object-oriented programming language Solidity [26]. Once deployed via a transaction, smart contracts implement the services and rules of DApps and are at the same time an interface through which users can access the application. DApps, respectively smart contracts often can also be accessed via a web-app, in order to increase the user-experience and reduce barriers of entry for users not familiar with blockchain technology.

2.2.3 Ethereum Accounts

In Ethereum, an account is an object that is part of the blockchain's state [26]. Each account is able to send, receive and store Ether, the currency of Ethereum. Accounts are associated to a 20-byte address for identification and accessibility. Thus, the interaction with other accounts, as for transferring Ether from one to another is possible. An account comprises a tuple of four fields [26]. The nonce, an integer value which is increased for each transaction sent from the account, in order to ensure that it can only be sent once. The balance field, which specifies the Ether the account currently stores, an optional field for storing contract code and a field for storage. Ethereum knows two different types of accounts: Externally Owned Accounts (EOAs) and Smart Contracts (SCs).

Externally Owned Accounts are associated to a public, private key pair. The private key consists of 32 bytes, which need to be randomly generated. By applying a cryptographic algorithm, the Elliptic Curve Digital Signature Algorithm (ECDSA), multiple 64-bytes public keys can be derived from one private key. However, only one public key is linked to an EOA, as the unique address of an EOA is the last 20 bytes of the public keys Keccak-256 hash [68]. The individual who possesses the private key is able to control any EOA, which public key is derived from the respective private key. Hence, the possessor is also able to control any Ether stored at the private keys accounts. As a transaction is signed data that comprises a message [26], the EOA can create and sign message with its private key to create a transaction. The implications are discussed in more detail in section 2.2.4.

A smart contract account is an account which comprises persistent storage and a program, i.e. smart contract, that resides on the blockchain and can be executed by EOAs or other smart contract accounts. Opposed to EOAs, smart contract accounts store the smart contract code in the respective contract code field in the account tuple. As a smart contract account does not have a private key, it only is controlled by its bytecode [26]. Furthermore, it can not sign messages and thus is not able to submit transactions. Nevertheless, it can emit events and execute other smart contracts functions and transfer Ether by sending messages. However, smart contract communication is limited to the Ethereum network, as Ethereum requires full determinism. Consequently, each and every execution on the blockchain needs to be repeatable at any given point in time. As the behaviour of off-chain data is unpredictable and might change over time, smart contracts are depending on Oracle services to access external data [31].
The smart contract code is deployed on the blockchain as bytecode, but usually is developed in Solidity, a Turing-complete, but higher-level programming language. Influenced by major programming languages as C++ and JavaScript it supports concepts as libraries and inheritance and allows developers to build complex applications comprising of multiple smart contracts in a user-friendly programming environment [62]. When the development in Solidity has finished, a smart contract account can be created by an EOA by deploying the code of the contract via a transaction to the Ethereum network. The address of a smart contract account consists of the last 20 bytes of the Keccak-256 hash of a value that is deterministically generated from the address and nonce of the creating EOA [68].
As smart contracts are immutable once deployed on the blockchain, neither its creator nor other entities can change the contract code [26]. Changes to the code require redeployment of the contract code to a different contract account address. Since deployment is associated with a fee and all Ether stored in the smart contract account could be lost in case of bugs, sufficient adherence to design patterns and testing (e.g. in a test network) is highly recommended before deploying the smart contract code on the blockchain. Once successfully deployed, any account can execute the smart contracts public functions. Nonetheless, developers can use concepts and rely on design patterns to restrict access to a smart contracts functions. However, it is not possible to keep the smart contract code or its storage secret, since both can

be retrieved by analyzing the blockchain with Blockchain Explorers as Etherscan[1].

2.2.4 Ethereum Virtual Machine (EVM)

The Ethereum blockchain is continuously transitioning from one state to another as the participants of the peer-to-peer network interact with its ledger. The Ethereum Virtual Machine (EVM) ensures the success of each state transition by providing peers with a model for the execution of code to guarantee identical output among all peers in the network - a prerequisite to achieve consensus among all peers on the next state of the blockchain [26, 68].

The Ethereum world state consists of all EOAs and smart contract accounts with their respective storage, as well as the blocks from the initial to the most current one [31]. The state is changed when a transaction is successfully created, signed, submitted and mined in a block. As previously described in section 2.2.3 only EOAs are controlled by a private key and hence are able to sign a message and submit transactions.
A *transaction* can either be created to transfer Ether to another EOA, deploy a smart contract on the blockchain or execute a function on an already deployed smart contract [26]. It consists of the receivers address, the signature of the sender, the value of the to be transferred Ether, a data field, a value that limits the maximum computational steps supported and the price the sender is willing to pay for each computational step (i.e. Gas price) [26]. Depending on the target of the transaction its payload differs. While the data field is empty when transferring Ether to another EOA, it is populated with the smart contracts bytecode when deploying a new smart contract on the blockchain. The bytecode of a transaction consists of EVM Opcodes. For the execution of each Opcode by the EVM a certain computational effort measured in Gas is needed. The Gas needed for each Opcode is defined as specified in the Ethereum Yellowpaper [68]. The total price the submitter has to pay, can be calculated by multiplying the Gas price the submitter is willing to pay with the amount of Gas required for executing all EVM Opcodes for executing a transaction. Hence, depending on the computational steps required for the execution, the submitter must store a sufficient amount of Gas in the transaction. If the transaction runs out of Gas before the execution has finished, it fails and the Gas is not refunded to its submitter. If the transaction succeeds, it is included in a block and sooner or later be appended to the blockchain, changing its state.
As smart contracts can not submit transactions, they communicate by sending *messages* which are not signed, as to execute a function on another smart contract or to transfer Ether. Messages are very similar to transactions, but always need to be triggered by a transaction. Consequently they do not need to be directly stored on the blockchain, as they can be reproduced when redeploying the respective transaction [26].

2.2.5 Asymmetric Cryptography, Authentication, Authorization in Ethereum

The state of a blockchain can be changed, when an EOA signs and submits a transaction to the Ethereum network. The EVM, Ethereums execution model for state changes, needs

[1] www.Etherscan.io, visited 13/12/2020

to ensure, that changes to the blockchains state can only be executed after authentication and in respective cases with sufficient authorization. Otherwise, anybody could access all Ether stored on any EOA and transfer it to any other account. Consequently, Ether would loose most of its value and Miners would not be incentivized to provide their computational resources for changing the blockchains state anymore, as they receive a reward in Ether for including transactions in a block and appending it to the blockchain. Furthermore, the system also needs to guarantee that a transaction can not be intercepted and modified after it was submitted. Else malicious entities could change the receiver of an Ether transfer, the code of a to be deployed smart contract or a function call.
To prevent these unauthenticated and unauthorized changes, Ethereum uses asymmetric cryptography. EOAs are required to sign any transaction before it is submitted with the private key of an asymmetric key pair. Asymmetric cryptography, or also called public key cryptography, leverages characteristics of mathematical functions to encrypt or sign data packages [4]. Such a function, also called a trap-door function, can very easy be calculated, however it is not possible to compute its inverse in reasonable time, unless a secret code is available that allows to take a shortcut for inversing the function [4].

As every information stored on the blockchain needs to be verifiable by anybody, encryption is not used in Ethereum. Nevertheless, Ethereum leverages asymmetric cryptography, more specifically the Elliptic Curve Digital Signature Algorithm (ECDSA) to create public, private key pairs for digital signing. While the private key is used for signing transactions, i.e creating a digital signature, the public key is used for validation. In order to create the digital signature for a transaction, the private key of an account is combined with the transactions content (i.e. message) by the ECDSA algorithm. Due to the specific characteristics of elliptic curve arithmetics, which we discussed earlier, only a person in possession of the signers private key can reproduce the signature [4]. However, for verification of the digital signature, only the public key is needed. Therefore, the EVM can use the EOAs public key to verify that the transaction was sent only by the owner of the EOA (i.e. owner of the private key) and was not modified during transfer. Moreover, the verification of the signature by the EVM is at the same time an authentication, as the EVM is provided with a very strong proof that the sender of the transaction is the owner of the EOA, thus the owner of all funds and smart contracts related to the account. It also is an indirect authorization, as if authenticated, the submitter of the transaction can use the stored Ether as desired. Furthermore, smart contracts in Ethereum might implement additional layers of access control. EOAs might be required to first become authorized by the respective or another smart contract before they are granted with access to the smart contracts services. One possible and simple solution is *whitelisting* of accounts, by adding addresses of approved accounts to a list stored in a smart contract. The address of an account that sends a transaction or message to invoke a function, is first checked against the members of the whitelist. Only if its address is listed, the invoking gets executed. However, these access control solutions have to be implemented by the developers of smart contracts and are not part of the out of the box Ethereum (cf. chapter 6).

2.3 Public Key Infrastructure (PKI)

A Public Key Infrastructure (PKI) enables the secure communication between users of a computer network. It is a framework that defines rules, policies, standards and programs that specify how to identify and register new users and provide them with means to establish a secure connection for communication in the network [6]. Moreover, PKI creates a trust infrastructure, that endows entities (e.g. servers) with proof that they are actually the identity they claim to be. Hence, it is much harder for a malicious entity to deceive users into communication and interaction with a fake identity which claims to be someone else in order to manipulate or steal.

In PKI public, private key pairs are usually created with asymmetric cryptography (cf. section 2.2.5) and are associated to participants of the network. The (hashed) public key is openly available and depending on the type of PKI represented in a different manner. The private key for signing messages in order to establish a secure connection by proofing ownership of the public key is kept secret by the owner of the respective public key [6].
The detailed specification of PKI varies depending on type and use-case. Hierarchical or traditional PKI is based on trust in central entities, i.e. roots which act as trust anchors. They endow entities with trust by issuing certificates, a specific representation of the public key. The secret private key can then be used by an entity to proof ownership of the certificate, hence proof trust endowment from a globally trusted central entity [6]. The most popular application of hierarchical PKI is to establish secure communication over the internet, as without PKI malicious entities could deceive users by faking the look of a trusted website to steal users credentials as username and passwords. Therefore, Certificate Authorities (CA) being trust anchors, issue SSL/TLS certificates to web-servers and website owners respectively [6]. At the end-user, browser verify the certificate and establish a secure connection before opening a website, given its SSL/TLS Root certificate is trusted.
Another approach to PKI is to create trust in a distributed network without relying on central authorities as trust anchors [1]. In such a system each user receives a public, private key pair. A Web-of-Trust is created among participants by signing claims of other users, given the signing party is confident that the claim reflects the truth. Before interacting with another user, users first have to check the claims and signatures of the respective signees. Based on the success of the signature verification and their general trust in the signees they can decide whether they do trust the other party. A prominent project by the W3C that is facilitating decentralized PKI and leverages blockchain are Decentralized Identifiers (DID) [55].
In section 2.3.1 we first introduce the traditional PKI that leverages SSL/TLS certificates, before we briefly introduce the concepts of DID in section 2.3.2.

2.3.1 Hierarchical PKI

The authentication of websites and web servers as well as the establishment of a secure communication connection between client and server on the internet today is conducted by executing the Transport Layer Security (TLS) protocol. The protocol leverages so-called

SSL/TLS certificates of the X.509 standard, structured in a hierarchical PKI [6]. In general it is a tree of certificates, linked by public key cryptography in a chain of trust from the top to bottom. For each SSL/TLS certificate a public, private key pair is created. The public key is stored in the SSL/TLS certificates document, while the private key is kept secret by the owner. Depending on the SSL/TLS certificates type and its level in the hierarchy the private key can be used to sign and endorse another certificate from an equal or lower level. As depicted in Figure 2.2 the different levels comprise three different types of SSL/TLS certificates. As SSL/TLS certificates are the primary type of certificate used in our work, we may refer to a SSL/TLS certificate only as a certificate for the remainder of this research.

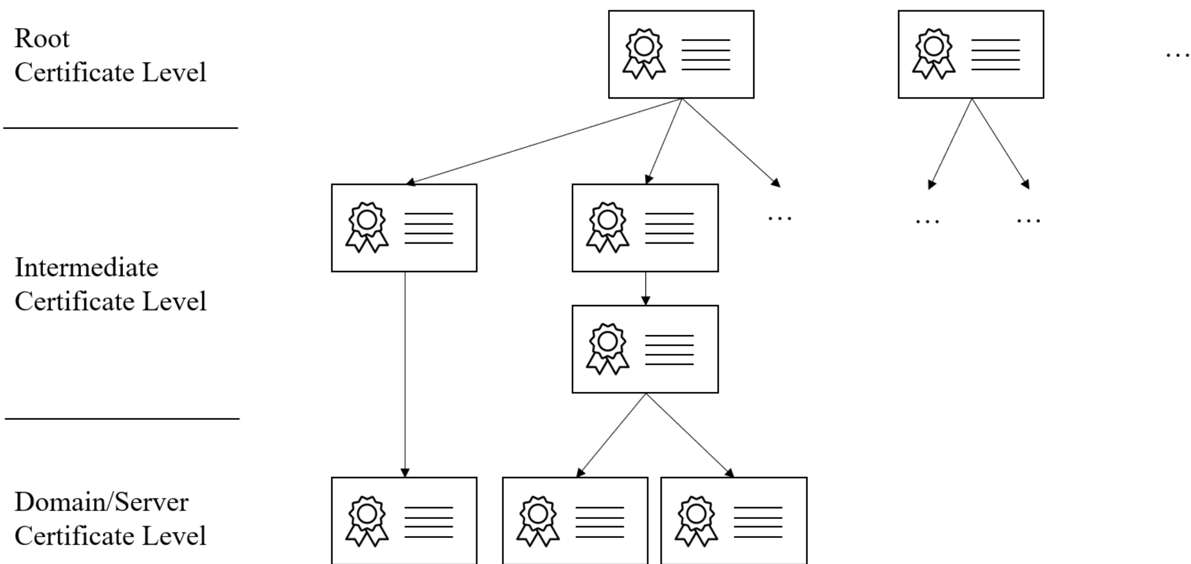


Figure 2.2: Exemplary SSL/TLS certificate PKI hierarchy

On the top level, a very limited number of *Root certificates* are acting as trust anchors, publicly stored for validation of lower-level certificates in root store of browsers, operating systems and other applications [6]. Root certificates are self-signed with their respective private key and are issued and owned by about 15 trusted central Certificate Authorities (CAs). These CAs are usually companies or consortia of large and trusted corporations, that are endowed with trust by its customers and partners to only issue and sign secure certificates to legitimate entities. However, as there have been controversies and compromises at CAs, questions whether such a strong centralization is desirable have been raised [38]. Nonetheless, as the system proofed to be working over the last 30 years and no sophisticated decentralized solution has been built and established yet, certification is still issued by central entities.
The second level in the chain of trust consists of one or multiple layers of *Intermediate certificates*, issued by Subordinate CAs [6]. In order to endow certificates with trust, they request a signature from the private key of a CAs Root certificate. However, Intermediate certificates can also be signed by the private key of another Intermediate certificate from a chain of

trust, which includes one Intermediate certificate signed by a Root certificate. Intermediate certificates are important, as they minimize the risk that a Root certificate needs to be revoked and a new one needs to be issued and added in a resource intensive process to the root stores in browsers, operating systems and multiple other applications.
The final level of the SSL/TLS certificate hierarchy comprises so-called *Domain or Server certificates*, that are issued by Subordinate CAs to the operators of servers or websites, given a successful identity verification at a Registration Authority (RA) [6]. To complement the chain of trust, these certificates are signed with the private key of an Intermediate certificate. As leaves of the tree-like structure, these certificates are not entitled to sign other certificates. However, the private key is still required in the server or website validation process, which we describe in more detail in the Validation subsection.

Although practically a signature in public key cryptography can not be forged, as no known computer system is capable to compute the private key of a respective public key in reasonable time, certificates can still be compromised as the private key can be stolen. This is especially problematic when an Intermediate or a Root certificate is compromised and used to issue certificates to malicious websites which aim to deceive users in order to steal their data. Hence, specific blacklists are maintained, which list all certificates which have been revoked by its issuer. Validating entities as browsers or operating systems check these Certificate Revocation lists in the validation process before establishing a secure connection with a website or server [6].

SSL/TLS Certificates

SSL/TLS certificates are specific digital certificates. The validity of a SSL/TLS certificate is evaluated by the client during the TLS handshake protocol, the sub-protocol of the TLS protocol which specifies how to authenticate a website or server and establish a secure connection [25]. In general digital certificates store a digital signature and assign a public and secret private key to an identity in a computer network, with which the identity can authenticate itself by following a specific protocol. Depending on the protocol certificates are defined by different standards, hence might comprise different data fields stored in a different format.
In the TLS protocol for website and web server authentication digital certificates are encoded with the ASN.1 Distinguised Encoding Rules (DER) and are specified by the X.509 standard from the X.500 Authentication framework. The third version of the standard, which is still used for SSL/TLS certificates, was defined in the RFC5280 in 2008 [17]. In addition to the definition of structure and content of the X.509 v3 certificate, the RFC5280 provides an overview of the general approach of the previously described PKI for the World Wide Web and defines the validation process, as well as the standard for the Certificate Revocation List v2 [17].

In the following we provide an overview of the structure and the content of a X.509 certificate as specified by chapter four of the RFC5280 [17] and depicted by Figure 2.3. Since some components are more relevant than others for the development of an access control mechanism

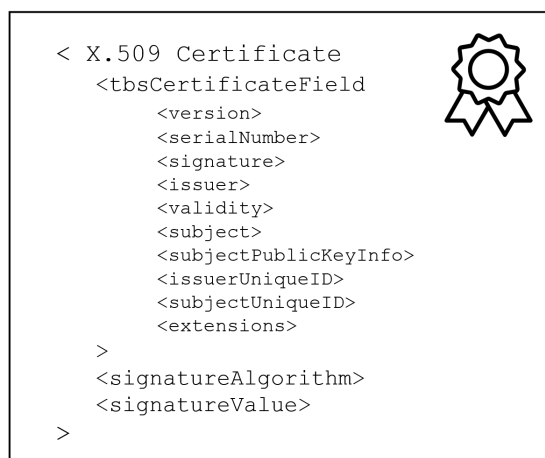


Figure 2.3: Structure of a X.509 certificate [17]

for real-world entities, we set the focus as required to best understand our research.
The certificate comprises three elements in a sequence:

- The **tbsCertificate** field, that stores among others information related to the issuer, subject and status of the certificate.
- The **signatureAlgorithm** field, that specifies the cryptographic algorithm that was applied by the CA to create the signature for the certificate.
- The **signatureValue** field, that stores the CA's digital signature of the tbsCertifcate.

The tbsCertificate basically is the body of a certificate, as it stores most information for the validation of the certificate. It consists of ten fields which can store data in different attribute types. Each attribute type (e.g. organizationName), stored in a certificate field, is associated to an unique Object Identifier (OID) (e.g. 2.5.4.10) to enable improved information retrieval. In the following we describe the fields as depicted in Figure 2.3, in the order they are stored in a X.509 certificate [17].

The *version* field specifies the version of a certificate. It is needed to determine which type of fields are supported by a certificate. While the most current version 3 supports all fields including extensions, a version 1 certificate only supports basic fields. Field *serialNumber* contains the serial number of the certificate - an unsigned integer assigned to each certificate, which is unique among all certificates issued by a CA. In combination with the issuer's name (i.e. CA's name) each certificate can be unambiguously identified. In field three - *signature*, the cryptographic algorithm that is used to create the signature is specified together with some additional parameters. The algorithm identifier must match the one in the *signatureAlgorithm* field in the parents sequence. The *issuer* field can store multiply attribute types and corresponding values that describe the issuer and signer of the certificate in the X.501 name type format. The attribute types are identical to the ones in the *subject* field and

OID	Attribute Type	OID	Attribute Type
2.5.4.3	commonName	2.5.4.11	organizationUnitName
2.5.4.4	surname	2.5.4.12	title
2.5.4.5	serialNumber	2.5.4.42	givenName
2.5.4.6	countryName	2.5.4.43	initials
2.5.4.7	localityName	2.5.4.44	generationQualifier
2.5.4.8	stateOrProvinceName	2.5.4.49	distinguishedName
2.5.4.10	organizationName	2.5.4.65	pseudonym

Table 2.1: Attribute types of subject and issuer fields [17] with respective OIDs [37]

are provided in Table 2.1. The fifth field - *validity* specifies the start and end time, within the CA is willing to endow the subject with trust and guarantee for the validity of the information stored in the certificate. The subject field stores relevant information of the entity to whom the certificate is issued. As in the issuer field, data is stored in the X.501 name type format. The supported attribute types of a X.509 certificate as specified in [17] and their respective OID defined in [37] are displayed in Table 2.1. In field seven - *subjectPublicKeyInfo*, the subject's public key, which is needed for the TLS Validation process is stored. The *issuerUniqueID* and *subjectUniqueID* fields were added in X.509 v2, but are not relevant for SSL/TLS certificates and thus not for our work.
In addition to the nine preceding certificate fields, a tenth *extensions* field was added in X.509 v3 to store further information in order to link additional attributes to subjects or to improve and control inter CA relationships [17]. Three very relevant extensions for SSL/TLS certificates are the *Basic Constraints*, *Key Usage* and *Extended Key Usage* extensions, as they specify whether the certificate is owned by a CA and in which extend it is allowed to sign other certificates [14]. Furthermore, the *Subject Alternative Name* can be used to assign identity related information as additional domain names and email addresses. Further extensions specified in [17] are not relevant for our research.

The structure of SSL/TLS certificates is specified by the X.509 certificate standard, however as some certificates are just used for the basic TLS domain authentication, not all the described fields are always required. Therefore, the CA/Browser Forum, a joint organization of CAs and browser software organizations defined different types of certificates [14]. The certificate types differ in terms of which X.509 fields are required and supported and how the stored information is verified. Subjects can decide which attributes and thus which level of trust endowment from a CA is needed. As some certificate types include more than others, a different extent of validation of the subjects information is required before certification. Hence, the duration and price of certification differ.
The three different types issued by CAs are *Domain-validated (DV) Certificates*, *Organization-validated (OV) certificates* and *Extended-validation (EV) certificates* [48]. *DV certificates* can be automatically created and are cheap. They only store the domain name and verify if the subject who is requesting the certificate is in control of the to be certified domain. *OV*

certificates require a more sophisticated validation process which requires human involvement, as organizations need to provide company records. Attribute types and values included in the subject field in addition to the domain name, are the official organization name, as well as the business unit, country name, state or province and locality. *EV certificates* endow subjects with the highest level of trust, as CAs request a review of legal records of a company. Attribute types require in addition to the ones in *OV certificates* are: full company address, registration number, business category and the subjects place of jurisdiction, incorporation or registration [15].

2.3.2 Decentralized PKI

A first approach to establish a trust infrastructure without central, trusted entities was proposed by Phil Zimmerman in 1991 with Pretty Good Privacy (PGP) [70]. PGP does not rely on central institutions to verify its participating users, but rather leverages a decentralized Web-of-Trust, where users verify themselves by providing a collection of certifying signatures from other users. The idea of such a Web-of-Trust faced multiple challenges, hence it never really was established. Advancements of research in decentralized technologies as blockchain and the need to increase trust beyond websites and servers has facilitated the development of new concept by the W3C [55] - Decentralized Identifiers.

Decentralized Identifiers (DIDs)

The goal of the Decentralized Identifier (DID) community is to create a decentralized structure, where the trust endowment of a Web-of-Trust is sufficient to enable secure authentication of identities and linked attributes without centralized CAs. In such a system, DID are a digital representation of an entity - a digital identity, that can be associated to any desired subject living offline (e.g. human, machine) or online (e.g. application). The W3C defined the core concepts of DID, central design goals (i.e. decentralization, control, privacy, extensible) and high-level technical specifications in [55]. However, it is not the W3C controlling the development. Each of the different DID implementations, called methods, are defined and built decentralized by individual contributors.

A DID is generate by its DID method and stored on a data structure defined by its method. Data structures differ, but usually are distributed ledgers as in Bitcoin or Ethereum. During the DID's generation it gets linked to a public key of an account on the blockchain. After successful deployment the account and its owner are represented by a digital, distinct identity which can be controlled with the account's private key.
While DID methods differ, the structure of every DID is similar: A DID is a string, that consists of the keyword "did", the name of the respective DID method and an unique method-specific identifier [55]. The method-specific ID is a string that is unique for each method and subject and ultimately resolves to a DID document. An exemplary DID generated by the *BTCR DID method*, a method which creates DIDs on the Bitcoin blockchain, is "*did:btcr:8ght-qzaq-3pwq-lksc-8t*". Here the method specific ID is the encoded transaction id of a Bitcoin transaction, in

which a link references to the location of the DID document [2]. The link to the document, as well as its location differs depending on the method. The DID document contains attributes describing the subject. Besides the subjects public key, personal information called a "claim" can be added to the document and linked to the identity. Trust in the correctness of a claim is endowed by a Web-of-Trust in which other identities confirm the validity of a claim with their signatures [55].
As a DID is a digital identity that supports attributes, is uniquely associated to an entity and is only controllable via a private key, it also can be used for authentication and authorization of real-world entities. Therefore, we provide a short literature review of research that applies the concept of DID in the context of authentication and access control in chapter 6.

2.4 On-Chain AuthSC

In Ethereum, in order to transfer funds from one account to another, a user needs to acquire the account address of the receiving EOA or smart contract. As an account address in Ethereum is about 42 hexadecimal characters long, it is not easy to be read and remembered. Furthermore, just from reading the hexadecimal address it is not possible for a human to distinctly associate an account address to an user or organization. Especially in cases, where an account address is published online to allow any user to access and pay for services of a DApp or to acquire tokens in an Initial Coin Offering (ICO), criminals have managed to deceive users into transferring money to wrong and malicious accounts. In most cases, websites were hacked and the account address to which users were asked to deposit funds to was exchanged. To overcome this challenge and to increase usability, solutions as the Ethereum Name Service (ENS) have been proposed. The idea of a name service is to link account addresses to easy-to-read and -remember, web like domains (e.g. blockchainuniversity.eth). Given the wallets support for ENS, the user can easily send money to an account by specifying the ENS domain address. However, adoption still is limited and the link between the account and the business is weak, as a judiciary system for controlling the assignments and reassignments of account addresses and ENS domains is as good as not existing. Moreover, criminals can easily create very similar domains, which might still deceive the user. Therefore, Gallersdörfer et al. propose AuthSC in [32], a SSL/TLS-based identity assertion and verification system. More specifically, an endorsement framework for smart contracts that leverages SSL/TLS certificates for trust endowment.

The general idea of AuthSC is that an organization or person, who owns a website and an Ethereum account, can link a smart contract or/and an EOA to the websites SSL/TLS certificate in order to bootstrap trust from its hierarchical PKI, respectively the CAs Root certificates (cf. section 2.3.1). Given a users trust in the Root certificates, trust is passed to the account by linking it to the PKI's chain of trust. The link, i.e. endorsement between the account and the certificate is created by signing a message that contains the accounts address with the private key of the websites SSL/TLS certificate. Given a successful validation of the endorsement and its signature, users can be sure that the account really is owned by the

person or business they want to interact with, as only the owner of the certificate's private key is able to create one. Furthermore, as the users need to verify the certificate's chain of trust during the validation process, they can decide for themselves, which Root certificates and hence Ethereum accounts to trust.
AuthSC overcomes the challenge to create a missing, trusted link between accounts and websites. However, the concept of AuthSC as proposed by Gallersdörfer in [32] still relies on manual off-chain endorsement validation and off-chain resources, as certificates and endorsements are still stored on central web-servers. In [33], Groschupp and Gallersdörfer refine and enhance AuthSC. They design *On-Chain AuthSC* and propose a first prototype that supports full on-chain authentication of accounts by storing and validating endorsements and signatures on the Ethereum blockchain. Moreover, account endorsements can now not only be created during smart contract deployment, but also in retrospective for already created accounts.

In the following we provide an overview of the systems architecture in section 2.4.1. Then we describe the endorsement framework in more detail in section 2.4.2, before we explain the certificate framework in section 2.4.3. For simplification, we may use the term AuthSC in order to refer to *On-Chain AuthSC* or any other version of the system.

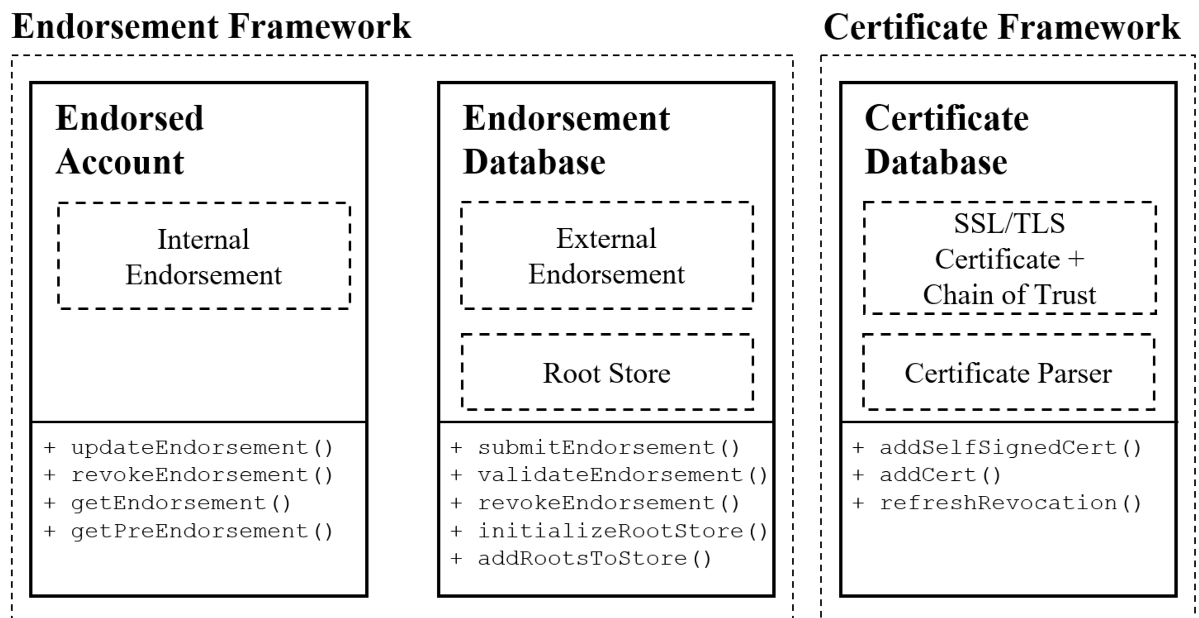


Figure 2.4: Simplified architecture of On-Chain AuthSC adapted from [33, pg.56]

2.4.1 Architecture Overview

On-Chain AuthSC is developed in Solidity for Ethereum. A simplified overview of *On-Chain AuthSC* is depicted in Figure 2.4. Its major components are the endorsement and the certificate

framework.
The **Endorsement Framework** comprises the *Endorsed Account* inclusive of the internal endorsement specification, as well as the *Endorsement Database*, including the external endorsement specification and the Root Store. Endorsements link a to-be-endorsed account to SSL/TLS server certificates. Depending on the account owners preferences endorsements can either be directly stored in the endorsed smart contract (internal endorsements) or in a central Endorsement Database (external endorsements). Each account, that wants to validate an endorsement, needs to create and maintain a Root Store. It contains the Root certificates trusted by the user. During validation the Root certificates trusted by the user are compared to the Root certificate of the endorsed account. Validation, but also creation of endorsements requires a **Certificate Framework**. It comprises the *Certificate Database*, including a certificate parser and the endorsing SSL/TLS certificates with their chain of trust (Intermediate certificates and Root certificate). To create and validate endorsements, information stored in the certificate chain is required. Hence, first the relevant information of all certificates in the chain is retrieved with a certificate Parser and then stored in the central Certificate Database.
On-Chain AuthSC requires all relevant data that is needed for account authentication and endorsement validation to be stored on the blockchain, as external data can not be retrieved very easily due to a blockchains need for determinism (cf. section 2.2.3). Consequently, beside the endorsed account, the Endorsement Database and Certificate Database, each verifier's Root Store is stored and maintained on the blockchain. In the following we introduce each of the components in more detail.

2.4.2 Endorsement Framework

The purpose of an endorsement is to be a distinct, verifiable proof that an account is endowed with trust from a SSL/TLS Server certificate of a specific website. Therefore, an endorsement E contains a verifiable signature S, that is a hashed message signed with the private key of that certificate. The content of that message is a claim C, which contains the account address of the endorsed account, the Fully Qualified Domain Name (FQDN), the ID of the endorsing certificate and the desired expiry date of the endorsement. As the claims content is also needed for the validation of E, C also is included as tuple in E:

$$E = \{S, C\}, \text{ with}$$
$$S = \{sign(hash(C), key_{priv})\} \qquad \text{and} \qquad C = \{address_{account}, FQDN, ID_{cert}, date_{exp}\}$$

On-Chain AuthSC supports the endorsement of EOAs and smart contracts. Since the different accounts have different specifications and requirements, *On-Chain AuthSC* distinguishes between two different types of endorsements: Internal and External endorsements [33].
Internal endorsements are directly stored in an account. However, since internal endorsements require the implementation of specific functions and EOAs can not store executable code, it is not possible to internally endorse EOAs. Furthermore, it is not possible to add an internal endorsement in retrospective to an already deployed smart contract. Only smart contracts that are deployed with a specific internal endorsement specification may be internally

endorsed. The specification includes functions to add, update and revoke endorsements, as well as variables to store the current and past endorsements, and the endorsing certificates ID. While internal endorsements are very easy to access, the validation needs to be conducted by each user individually. As a consequence, total Gas costs are higher than for external endorsements.
External endorsements are stored in a central, on-chain[2] Endorsement Database. Any smart contract or EOA can be endorsed by an external endorsement, as external endorsements do not require any modifications as functionality and storage is provided by the Endorsement Database. Supported functions are `submitEndorsement()`, `validateEndorsement()` and `revokeEndorsement()`. The Gas cost intensive signature validation, which needs to be conducted by each validator for an internal endorsements, is only conducted once after the submission and before adding it as an `Endorsement` struct to a mapping in the Endorsement Database. Hence, the validation of an external endorsement is less Gas cost intensive, as for internal endorsements. However, users that want to verify an external endorsement, can not retrieve the endorsement directly at the account. They first have to initialize a Root Store at the Endorsement Database and add trusted SSL/TLS Root certificates (if not already done during a previous validation). The Root Store in the Endorsement Database is similar to the Root Store of browser or operating system vendors. It allows validators to store the SSL/TLS Root certificates of trusted CAs. Only endorsements, that are linked to SSL/TLS Server certificates in a chain of trust with a Root certificate from the validators Root Store can be validated successfully.

The process of creating an endorsement differs depending on the type of account and its deployment status. As for the development of our system, only external endorsements are needed, we only explain the respective process. Readers interested in the creation of internal endorsements are referred to [33].
There are two prerequisites a user needs to satisfy to create an external endorsement: Being the owner of an Ethereum account and possessing a SSL/TLS certificate and its private key. Given these requirements are satisfied, the user can leverage the functionality of the Endorsement Database to create and then add an external endorsement for an EOA or smart contract. To create the endorsement, the user first needs to submit the certificate chain to the Certificate Database as described in section 2.4.3. Next, the attributes of the claim ($address_{account}$, FQDN, ID_{cert}, $date_{exp}$) need to be acquired. The ID_{cert} is the fingerprint of the certificate and is emitted in an Event after successful submission of the certificate to the Certificate Database. The attributes are then passed to the `getPreEndorsement()` function of the Endorsement Database. It returns the hashed claim, that then needs to be signed with the private key of the certificate. Once the signature is created, it is passed together with the claims attributes to the `submitEndorsement()` function. Given a successful validation, the endorsement is added to a struct which stores the account address, the FQDN, the ID_{cert}, the

[2]Groschupp also proposes an off-chain Endorsement Database and external off-chain endorsements (cf. [33]). However, as the prototype of *On-Chain AuthSC* uses an on-chain Endorsement Database, we only describe the on-chain version.

ID of the respective Root certificate, the $date_{exp}$ of the endorsement, a timestamp of when the endorsement was added and a variable that specifies the revocation status of the endorsement. The `Endorsement` struct is then stored in the `EndorsementStore` mapping for the respective account (an account can have multiple endorsements from multiple certificates), as well as in the `endorsements` mapping, where all endorsements are stored.

A concept which has not yet been implemented in the *On-Chain AuthSC* prototype, but was proposed by Gallersdörfer in [32] are flags. The idea is to include a tuple of parameters in the claim of an endorsement. These flags allow to restrict or extend the usability of endorsements in *On-Chain AuthSC*. Flags proposed by Gallersdörfer in [32] are:

- *DOMAIN_HASHED*: Specifies that the FQDN of an account is hahsed. Increases privacy as it is much harder to associate an account directly to a website for a stranger.
- *ALLOW_SUBENDORSEMENT*: Determines whether an account can create a sub-endorsement for another account.
- *EXCLUSIVE*: Limits the number of accounts endorsed by one domain to one.
- *TRUST_AFTER_EXPIRY*: Specifies if data included in the endorsed account should still stay valid even though the endorsement is expired.

2.4.3 Certificate Framework

Endorsement creation and validation requires the signature and the public key of a SSL/TLS Server certificate. The certificate framework provides the means for storing, maintaining, validating and revoking certificates. However, as the signature algorithms of X.509 certificates differ, not all certificates are supported. Currently *On-Chain AuthSC* only accepts X.509 version 3 certificates, with RSA-SHA1 and RSA-SHA256 signatures. Therefore, only websites and servers with such a SSL/TLS certificate can endorse an account with *On-Chain AuthSC*.

In *On-Chain AuthSC* the endorsing SSL/TLS certificate and every certificate in its chain of trust is stored in the Certificate Database. In order to successfully add a certificate, the user needs to submit the certificates DER-encoded data. However, the user needs to abide to the hierarchical order of the certificate PKI, when adding a chain of certificates to the database: First one Root certificate, second one or multiple Intermediate certificates and third one Server certificate. Furthermore, as Root certificates are self-signed and Intermediate and Server certificates require a signature from a higher-level certificate, users need to invoke different functions with different payloads. Root certificates are added by passing its DER-encoded data to the `addSelfSignedCert()` function. Intermediate and Server certificates, by passing the certificates DER-encoded data, as well as the unique SHA-256 fingerprint of the signing preceding certificate to the `addCert()` function. Once the data is received by the Certificate Database, certificates are parsed and relevant attributes are stored in a certificate struct. Certificates are not stored in their original DER formatting as a whole document, since

attribute retrieval from a large file requires computation and thus Gas cost intensive parsing. Therefore, when a new certificate is submitted to the Certificate Database, the Parser extracts most attributes already. However, some attributes are still stored together as a DER-encoded ASN.1 hexadecimal bytes variable. An overview of the certificate structs attributes, which are relevant to our work, is depicted in Figure 2.5.

Before the certificate struct is finally stored in the certificate mapping, a validation function checks the validity of the submitted certificate by verifying the extracted signature with the public key of the signing preceding certificate, the expiration date and the content. If any of these validations fail, the process is aborted and the certificate not added. Once all the certificates in a chain of trust are successfully stored, an endorsement can be created.

As SSL/TLS certificates expire, the Certificate Database needs to support the revocation of certificates. A certificate is considered revoked, if the expiry date is larger than the current date. When the endorsing certificate of an endorsement is revoked, the validation of it fails. While it is not possible to delete a revoked certificate, as other certificates or endorsements might still rely on it, a certificate can still be reactivated by invoking the `refreshRevocation()` function. However, since the revocation does not affect the development of our system, we do not provide a detailed description here and refer the interested reader to [33].

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.

Figure 2.5: Selected attributes of certificates stored on the Certificate Database from [33, pg.63]

3 System Design

The goal of this research is to design and implement an authentication and access control system for smart contracts, that evaluates access requests of blockchain accounts from real-world entities, based on their endowment with trust from SSL/TLS certificates. To link real-world entities blockchain accounts to SSL/TLS certificates, we bootstrap the identity assertion and verification system *On-Chain AuthSC* by Groschupp and Gallersdörfer [33][32] and create an endorsement framework. Subsequently we design an access control framework, in which we leverage the trust and attributes of the SSL/TLS certificates for access control at smart contracts. To reduce complexity in this and the following chapters we may refer to real-world entities as *user* and to their account as *user account*.
In section 3.1 we introduce a reference and multiple exemplary use-cases to introduce the concept of our system and to demonstrate its need and applicability. Subsequently in section 3.2 we describe the functional and non-functional requirements. In section 3.3 we introduce the endorsement framework of our system, before we conduct a survey of access control mechanisms in section 3.4 to identify a matching foundational theory for the design of our access control system in section 3.5.

3.1 Use-Cases

Access control of real-world entities at smart contracts enables a differentiated degree of user interaction at a protected resource. Together with a SSL/TLS-based identity assertion and verification system it allows service providers to restrict access to their smart contracts based on the requester's SSL/TLS certificate attributes. In this section we would like to demonstrate some use-cases for such a system. First we illustrate a reference use-case, before we introduce three more complex exemplary ones.

3.1.1 Reference Use-Case

The goal of the reference use-case for an authentication and access control system at smart contracts is to introduce the general concept of the system we aim to develop in this research. It provides guidance for the definition of the requirements, the system design and prototype implementation.

Automated Authentication and Access Control at Smart Contracts for Higher Education

Authentication and access control usually require to first register and authorize a user at a trusted entity of the requested service or to rely on identity federation protocols, which

are neither defined nor established for smart contract access control yet. Nevertheless, authenticating and authorizing at a smart contract without a previous registration is feasible on blockchain when leveraging *On-Chain AuthSC* in combination with an access control system which evaluates the attributes of a SSL/TLS certificate that endorses the access requesting user account. This enables the automated authentication and authorization of a whole group of accounts that is endorsed by SSL/TLS certificates that have common attribute types and values.
In our use-case we assume that *On-Chain AuthSC* is deployed on the Ethereum blockchain and an Application A, i.e. a smart contract is protected by an access control mechanism. It ensures that A's functionality can only be executed by a limited group of entities, endorsed by a SSL/TLS certificate of an ".edu" Top-Level-Domain (TLD). Furthermore, let us assume, that a university with a ".edu" TLD is the issuer of a smart contract B, which is endorsed by the universities SSL/TLS certificate. If B wants to use Application A, B is first authenticated and its authorization checked by the access control mechanism at A. To do so, it looks up the endorsement and the attribute type "domain" of B and compares it with the required attribute value ".edu". In the case that both attribute values match, A checks B's endorsement by looking up and checking the relevant certificate chain. Given that the certificate of B is included, not revoked, still valid and the Root certificate trusted by the Application, A grants access to B.

The use-case can be elaborated even further to enable automated access control for EOAs. Since most individuals and owners of EOAs are not owner of a website and thus not a SSL/TLS certificate, these entities need to be first linked to, i.e. sub-endorsed by a smart contract that acts as a Registry. The Registry is endorsed by a SSL/TLS certificate and its owner may create sub-endorsements for a users EOA. When the sub-endorsed EOA then requests access to Application A, the access control mechanism checks B's authorization by validating the endorsement of the Registry, the sub-endorsement of the EOA and evaluating the attributes of the endorsing SSL/TLS certificate.
Given the system as described above, Bob is a student at the previously described university and has an EOA on the blockchain. Given that his university links his account to their Registry B, Bob should be able to access the functionality of Application A. That means, when Bob wants to request access to the service of an Application A, instead of previously registering at A, he or she provides B's address to A. Subsequently, the Application checks if Bob's EOA is sub-endorsed by the universities Registry B. That being the case, A validates the endorsement of Registry B and evaluates the "domain" attribute type of B's endorsing SSL/TLS certificate. Given success, Bob can use Application A.

3.1.2 Exemplary Use-Cases

We introduce three exemplary use-cases to demonstrate the potential of our system beyond the reference use-case. At first we outline a modified concept, which describes an improved admission process of new members in consortia blockchains. Next we show how our system can be used to improve the process of online identification at Identification Verification Providers. Finally, we illustrate the system in the context of managing a fleet of vehicles.

Member admission at Consortia Blockchains

Consortia blockchains are created and maintained by a group of organizations to jointly pursue a common interest. Very often a common business or research environment requires high standards in data privacy, scalability and governance [24]. Therefore consortia deploy persmissioned blockchains, where participation is limited to admitted members. An example for a research oriented consortia blockchain is Bloxberg, which was initiated by major research institutions as the Max-Planck-Institute in order to facilitate the development of distributed research services [30]. Admission to the blockchain can be requested online, while admission is only provided to organizations with focus on research. Each application is manually reviewed by each consortia member, which then can casts a vote for or against admitting the requester. This process requires extensive manual effort, particularly if the consortia is receiving a large amount of applications. Depending on the requirements for the reviewing process, an automated access control based on certificate attributes can eliminate the need for any manual work or at least create a first quality gate which reduces the manual workload.

Let us consider the example of a permissioned blockchain from a research consortia C, where admission of new members is reviewed manually by each active member. Assuming each member is casting the vote only based on the fact that the requester R is a research organisation, then we can automate the process by comparing the domain of R with a list of acknowledged research organizations (e.g. list of recognized research entities published by the European Statistical Office [28]).
To start the process, R first has to request access by calling a function at the admission smart contract of C. The admission smart contract can either be deployed on the consortia blockchain, if some interaction with the network is available for external parties and the blockchain already supports *On-Chain AuthSC*, or on a public blockchain as Ethereum. In both cases, R needs to interact with C using a smart contract or EOA endowed with an endorsement of R's SSL/TLS certificate, because the certificate is providing the prove that R is a research organization. The admission contract then checks the endorsement of R, by retrieving the certificate chain from the Certificate Database. If C's smart contract decides to trust the Root certificate, i.e. endorsement, it retrieves the attribute "domain" from the SSL/TLS certificate and compares the value with a list of domains from acknowledged research institutes. If R's domain is included on the list, R is admitted to the consortia blockchain immediately. If that is not the case, R's request has to be reviewed manually or depending on the policy the request gets rejected immediately.

Blockchain Enabled Identity Verification Provider for DApps

Financial Services as stock trading applications require the user to first provide proof of identity to comply with legal requirements which prevent tax evasion and fraud. Currently, the verification of a users identity online is enabled by third party-services as POSTIDENT[1]

[1]https://www.deutschepost.de/en/p/postident.html, visited 13/12/2020

or WebID Solutions[2]. During that process the user has to participate in a video call, to answer some questions and present an official government issued ID. Moreover, usually a phone number has to be provided to receive a TAN which needs to be entered to finalize the process. Online verification saves time and efforts by eliminating the need to engage in a lengthy real-world face to face verification. Nevertheless, instead of linking the verified user to a personal online identity, which he or she can present to the next service where verification is required, the user needs to run each time through the same verification process.

Leveraging our access control mechanism with *On-Chain AuthSC*, parts of the process can be transferred to the blockchain. It allows to create a link between an account and a real-world identity, as the account can be added to a Registry of the Identity Verification Provider on the blockchain once the verification process was successfully conducted. Consequently the users only need to participate once in the verification process, since their identity is immutably linked to the EOA or smart contract on the blockchain via the Registry. Subsequently, the EOA or smart contract can act on behalf of the linked real-world identity and interact with any other smart contract.

Considering an example where the company C plans an ICO and the user Bob wants to acquire some of the companies tokens. Due to security concerns the company requires Bob first to verify through a trusted Identity Verification Provider V. At first Bob participates in a video call with V, where Bob provides the address of his EOA, provides a government issued ID and answers some questions. Still during the call V asks Bob to sign a specific piece of data with his EOA's private key and send it back to V. After successfully checking the signed data with the EOA's public key, V ends the video call and adds the EOA's address and some of the information from Bob as his full name to V's Registry. The EOA is now trusted by V to be acting only on behalf of Bob, i.e. Bob is sub-endorsed by V. In order for Bob to be trusted by C, C still needs to trust the Registry of V. Therefore the Registry needs to be endowed with trust from the endorsement created from the SSL/TLS certificate of V. When Bob now wants to acquire tokens at C, Bob calls the respective function of C's smart contract and provides V's Registry address in addition to the input required by the function to acquire the tokens. Before executing the acquisition of the tokens, two conditions are checked by C's smart contract. At first, if Bob's address and name match the entry in the Registry of V. Second, if the Registry can be trusted, i.e. if the Registry is endorsed by the certificate of a trusted identity provider of C. If both are met, Bob is identified and authorized to acquire tokens of C's smart contract. The next time Bob is asked to verify his identity on the blockchain to acquire tokens or use a smart contract, he can directly interact with the smart contract and does not need to verify again at an Identity Verification Provider, as long as V is trusted by the service provider and included in its Registry.

[2]https://www.webid-solutions.de/en/, visited 13/12/2020

Automated Authentication and Access Control System for a Fleet of Vehicles at Smart Contracts

Fleet management and leasing companies as well as car manufacturers are exploring the blockchain to track vehicles transactions by linking the Vehicle Identification Number to entities in the blockchain. Leveraging our access control system together with *On-Chain AuthSC*, we enable vehicles to authenticate and authorize at, as well as interact with other entities on the blockchain. More specifically, the owner of the vehicle can use the EOA of the vehicle, i.e. the identity of the vehicle on the blockchain, to use and/or acquire services only available to a certain vehicle or user group.

At first the vehicle needs to be registered at a Registry smart contract of the fleet management company C, which is stored on the blockchain and endorsed by C's SSL/TLS certificate. Then, the driver of the vehicle can use multiple services, as refueling at a gas station or visiting a workshop at a service provider P who maintains a smart contract. When requesting a service, the vehicles EOA calls the function of the respective service at P's smart contract and provides the address of C's Registry contract. Before approving a service for the vehicle, the smart contract of P authenticates the vehicle and check its authorization at C's Registry smart contract, by first evaluating if the EOA of the vehicle is included in the list of sub-endorsed vehicles. Then it looks up C's contracts endorsement and retrieves the "domain" attribute type from the certificate to check if C is a known business partner. Furthermore, it checks that the certificate is not revoked and still valid. At last it retrieves the whole certificate chain from the Certificate Database to check if the certificate claiming that C is the actual business partner can be trusted. If the Root certificate is trusted by P, the service can be provided to the vehicle and the fleet management company can be billed.

3.2 Requirements

The next step in our system design process is the definition of the requirements, conducted in a requirement analysis. Our goal is not only to cover the requirements for the features of our system (Functional Requirements (FR)), but also requirements as in the context of usability, cost and blockchain technology (Non-Functional Requirements (NFR)).
The initial goal of the analysis is to first derive basic functional and non-functional requirements from the reference use-case introduced in the previous chapter and our research. In a second step we test and refine the requirements by conducting expert interviews in section 5.3.2.

3.2.1 Functional Requirements

Based on the reference use-case of an automated authentication and access control system for smart contracts, we determined the functional requirements for our systems key components, the endorsement framework with a Registry and the access control framework at an Application. Our system should be able to allow users to act on behalf of a SSL/TLS certificate

endorsed Registry. Hence, this Registry needs to conduct user sub-endorsement management. The owner (e.g. individual or organization) of the Registry needs to be able to sub-endorse a user account, given it is trusted (FR1). Being sub-endorsed the user is endowed with trust and may leverage the attributes of the SSL/TLS certificate which endorses the Registry for authentication and access control at an Application. In case the owner of the Registry does not want to endow the user with trust anymore, the endorsement of the user needs to be able to be revoked (FR2). The user should then not be able to leverage trust and attributes for authentication and access control at an Application anymore. Furthermore, the owner needs to be able to update an existing sub-endorsement (FR3).

The access control mechanism at the Application should only allow a user to access the protected resource if the user is trusted by a Registry, that is endorsed by a SSL/TLS certificate with desired attributes. Hence, a user first needs to be able to check if a sub-endorsement is already created for his or her account (FR4). If that is the case, the user should be able to request access to the protected resource of the Application (FR5) and should then be automatically authenticated (FR6). The central functional requirement is the ability of the access control mechanism to check the authorization the user account (FR7). At first the mechanism needs to be able to check the endorsement of the user, i.e. if the user is sub-endorsed by an organizations Registry (FR7.1). Secondly, it needs to be able to validate the Registry's endorsement from the SSL/TLS certificate, to determine if the Registry and consequently the user can be trusted (FR7.2). Finally, the mechanism needs to retrieve and check the attribute values of the endorsing SSL/TLS certificate against the attribute values required by the Application (FR7.3), to certainly determine if the user is authorized to use the functionality the Application is offering.

3.2.2 Non-Functional Requirements

The goal of our research is to develop a system which leverages SSL/TLS certificates to enable access control at smart contracts. Access control requires authorization of entities based on characteristics and policies. As described in the reference use-case we propose to use the attributes in X.509 TLS certificates as characteristics for authorization of entities (NFR1). This enables our system to control access at smart contract Applications without the need for active management of the characteristics, since the SSL/TLS certificates are part of an externally managed infrastructure. Moreover, the SSL/TLS certificate PKI is an existing trust layer, that guarantees the correctness and immutability of characteristics. To expand the trust and associate attributes to Registry smart contracts on the blockchain, we need to leverage the previously described system *On-Chain AuthSC* (NFR2), by creating an immutable connection between the Registry and a SSL/TLS certificate through an endorsement. Determinism is a requirement for blockchain systems, since all nodes of the system need to compute the same outcome independently. To guarantee deterministic behavior, although user endorsements change over time and hence access control decisions are time dependent, the policies, characteristics and access control decisions need to be on-chain (NFR3). As soon as the owner of the Registry is endorsed by the SSL/TLS certificate, he or she is then able to decide for

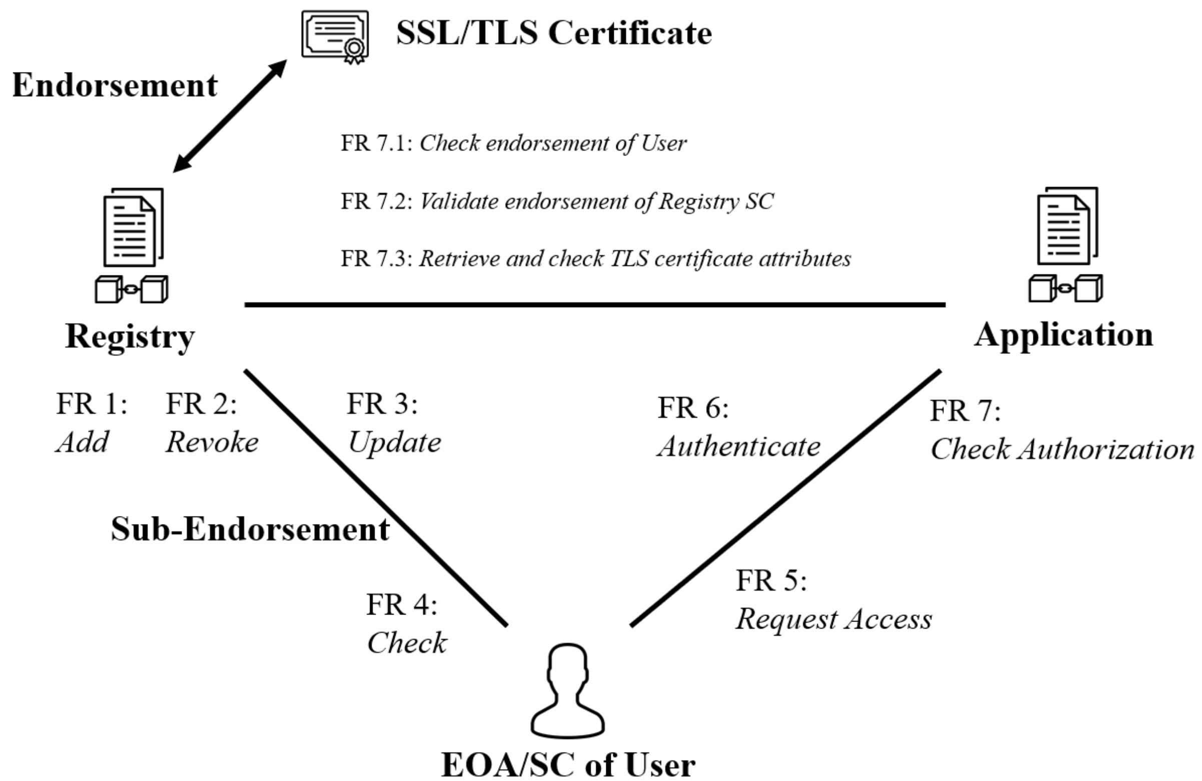


Figure 3.1: Functional requirements

which entity a sub-endorsement should be added. In order to prevent centralization and increase flexibility and participation, it should be the independent decision of the Registry's owner and not the one of a central authority whom to endow with trust and the linked certificates attributes (NFR4). Another non-functional requirement of our system is that access control decision should be independent of the Application's direct trust in the Registry, but only rely on the trust endowed by the certificates through an endorsement (NFR5). This enables easy addition and removal of Registries or Applications without the need to maintain trust relationships between both components. Furthermore, to allow any user to attempt authentication and access control at any smart contract independently of its association, an access request at an Application which uses our access control mechanism should not require any kind of pre-provisioning of the user account at the Application (NFR6). This kind of federation increases flexibility, decrease barriers of entry and increases the user-friendliness of our system. Finally, we also need to consider the costs related to the on-chain computations required for managing access rights and securing access control. More specifically we need to design a system where the cost for registration, authentication and authorization is minimal and thus competitive (NFR7).

The requirements can be summarized as followed:

Functional Requirements

FR1: Add sub-endorsement at Registry
FR2: Revoke sub-endorsement at Registry
FR3: Update sub-endorsement at Registry
FR4: User account may check status of sub-endorsement at Registry
FR5: Request access to Application
FR6: Authenticate user account at Application
FR7: Check authorization of user account at Application
  FR7.1: Check sub-endorsement of user account at Registry
  FR7.2: Check endorsement of Registry
  FR7.3: Check attributes of endorsing SSL/TLS certificate

Non-functional Requirements

NFR1: Leverage attributes of SSL/TLS certificates
NFR2: Use On-Chain AuthSC
NFR3: On-Chain access control decisions
NFR4: Decentralized sub-endorsement allocation
NFR5: Access control without a direct trust relationship with the Registry
NFR6: Access control without pre-provisioning of the subject at the Application
NFR7: Minimal costs of user management, authentication and authorization

3.3 Endorsement Framework of the Smart Contract Access Control System

In this section we elaborate the endorsement framework for the authentication and access control system for smart contracts, that evaluates access requests of blockchain accounts from real-world entities, based on their endowment with trust from SSL/TLS certificates.

An authentication and access control mechanism needs to evaluate an access request based on characteristics specified by a trusted entity. Yet, endowment of trust and association of trusted characteristics is especially challenging in public blockchain systems, as the users real-world identity and its level of trust is not distinctly determinable. Furthermore, as a public blockchain is a decentralized system, central trust providing entities are not available. Although recent research is actively elaborating a decentralized trust providing infrastructure, no such system has been successfully established yet. Therefore, we are facing the decision to either contribute to the creation of a decentralized trust infrastructure as DID on the blockchain or to leverage an existing, fully functional system from outside the blockchain.

DIDs might be the most advanced decentralized trust infrastructure on the blockchain, as they are already relatively functional and accepted by the community. Furthermore, DIDs already

supports the association of characteristics via attributes in claims. However, the endowed trust is still very limited since no Web-of-Trust has been successfully established yet. Hence, as we want to bootstrap trust, DIDs and decentralized trust infrastructures are not a solution to our problem yet.
Our second option is to leverage trust from a blockchain external trust infrastructure. By bootstrapping an existing trust infrastructure we can rely on a proven infrastructure that already inheres high levels of trust. Furthermore, users of the existing infrastructure have limited on-boarding costs. However, all the trust infrastructures we identified are centralized and the trust still has to be expanded to the blockchain. Nevertheless, we are certain that by picking a large, widely supported and established PKI that already has a long and relatively successful track record, trust endowment is sufficient.

We bootstrap the SSL/TLS certificate PKI, as it meets the just described criteria and with *On-Chain AuthSC* a system to expand the PKI to the blockchain is already available. Furthermore, SSL/TLS certificates also provide a source of trusted attributes that describe the certified entity. However, *On-Chain AuthSC* only supports direct trust endowment of user accounts that own a SSL/TLS certificate. This limits the applicability of our system as SSL/TLS certificates are only available to entities that own a website. Therefore, in our endorsement framework we need to separate the trust link between SSL/TLS certificates and the user accounts and implement a trust interface in between: The **Registry**, a smart contract which allows certificate owners to act as attribute provider and manager for user accounts. The first link is an endorsement that expands the trust endowment of a SSL/TLS certificate to a Registry on the blockchain. The second one is a sub-endorsement, that can be created by owners of a Registry in order to link a user account who does not own a SSL/TLS certificate to the owners certificate. Consequently, user accounts are then indirectly endowed with trust from the Registry's SSL/TLS certificate and may leverage its attributes for authentication and access control. An overview of the endorsement framework is depicted in Figure 3.2.

In the following we describe the two components of our endorsement framework: The endorsement for the Registry in section 3.3.1 and the sub-endorsement for the users account in section 3.3.2. Moreover, we revisit flags (cf. section 2.4.2) for endorsements and discuss how this concept proposed by Gallersdörfer in [32] might be beneficial to our research.

3.3.1 Endorsement - Endorsing the Registry

In our research an endorsement is a verifiable link between a Registry and a SSL/TLS certificate PKI that can be created by the owner of the SSL/TLS certificate's private key. The endorsement creates credibility in sub-endorsed user accounts and attributes, as it is proof that the owner of a trusted SSL/TLS certificate approves sub-endorsements of the respective user accounts. Hence, it allows an entity to expand its public confidence to the blockchain and endorse other accounts. To enable the creation of such a link, we draw from existing research of Groschupp [33] and Gallersdörfer [32] and leverage the SSL/TLS-based identity assertion and verification system *On-Chain AuthSC*.

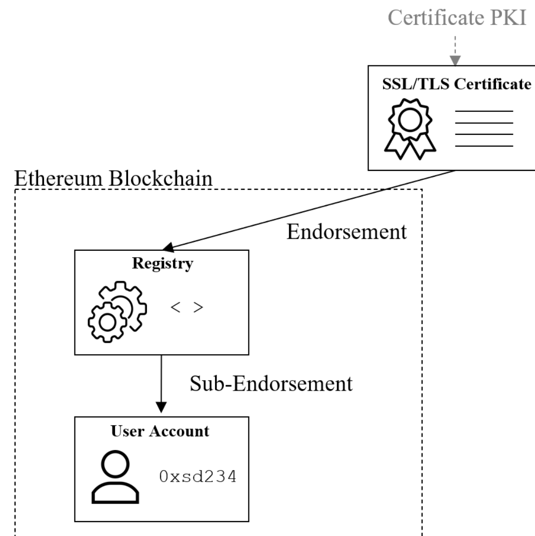


Figure 3.2: Endorsement framework

On-Chain AuthSC is the only system, that can create such verifiable links, i.e. endorsements between SSL/TLS certificates and blockchain accounts. Moreover, it supports on-chain validation and revocation of an endorsement and already provides a database to store and manage SSL/TLS certificates on the blockchain. Implemented in Solidity for the Ethereum blockchain we can easily integrate it in our developments. Furthermore, as we explain in section 3.5 it already provides an on-chain Certificate Database from which we can easily access the SSL/TLS certificate data our system requires to conduct smart contract authentication and access control of real-world entities.

In *On-Chain AuthSC* an endorsement is specified as a tuple that contains a verifiable signature and a claim, which comprises the account address of the Registry smart contract, the FQDN, the ID of the endorsing SSL/TLS certificate and the desired expiry date of the endorsement (cf. section 2.4.2) [33]. Furthermore, *On-Chain AuthSC* supports two types of endorsements: external and internal endorsements. While the specification of the endorsement remains similar among both types, they differ in how they are associated to an account (cf. section 2.4.2).
For the development of our endorsement framework we are now facing the decision of how to adopt *On-Chain AuthSC*. With regards to the endorsement type, we only use external endorsements to link SSL/TLS certificates and the Registry, as external endorsements can be validated more efficiently than internal endorsements, can easily be accessed via an Endorsement Database and do not require a specific structure of the Registry. The decision of how to adopt the endorsement specification is a more complex one. We may either adopt the specification of the endorsement from Groschupp in [33] or change and improve the design if needed. In order to determine whether changes are needed, we evaluate two additional design choices in the following. An **Endorsement with Flags** and the **Nesting of**

Sub-Endorsements in Endorsements.

Endorsement with Flags

One possible design choice is to add flags as proposed by Gallersdörfer in [32]. These flags are a configuration for the Registry, defined and approved by the owner of the signing SSL/TLS certificates private key. In our case they could allow to limit the scope of sub-endorsements, more specifically the scope of how users are sub-endorsed and how and where they can use the sub-endorsement. As an example, the flag *ALLOW_SUBENDORSEMENT* may indicate whether sub-endorsements are allowed by the Registry. The Registry might still have the functionality to endorse accounts or even maintain a list of sub-endorsements, but once the flag is changed to FALSE, access requests are rejected and no new sub-endorsements can be created.
In general we think that flags add value to endorsements, as they increase trust in the functionality of the endorsed smart contracts. Especially in cases where the owner of the endorsing SSL/TLS certificate and the Registry differ, it increase the control of the trust provider. However, in our use-cases we presume that the Registry and the SSL/TLS certificate have a similar owner. Furthermore, as for any changes to the claim the whole endorsement needs to be recreated, validated by and added to the Endorsement Database, changes to flags that are directly stored in the endorsement are very computation and Gas cost intensive. This especially becomes a problem if the authentication and access control system is deployed on a public blockchain. Moreover, changing the design of the endorsement also requires to change a large component of *On-Chain AuthSC*, thus add significant complexity. Hence, in our system design we do not add flags to the endorsement.

Nesting of Sub-Endorsements in Endorsements

A second possible design choice is to complement the initial claim of the endorsement specified by Groschupp in [33] with a list of the sub-endorsed accounts. In this case the sub-endorsements is not stored in the Registry but in the endorsement itself. Although this increases the credibility of the sub-endorsed accounts, it significantly increases Gas costs as every creation and revocation of a sub-endorsement requires a new endorsement for the Registry. Since creation and revocation are very common operations, this design is not economically feasible on a public blockchain.

After the evaluation of both design choices we conclude that there is no need to modify the endorsement specification of *On-Chain AuthSC*. Furthermore, as *On-Chain AuthSC* supports all relevant functionality to create an endorsement for the Registry, we can use the system as described by Groschupp in [33]. Hence, we do not discuss its design and functionality in more detail in our system design chapter. For the detailed explanation we refer to section 2.4.2 and to [33] for its specific implementation.

3.3.2 Sub-Endorsement - Sub-Endorsing the User Account

The sub-endorsement resembles the second link in our endorsement framework. It is added to the Registry and expands trust and associates attributes of SSL/TLS certificates to the sub-endorsed user accounts. Hence, it allows sub-endorsed user accounts to leverage the public confidence of the owner of the Registry to access restricted services at an Application.

Opposed to the endorsement of the Registry, the sub-endorsement of the user account does not require a complex system that expands trust from a blockchain external SSL/TLS certificate to a blockchain internal account. Hence, it is sufficient to store the address of the user account at the Registry, such that other entities can check whether a user account's address is included in the Registry. Thus, the simplest specification of a sub-endorsement only contains an account address of the to-be-sub-endorsed user account. However, in order to indicate the current status of a sub-endorsement, whether it is valid or revoked, we add a status field to its specification. From the expert interviews we learned that the owner of the Registry might want to associate additional information as user specific attributes to the sub-endorsement of a user account. A potential use-case is to leverage these attributes for authentication and access control at Application smart contracts in addition to the attribute of SSL/TLS certificates. This would allow more granular access control decisions, as the attributes specifically describe the user. However, as data on public blockchains can be viewed by anyone, the General Data Protection Regulation (GDPR) of the European Union would most likely be violated. Therefore, and also since the focus of our work is to explore the use of SSL/TLS certificates attributes for authentication and access control, we do not pursue the suggestion for now.
Considering our just described design decisions a sub-endorsement SE comprises the address of the to-be-sub-endorsed user account $address_{user}$ and the status of the sub-endorsement SE_{status} (valid or revoked):

$$SE = \{address_{user}, SE_{status}\}$$

In the following we describe the CRUD (Create, Read, Update and Delete) operations for the sub-endorsement, hence the functionality that needs to be supported by the Registry.

Create: A sub-endorsement can be created by the owner of the Registry, by submitting the account address of the to-be-sub-endorsed user account to the Registry. A sub-endorsement that contains $\mathrm{address}_{\mathrm{user}}$ and the $\mathrm{SE}_{\mathrm{status}}$ is then created and stored. It is not possible to create more than one sub-endorsement for a user account. However, it is important that only the owner of the Registry is allowed to add a sub-endorsement, as otherwise any user could add his or her account.

Read: Any user can retrieve the sub-endorsements from the Registry with the respective getter function. As the data stored on a public blockchain can be read by any user, we do not limit retrieval of the sub-endorsement to only the respective sub-endorsed user account.

Update: The data stored in a sub-endorsement can only be updated by the owner by submitting a new sub-endorsement for the already sub-endorsed user account to the Registry via the *Create* operation. During this operation the previous content of the sub-endorsement is overwritten by the new data.

Delete: An endorsement can not be deleted, but it can be revoked by updating the sub-endorsement status to false (see *Update*). However, only the owner of the Registry is allowed to revoke an endorsement for the same reasons described in *Create*.

Beyond CRUD operations for a sub-endorsements we also consider to include a configuration for the Registry, where the owner can configure if and how sub-endorsements can be used. Such a configuration is especially relevant if rules that affect the functionality of the entire Registry, hence every sub-endorsement, need to be specified. For example, if the private key of the endorsing SSL/TLS certificate of a Registry was stolen, the owner of the Registry might want to immediately deactivate every sub-endorsement. Hence, it should be possible to add and update a *ALLOW_SUBENDORSEMENTS* rule, that specifies whether the sub-endorsements of the Registry are currently active. Without a configuration in the previously described example the owner would need to revoke each sub-endorsement independently via the *Update* operation. In use-cases where the status of every user account needs to be frequently updated, costs and effort would accumulate quickly. From the expert interviews we also learned that rules should not only be defined by us, but should be up to the owner of each Registry. Therefore, we design a configuration for the Registry, which can be extended with individual rules. Hence, the configuration C at a Registry comprises tuples CT, that contain a configuration type $rule_{type}$ and a configuration value $rule_{value}$:

$$C = \{CT_1, ..., CT_n\}, \text{ with } \qquad CT = \{rule_{type}, rule_{value}\}$$

In order to manage the configuration, the Registry needs to support all CRUD operations. However, the functionality that enforces a desired rule needs to be implemented by the rule creator. Nevertheless, to provide an example we include the previously described *ALLOW_SUBENDORSEMENTS* rule, that specifies whether sub-endorsements are currently active at a Registry. The respective functionality, that enforces the *ALLOW_SUBENDORSEMENTS* rule is describe at the access control framework, as it affects the access request evaluation.

3.4 Survey of Access Control Mechanisms

Since the goal of our research is to design authentication and access control at smart contracts, we first explore current standards by conducting an analysis of established access control mechanisms. Although these access control mechanisms are not tailored to blockchain yet, we are still certain to draw insights from established systems for the design of our access control framework.
The goal of access control is to restrict the access to a resource and only grant access as defined by policies to a limited number of users. The term access control is used as a

synonym for authorization and vice versa by [41]. We stick to this convention in our research and consequently define both terms as "[...] the decision to permit or deny a subject access to system objects (network, data, application, service, etc.)" [41, pg.2]. Policies are defined by the resource owner or an appointed administrator and are enforced by the access control mechanism. A policy can be defined as " [...] the logical component that serves to receive the access request from the subject, to decide, and to enforce the access decision" [41, pg.4]. Depending on the manifestation of the mechanism the evaluation of the access control is handled differently. Nevertheless, in all systems a subject is requesting access to a protected object. The subject can either be human or a Non-Person Entity (NPE) as a smart contract. The object is the resource protected by the access control mechanism, to which the subject requests access for [41].
In the following we introduce the four most common access control mechanisms and some of their manifestations. Moreover, we also conduct an analysis of each of the four, to identify the mechanisms to build our system design on. We can not cover all mechanisms and manifestations due to the extensive research which was conducted since the 1960s. This also limits us to only focus on the independent mechanisms and not cover potential combinations of the different mechanisms. Nevertheless, we are confident that this survey and the analysis help to develop the design of our smart contract access control system.

3.4.1 Mandatory Access Control (MAC)

In Mandatory Access Control (MAC) a central authority defines and imposes the regulations which enforce the access control to objects [57]. Users can not acquire the ownership of the object, only the security clearance to interact with it [8]. Subjects that request access to objects are not only users, but also other active entities as processes [46] through which the user interacts with the objects. In order to evaluate access requests, MAC requires that access classes, which are in a partial order, are assigned to all subjects and objects. Each access class is a tuple (SL, C) of a Security Level (SL), as Confidential, Secret or Top-Secret (hierarchically ordered) and a category C which describes the functionality as Development, Finance or Marketing [57]. The access class tuples stand in a relationship to each other, in which one can dominate the other if a certain condition is met. More particularly, access class AC1 is dominated by access class AC2 if and only if AC2's security level is on an equal or higher level in the hierarchical order than the one of AC1 and if and only if AC2's categories includes the ones of AC1 [57]. Access classes can also be incomparable if neither of them dominate the other. The meaning of the classification and hence the outcome of the evaluation, i.e. the process of comparing subjects and objects access classes in order to make an access control decision is depending on which policy the MAC system is implementing. Most common are the secrecy-based and the integrity-based mandatory policies [57]. If confidentiality and integrity is required, it is also possible to combine both policies by assigning one secrecy and one integrity access class tuple to each user, subject and object.

Secrecy-Based Mandatory Policy

The goal of the secrecy policy is to prevent that classified information is leaked to a lower security level, by only providing the user with information that he or she currently needs to get his or her job done. Users can access the system using different subjects, i.e sessions or processes with different access classes, but the subject's security level with which an user interacts in the system always needs to be at an equal or lower level than the user's security level. Here the security level describes the trust needed to access an object, consequently the trust of the system in the subject, i.e. user not to release confidential information. Potential security levels are Unclassified, Confidential, Secret or Top-Secret [57].

Integrity-Based Mandatory Policy

The goal of the integrity policy is to prevent indirect modification of objects by subjects with no write access due to a lower security level. Hence, the system assures the integrity of an object, but not the confidentiality. In this policy the security level describes the trust in the user to interact with objects of different security levels. At objects the security level describes the required trust-level to interact with the object without adding wrong or removing relevant information. Potential security levels are Critical, Important or Low [57]. A user which wants to read in a document, i.e. object which has a low security level, needs to request access to the system via a subject with that low security level. Using that lower level he or she is not able to write objects which have a higher security level. Restricting the subject's capabilities to write to an object with a higher security level, i.e. integrity-level, when reading objects with a lower-integrity level prevents the flow of less assured information form objects with less integrity to objects with high integrity [57].

MAC is a very centralized approach to access control, because only a central entity is allowed to assign security levels and categories. Furthermore, individual resource ownership and role management is not supported [40]. These measures ensure high confidentiality and strong integrity of the protected objects, but also lead to inflexibility and dependency on one authority. Therefore, MAC is often applied by government agencies or the military, where confidentiality and integrity is more important than flexibility and decentralization. Our system however requires that the allocation of access rights are determined in a decentralized manner independently by each owner of the Registry (NFR4). Furthermore, the use of SSL/TLS certificate attributes in the access control process (NFR1) is not natively supported. Enabling access control based on attributes in MAC creates significant overhead only to use a mechanism which is specialized on assuring high confidentiality and integrity - both requirements not prioritized in our research .
Since NFR1 and NFR4 are violated and our use-case does not require such as strong measures to protect the confidentiality and integrity of smart contracts, we do not consider MAC as the preferred access control mechanism for our system.

3.4.2 Discretionary Access Control (DAC)

The central paradigm of Discretionary Access Control (DAC) is that the decision of who can access the resource is at the discretion of the resource owner [64]. Furthermore, its also possible that a subject is endowed by the resource owner with the right to transfer access to other subjects. DAC is often also called an Identity based Access control system, since access control decisions are based on the subjects identity and policies which define the subjects scope of action [57].

Multiple implementations of the DAC were elaborated, sometimes in combination with other access control mechanisms. Access Control Lists (ACL) are one of the most widely used DAC implementations. As well as Authorization Tables and Capabilities, ACL is an implementation of the access matrix model which basically "[...] gives an abstract representation of protection systems" [57, pg.140]. In ACL the objects to-be-protected are assigned each to a column of the matrix, while the subjects which want to access the objects are each assigned to a row. The respective record in a cell of the matrix is the privilege of the subject with regards to the object. An example is shown in Figure 3.3. The entries of the matrix can be modified by the access control administration and the respective resource owners by basic commands [57].

	Doc1	Doc2	Doc3
Alice	Read	Owner Read Write	
Bob	Read Write		Read

Figure 3.3: Exemplary Access Matrix

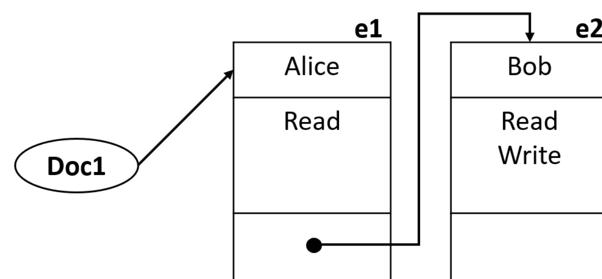


Figure 3.4: Exemplary Access Control List

In ACL a linked list which contains an element for each subject is assigned to each object. The subject's list element contains the respective privileges granted [57]. As an example, in Figure 3.4 the list is assigned to document Doc1 and contains an element e1 associated with the subject Alice, as well as an element e2 associated with the subject Bob. Potential privileges are for example owner, read, write or delete.
In Capability based mechanisms it is the opposite to ACLs. A list which contains elements with the privileges for each object is assigned to each subject. In contrast to the two list based approaches, the *Authorization Table* stores subjects, privileges and objects in a column each, consequently each row is one authorization [57].

DAC, especially access control lists are used by multiple Windows and UNIX operation systems to manage file access [40]. In general, DAC is a rather flexible access control mechanism, among others because it allows other trusted subjects, which are not the owner of a file, to manage the files access rights. As a consequence the system is weaker in security and therefore less suitable for use-cases which require strong access control [64]. Since our system

does not require extremely strong security measures, we could consider DAC for our system. Furthermore, since ACL are not centrally managed and access rights are at the discretion of the resource owner DAC could work well in a decentralized system as a blockchain.
Nevertheless, DAC is not the mechanism of choice as it is in some cases very performance intensive. When we consider ACL, subjects and privileges are assigned via a list to each object. Therefore, it is easy to determine which subject has access to which object. However, it is difficult to determine all the privileges of the user, since they are spread in many different lists [40]. The same issue applies when it is required to determine every user which has access to an object in the Capability based model. Especially for large and/or distributed systems, such a search requires a large amount of performance and data traffic [40]. Hence, the goal to minimize the cost of user management, authentication and authorization (NFR7) is violated. In the case of a blockchain the request could additionally be very slow, since it would require to scan the whole chain to retrieve the desired information. Consequently the revocation of subjects access rights (FR2) might not be feasible in a reasonable timeframe. Furthermore, another requirement, the use of attributes in the access control process (NFR1), is not natively supported and would require a lot of overhead to be included in DAC.

3.4.3 Role-Based Access Control (RBAC)

The central elements of the basic Role-Based Access Control (RBAC) mechanism are the subjects, also called users who want to access a resource, the roles to which users are assigned to and the object, a resource protected by the access control mechanism [44]. Each role resembles a set of assigned permissions, which determine the operations that can be executed on the object by the subject who is a member of that certain role. Common operations are read, write, delete and update. The assignment of a user to a role, as well as the assignment of permissions to roles are managed by the resource owner or a assigned delegate. The assignments are many-to-many relationships to achieve maximum flexibility. A session is created by each user who activates the relevant roles to access a respective protected object. While each user can have an arbitrary number of sessions, each session only has one user.
There is a large number of different RBAC mechanisms and implementations which extend the described basic mechanism illustrated in Figure 3.5.

One common approach is a hierarchical RBAC mechanism [44] [59] [58], in which roles are structured in a hierarchical manner as functions in a company. Consequently more general roles, which are on a higher-level in the hierarchy, inherit the permissions from the lower-level ones. For example, the specific roles of a data science analyst are inherited by the more general ones of the respective data science manager.
Another relevant element, which can be added to RBAC, are constraints [58]. As RBAC systems grow larger and/or become decentralized, a central management of role distribution is not possible or desired anymore. To still guarantee that compliance is enforced and individuals can not be assigned to conflicting roles, constraints are introduced. Potential constraints are the mutual exclusivity of roles, limitation of users per role and the requirement of prerequisite roles before assigning a related role [58].

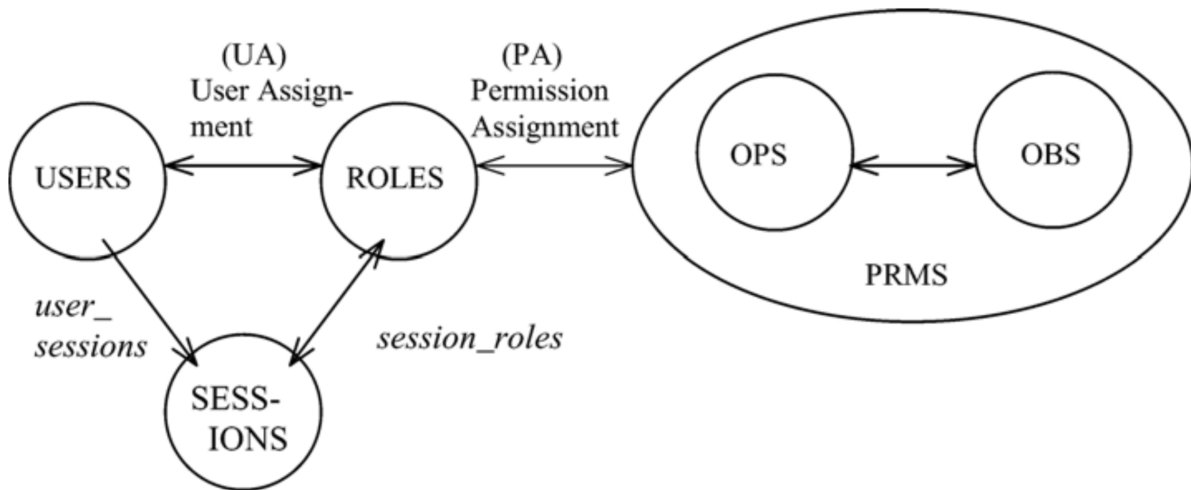


Figure 3.5: Schematic of Role-Based Access Control from [44, pg.4]

The trans-organization RBAC mechanism enables role base access control among users and objects of different organizations [19]. Users and objects are assigned to their respective organization, while the set of rules is split into multiple subset which are each assigned to elements of the different organizations. The role of the subset is managed only by the organization the subset is assigned to, but any user in the system can be assigned to that specific role. Therefore, user access to objects from different organizations are possible, given the user is assigned to the required role [19].

A role-based system enables fast and flexible assignment of subjects to access rights, which can easily be revoked or transferred. The system can be maintained and administered easily, as long as the access control does not get too granular [64]. If granular access control is required, it becomes challenging to manage an increasing number of roles, because for each added permission a verity of new roles with different combinations of the permissions can be created. This might lead to the undesired case where each user has a separate role. For our system this should not be an issue, since the permissions would be based on the limited number of attributes of the SSL/TLS certificate. However, since a requirement is to use the certificates attributes for access control (NFR1), the required multiple layer structure with attributes, permissions and roles would create a significant amount of overhead. This might lead to significant decrease of performance and an increase in cost, thus violate the requirement to minimize cost and impact on performance (NFR7).
Another challenge of RBAC is that managing roles and permissions normally requires a central access control administration which defines and manages roles, i.e. the access control is not defined and executed by the resource owner. A potential solution could be a consensus mechanism, in which a decentralized community could define and link permission and roles. Furthermore, a central role registry could be created where roles are managed. Nevertheless, both solutions would create a significant amount of modification and again require a lot of

overhead, which increases the required performance and cost of our system [40]. Since NFR1 and NFR7 are violated and significant modifications are needed to achieve descentralized sub-endorsement allocation (NFR4), we do not consider RBAC as the preferred access control mechanism for our system.

3.4.4 Attribute-Based Access Control (ABAC)

Attribute-Based Access Control (ABAC), sometimes also referred to as policy based access control, is a more recently developed access control mechanism, which evaluates access requests based on attributes [41]. Attributes describe the characteristics of entities, which are relevant for access control decisions. More precisely, ABAC relies on subject attributes and object attributes, which are together with environmental conditions, evaluated under access control policies in order to determine the outcome of an access control decision. Exemplary, attributes of a document object are file name, owner, edit date and confidentiality. For a subject exemplary attributes are ID, Name and Clearance level [41].
Policies define the logic that determine which combinations of subject and object attribute characteristics are required considering the current environment conditions to be granted with access to the requested object. A policy or a combination of policies, which all are targeted towards the access control of a single object are called access rules. For each object protected by the ABAC at least one policy needs to be defined [41].

If a user submits the access request to the access control mechanism, it needs to decide which policies and whose attributes to retrieve from which source to evaluate the request [41]. Depending on the complexity of the infrastructure this can be managed by a single entity (server) or a system of functional entities. The example depicted in Figure 3.6 by [41] includes the most common entities and describes their interactions within the system.
In order to access an object protected by ABAC a user, i.e. subject has to submit a request to the authorization services. The authorization service consists of the *Policy Enforcement Point (PEP)* and the *Policy Decision Point (PDP)*, which functionality can be either combined on a central server, or be distributed on multiple ones. The PEP receives the access request from the subject and forwards it for evaluation to the PDP, based on which decision it enforces the access to the resource. In order to determine the access control decision, the PDP retrieves the appropriate policies from the *Policy Repository* and the subject's and object's attributes from the *Policy Information Point (PIP)*. To increase the efficiency and performance of retrieval, a Context Handler can also be implemented in the mechanism. It creates an order of retrieval, considering constraints as the urgency of a request to already preload required attributes.
The policies retrieved from the Policy Repository are created, tested, deployed and administered via the *Policy Administration Point (PAP)*. The PIP stores information about the subjects and objects attributes, as their association and the location on one or multiple different Attribute Repositories. Furthermore, it is also able to retrieve the relevant environment conditions, which might be required to evaluate a certain policy. An example for an environment condition is "the current Manager on Duty", if a policy only allows a subject to access a resource, when his or her manager is on duty.

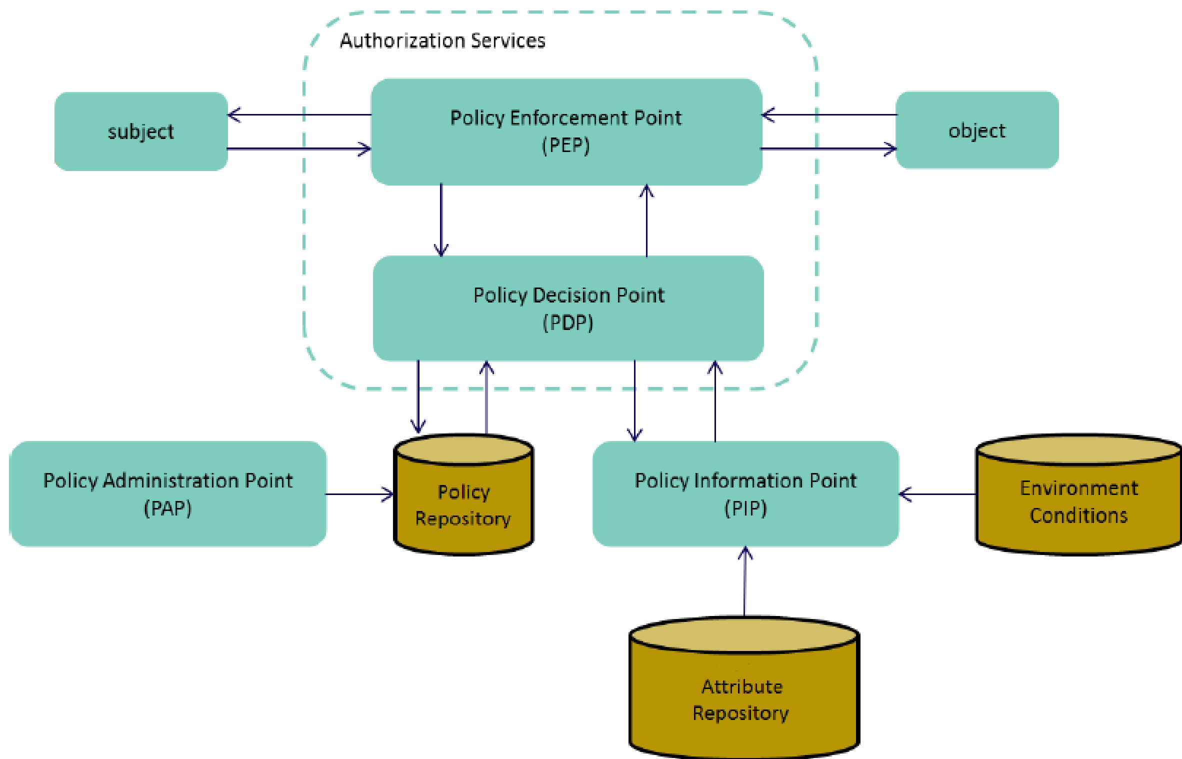


Figure 3.6: Schematic of functional entities of Attribute-Based Access Control from [41, pg.15]

Finally, when the PDP receives the appropriate policy, the subject's and object's attributes as well as the environment conditions, it can assess the subject's request, determine an access decision and respond back to the PEP. Subsequently, the PEP either grants access to the resource or rejects the subject's request.

ABAC can be implemented by multiple programming languages and is used in access control frameworks as the Extensible Access Control Markup Language (XACML) model and the Next Generation Access Control Standard [41]. It is increasingly used in the enterprise context because a large pool of attributes can flexibly describe access rights at any level of granularity [39], while it is less complex and time-consuming to manage than complex permission and role relationships or ACLs [41]. Moreover, it works particularly well with distributed systems [39] - especially due to its support for federation among multiple organization networks without the need for pre-provisioning. The authorization services of the organization networks do not need to locally register the requester individually, but only need to check the attributes of the external requester (i.e. subject) against the attributes required for accessing the object. As long as the separate authorization service trusts the authenticity of the subject's attributes, each can individually evaluate the access request [41].
The strong compatibility of ABAC with distributed systems makes it a good choice for the underlying access control mechanism of our access control framework, as we want to design

an access control mechanism for objects (i.e. smart contracts) in a distributed system (i.e. blockchain). The previously discussed inherent support of federation aligns very well with our requirement to design a system where the access control decision should be evaluated independently of the previous relationship of the subject to the Application (NFR6). Furthermore, our goal to leverage SSL/TLS certificates' attributes for access decisions (NFR1) obviously favors the choice of an attribute-based mechanism. Moreover, there are no apparent incompatibilities of desired functionalities with the ABAC mechanism and opposed to other mechanisms, no increased cost of user management, authentication and authorization (NFR7) that can be directly associated to a system design with ABAC.

Hence, as we want to develop an access control mechanism for objects in a distributed system, which evaluates access requests based on attributes of SSL/TLS certificates, and our analysis of ABAC suggest that the concept helps to meet many of our requirements while it does not violate any, we design our access control framework based on this mechanism. Nevertheless, as for all the other mechanisms evaluated in this chapter, modifications are still necessary to meet all requirements. However, in contrast to the other mechanisms the required modifications are minor and mainly limited to the implementation of ABAC in the context of blockchain, *On-Chain AuthSC* and SSL/TLS certificates. If we design our system based on RBAC, MAC or DAC and still plan to meet all requirements, additional extensive modifications to the general concept of each of the mechanisms would be necessary.

3.5 Creating an ABAC System for Smart Contract Access Control

In this section we elaborate the access control framework for the authentication and access control for smart contracts, that evaluates access requests of blockchain accounts from real-world entities, based on their endowment with trust from SSL/TLS certificates. Furthermore, we integrate the access control framework together with the endorsement framework and *On-Chain AuthSC* into a holistic system design. As we are using ABAC, we may also refer to the access control framework as ABAC framework and the complete access control system as ABAC system.

Attribute-based access control decisions are based on how specific characteristics of the entity, that requests access to the protected resource, evaluate under a certain policy. The policy is defined by the owner of the resource and specifies which attributes are required in which manifestation. In some cases the attributes are very specific to the individual user, however in other cases attributes depend on an affiliation to an organization. As our research leverages attributes of SSL/TLS certificates, which are usually issued for the website of an organization, user accounts are sub-endorsed in the context of their organization. Hence, the system allows to limit access to functionality in an Application for user accounts that share similar attributes of an organization.

In the context of our work an **Application** is a smart contract that offers a function that is

protected by our ABAC system. Any user, that wants to access such a protected function of an Application, needs to be authenticated before the access request is evaluated by the access control mechanism. However, authentication is inherent in blockchain systems. When a user submits a transaction to invoke the function of a smart contract, the transaction needs to be signed with the user accounts private key. During the validation process of the transaction the blockchain network verifies with the public key that the transaction was sent only by the owner of the user account (i.e. owner of the private key) and was not modified during transfer. The verification of the signature is at the same time an authentication, as it is a very strong proof that the sender of the transaction is the owner of the user account. Hence, authentication of the user account is conducted by the blockchain network and thus does not need to be conducted by our access control framework.
Given a successful authentication by the blockchain, the authorization of the user account still needs to be checked. Therefore, the user needs to proof a link between his or her account and attributes of a SSL/TLS certificate, that is trusted by the respective Application. Furthermore, the value of the attributes of the linked SSL/TLS certificate must additionally match the requirements defined by the owner of the Application in a policy. The link between a SSL/TLS certificate's attributes and the account of a user is indirectly established via the endorsement framework we introduced in section 3.3.1. As the endorsement links the certificate to the Registry and via the sub-endorsement the Registry to the user account, the user account is not only sub-endorsed by the certificate itself, but also by its attributes. Hence, if the SSL/TLS Root certificate of the Registry's endorsing certificate is trusted by an Application and its attribute matches the requirements of the policy, then accounts sub-endorsed by the Registry can execute the protected function of an Application.

In this section we explain the previously described concept of our ABAC framework, as well as its functionality and implications in more detail. Therefore, we first discuss the relevant SSL/TLS certificate attributes and introduce the policy framework for the Application in section 3.5.1. The complete architecture of the ABAC framework, as well as its functionality and the interaction of components during access control evaluation are discussed afterwards in section 3.5.2. Finally, in 3.5.3 we walk the reader through the application lifecycle of our system design.

3.5.1 Attributes and Policies in the ABAC System for Smart Contracts

Attributes of SSL/TLS certificates are stored in the X.509 certificate file, that is signed by the private key of a certificate that is higher up in the hierarchy of the PKI (cf. section 2.3.1). Consequently once signed, attributes of a certificate are immutable and endowed with trust. Entities, that successfully validate the SSL/TLS certificate and trust the respective Root certificate, can be certain that the attribute values of the certificate are correct. Furthermore, as the structure of SSL/TLS certificates are standardized, the policy creators at an Application know with which attributes user accounts may be sub-endorsed with. Hence, a second advantage of using attributes of SSL/TLS certificates is that it is possible to conduct access control, independently from a previous relationship between the endorsing certificate and

the Application. However, we are also aware of the downsides of leveraging attributes of SSL/TLS certificates, including decreased flexibility and higher costs because of attribute immutability. Furthermore, the attributes of the certificate primarily describe the subject it is issued for, and not each sub-endorsed user account individually.

Instead of bootstrapping attributes of SSL/TLS certificates we can also leverage a different attribute infrastructure or even create a new one. However, as we previously discussed in section 3.3 we did not identify a different trust infrastructure (centralized or decentralized) for the public blockchain, that provides a sufficient level of trust for attributes. Even DIDs, which support the association of attributes to identities via claims, are not a solution to our problem, as trust endowment still is limited since no Web-of-Trust has been successfully established yet.
If we create a new attribute infrastructure, we can design the structure as desired, hence maximize the flexibility and performance of our system. However, creating a new attribute infrastructure once again requires to endow attributes with trust. A possible solution can be to use SSL/TLS certificates to endorse attributes stored on the blockchain. For example, as we previously described in section 3.3.1 and as one interviewee suggested, attributes can be directly added to the sub-endorsement of the user account. This increases the flexibility of our system, as now every owner of an endorsed Registry can define and assign attributes to users. Hence, this approach allows granular access control decisions, as the attributes specifically describe the user. However, trust in such attributes is much lower than in attributes stored in a SSL/TLS certificate, because these attributes are not verified by the CA. Moreover, there is no standard defined that specifies which attributes are supported by which Registry and Application. Furthermore, attributes describe the user in detail and data on public blockchains can be viewed by anyone, the GDPR of the European Union would be violated.

Considering the advantages and disadvantages of any of the three solutions we conclude that bootstrapping attributes of SSL/TLS certificates is the most feasible and promising solution. Although some disadvantages persist, we are certain that benefits outweigh the challenges and it is therefore worth to use this attribute infrastructure in our system design. Hence, in the following we elaborate how to best apply the attributes of SSL/TLS certificates in our ABAC framework. Subsequently we explain the design of the policy framework for Applications.

Attributes of SSL/TLS Certificates

The attributes, that are relevant for access control policy makers, are the ones that describe the access requesting user. As in our ABAC system the access requesting user is indirectly described by attributes from an organization's SSL/TLS certificate, the attributes that describe the organizations are relevant. These attributes are stored in the *subject*, a sub-field of the *tbsCertificate* field. However, the supported attribute types differ depending on the type of the endorsing certificate. As we already explained in section 2.3.1 there are three different types of SSL/TLS certificates: **Domain-validated (DV) certificates**, **Organization-validated (OV)**

OID	Attribute Type	DV	OV	EV
2.5.4.3	commonName	X	X	X
2.5.4.6	countryName	-	X	X
2.5.4.7	localityName	-	X	X
2.5.4.8	stateOrProvinceName	-	X	X
2.5.4.10	organizationName	-	X	X
2.5.4.11	organizationUnitName	-	X	X

Table 3.1: Attribute types [17] supported by our ABAC system with respective OIDs [37] and certificate types

certificates and **Extended-validation (EV) certificates**. Depending on the certificate type a more sophisticated validation process is required, in order to guarantee the credibility of the attributes. The DV certificates support the least attributes, while EV certificates support the most. In our current system design we only support attributes that are included in the subject field of DV and OV certificates, as these are most commonly used and already support a variety of relevant attributes. Table 3.1 depicts the attribute types supported by our ABAC system, as well as which attribute types are included in which type of SSL/TLS certificate. Further attribute types included in EV certificates and other certificate fields, as certificate extensions, are not supported to minimize complexity of our first prototype and as we did not identify a use-case in which these attributes are relevant for access control. However, if future work identifies such use-cases, our system can easily be extended to support more attribute types.
In the following we provide a brief overview of the supported attributes types, specify their OIDs and provide a fictional example:

- **commonName** (2.5.4.3) - *e.g. www.blockchainuniversity.edu*: The FQDN of the organization.
  Can be used to only limit access to users, that are sub-endorsed by a specific organization domain or top-level-domain.

- **countryName** (2.5.4.6) - *e.g. DE*: The country the organization is located in.
  Can be used to only limit access to users, that are sub-endorsed by an organization from a specific country.

- **localityName** (2.5.4.7) - *e.g. Muenchen*: The city the organization is located in.
  Can be used to only limit access to users, that are sub-endorsed by an organization from a specific city.

- **stateOrProvinceName** (2.5.4.8) - *e.g. Bayern*: The state or province the organization is located in.
  Can be used to only limit access to users, that are sub-endorsed by an organization from a specific state or province.

- **organizationName** (2.5.4.10) - *e.g. Blockchain University*: The name of the organization. Can be used to only limit access to users, that are sub-endorsed by a specific organization.
- **organizationUnitName** (2.5.4.11) - *e.g. IT*: The name of a business unit within the organization.
  As it is possible to have multiple certificates in an organization, the organization unit can be specified in the certificate. The attribute can be used to only limit access to users, that are sub-endorsed by a specific organization's business unit.

Access control evaluation requires access to the attributes of a specific SSL/TLS certificate that endorses a user ccount. Therefore, for each endorsing certificate the attribute values have to be stored on the blockchain. As *On-Chain AuthSC* already stores most information of a SSL/TLS Server certificate and its certificate chain of trust on a on-chain Certificate Database, we can retrieve all attributes required for our work from the central database. However, as the subject is stored as DER-encoded data, we have to complement the Certificate Database with a parser, that extracts the attributes relevant for access control.

Policy Framework for an Application

Policies are the rules that specify which attribute types and values are required to access a protected resource. In our ABAC framework they define the attribute type and the respective value an endorsing SSL/TLS certificate of a user account must specify. Hence, when a user invokes a function of an Application that implements the ABAC framework, the attributes of the SSL/TLS certificate are evaluated under the respective function's policy. As the goal of our research is to first evaluate the feasibility of SSL/TLS certificate based ABAC for smart contracts, our system design only supports very basic policies P, that comprise one attribute type $\mathrm{A}_{\mathrm{type}}$, its value $\mathrm{A}_{\mathrm{value}}$ and a equality comparison operation:

$$P = \{A_{type} = A_{value}\}$$

The policies supported by our framework are stored in and maintained via respective functions of the Application. However, for each protected function only one policy can be defined. Nevertheless, an Application, which implements our ABAC framework, should support the following operations for managing policies:

Create: A policy is automatically created for every function that supports ABAC functionality once the Application is deployed to the blockchain. However, only the attributes specified in the previous section are supported.

Read: Any user can retrieve the current policy of a function from the Application with the respective getter function.

Update: A policy can be updated by the owner of the Application, by submitting a new attribute value and/or the OID of the new attribute.

Delete: A policy can not be deleted, however it may be deactivated. In such case, any sub-endorsed user account can access the function, as no attribute checks are conducted.

3.5.2 ABAC Framework of the ABAC System for Smart Contracts

In the previous sections we discussed design choices and elaborated the attributes and the policy framework for the Application. In this section we want to complement both with an ABAC mechanism to finalize the design of our ABAC framework. The ABAC mechanism defines how an access request for invoking a protected functionality at an Application is evaluated. To retrieve relevant data and execute required functionality for the evaluation process it leverages the endorsement framework and *On-Chain AuthSC*. Hence, we need to consider both together with the ABAC framework to design the architecture of the ABAC mechanism. As we design an attribute-based mechanism, we stick to the established terms and conventions used in ABAC literature (cf. section 3.4.4). However, we also introduce some new terms and components that are specific to our system design.

The ABAC mechanism is distributed among four components on the blockchain: The Application, Registry, Certificate and Endorsement Database. It comprises the functional entities (e.g. Policy Enforcement Point) we described in the survey of ABAC in section 3.4.4. In the following we explain the functionality of the ABAC mechanism, its architecture and our design choices. An overview of the architecture is depicted in Figure 3.7.

The to-be-protected functionality (i.e. object) always resides at an Application, that can be independently deployed from all the other components. However, each Application needs to implement specific functional entities to support the access control mechanism: A *Policy Enforcement Point (PEP)*, that receives the access requests from the user account and forwards the parameters relevant for the access decision to the *Policy Decision Point (PDP_{app})*. The PDP_{app}, is the central component that evaluates user data under a policy stored in the Policy Repository to determine an access decision the PEP can execute. Finally, a *Policy Administration Point (PAP)* allows the owner of the Application to manage the policy framework.
More specifically for a request to be successfully evaluated by the PDP, the user has to provide proof that he or she is sub-endorsed by an intact chain of trust and that the attributes of the endorsing SSL/TLS certificate are successfully verified under the access control policy. However, the *Policy Information Point (PIP)*, that stores the location and procedure to retrieve the user accounts endorsements and attributes can not be implemented in the Application. This is due to the fact that user accounts and hence their sub-endorsing Registry and the attribute providing SSL/TLS certificates are initially unknown to the Application. Furthermore, some functionality that is required to retrieve endorsements and attributes from the Registry, Endorsement and Certificate Database can only be accessed by internal functions. Hence, we design our system as depicted in Figure 3.7 such that a PIP is directly located at each of the

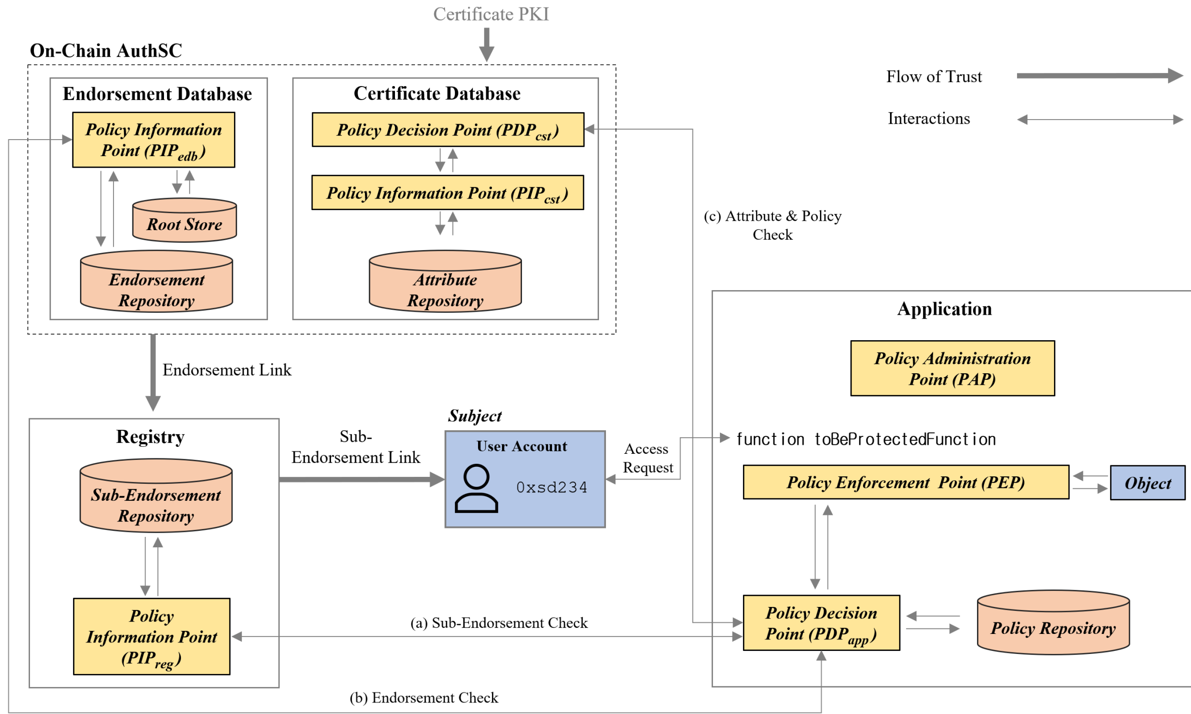


Figure 3.7: Architecture of our blockchain-based ABAC mechanism for smart contracts

three components and the user provides the initial link to the Registry's PIP_{reg} via the access request.

First in *(a) Sub-Endorsement Check* the PDP_{app} at the Application receives the user data from the PEP and initiates contact with the PIP_{reg}. The PIP_{reg} checks if the requesting user account is included in the sub-endorsement repository and returns its response to the PDP_{app}. If a positive response is received, the PDP_{app} successfully verified the sub-endorsement of the user account.
In *(b) Endorsement Check* the PDP_{app} checks the credibility of the Registry, i.e. if the smart contract is endorsed by a trusted SSL/TLS certificate. Therefore, the PDP_{app} submits the Registry's address to the PIP_{edb} of the central Endorsement Database, which checks if an endorsement is included in the Endorsement Repository. Given success, the PIP_{edb} checks the Root Store of the Application to determine if the Root certificate of the endorsing certificate is trusted by the Application. If this is the case, it returns the certificate ID to the PDP_{app} - the chain of trust between the user account and the certificate PKI is successfully validated.
Finally, in *(c) Attribute & Policy Check* the PDP retrieves and validates the attributes under the respective policy. However, as for the parsing of the certificate's subject field at the Certificate Database, the prescribed value of the attribute in the policy is already required, the evaluation needs to be conducted at the Certificate Database. Hence, in our design the PDP_{app} does not execute the evaluation but submits the policy with the ID of the certificate to the Certificate Database. Therefore, besides the Attribute Repository a PIP_{cst} and a PDP_{cst} are required

at the Certificate Database to retrieve the attributes from the Attribute Repository and to evaluate if values match. Once evaluation is completed, the PDP_{cst} returns the outcome to its counterpart on the Application.

Based on the outcome of the three evaluation requests the PDP_{app} determines its access control decision and forwards it to the PEP. Depending on the PDP's$_{app}$ decision the PEP either grants or denies access to the user account.

3.5.3 Application Lifecycle of the ABAC System for Smart Contracts

In this section we summarize our authentication and access control system for smart contracts, that evaluates access requests of blockchain accounts from real-world entities, based on their endowment with trust from SSL/TLS certificates and describe its application lifecycle. We first briefly recapitulate the Registry deployment and the endorsement creation process, as well as the Application deployment. Subsequently we walk the reader through the exemplary evaluation of an access request. For an illustration of the process we refer to Figure 3.7.

Design of the Registry and Endorsement Creation

In order to invoke a protected function of the Application (i.e. object), the user account (i.e. subject) has to be sub-endorsed by a SSL/TLS certificate, which is linked to a Root certificate trusted by the Application. Hence, in this section we briefly describe the endorsement creation, including the Registry deployment. The consequent flow of trust is depicted in Figure 3.7. The process can be described as follows:

1. **Deployment of the Registry**
   The owner of the private key of a SSL/TLS certificate implements a Registry according to our reference design, tests it and deploys it to the public blockchain.

2. **Creation of the Endorsement Link**
   The Registry can be endorsed with *On-Chain AuthSC*, by first submitting the SSL/TLS certificate to the Certificate Database (i.e. Attribute Repository) and second by creating, signing and submitting an endorsement for the Registry to the *On-Chain AuthSC* Endorsement Database (i.e. Endorsement Repository). After successful validation of the signature with the SSL/TLS certificate's public key the endorsement is added to the Endorsement Database[3].

3. **Creation of the Sub-Endorsement Link**
   In this step the account of a user that wants to access an Application that is protected by the ABAC mechanism can be added. In order to do so, the owner of the Registry has to submit the address of the user account to the Registry. Once the sub-endorsement is successfully created, it is added to the Sub-Endorsement Repository. From now on the user account is sub-endorsed and third party entities may check its status.

[3]As the creation of an endorsement is part of *On-Chain AuthSC*, which we described in section 2.4, we only briefly cover the process.

Design of an Application that leverages our ABAC System

The ABAC mechanism can be used by an owner of an Application to protect the Application's functions. In order to add support for our system, the owner needs to comply with the following process:

1. **Deployment of Application**
   A person that wants to use our ABAC system needs to complement the implementation of the Application with the functionality of our reference design. Subsequently the Application can be tested and deployed.

2. **Adding trusted Root certificates**
   Once deployed, the owner of the Application needs to initialize a Root Store for the Application at the Endorsement Database, before he or she needs to decide which Root certificates should be trusted by the Application during the access request evaluation.

3. **Definition of a Policy**
   Finally, the owner of the Application needs to specify the policy at the PAP, hence which attribute type and value needs to be provided by the endorsing SSL/TLS certificate of the access requesting user account.

Evaluating an Access Request of a Real-World Entity

Given a sub-endorsement from a Registry, a request from a user account of a real world entity to access the functionality of an Application may be granted. The evaluation of the access request is depicted in Figure 3.7 and described in the following:

1. **Access Request submission**
   A user account (i.e. subject) selects and invokes a function (i.e. object) of an Application. As payload it submits the account address of its sub-endorsing Registry.

2. **Access Request evaluation**
   The access request is received by the PEP at the Application. It forwards the user account's address as well as the address of the sub-endorsing Registry to the PDP_{app} at the Application. The PDP_{app} initiates the following three step evaluation process.

   (a) **Sub-Endorsement Check**
   The PDP_{app} contacts the PIP_{reg} at the Registry and requests a check whether the user account is sub-endorsed. The PIP_{reg} checks the Sub-Endorsement Repository and responds whether the sub-endorsement is included.

   (b) **Endorsement Check**
   The PDP_{app} contacts the PIP_{edb} at the Endorsement Database and requests a check whether the Registry is endorsed. The PIP_{edb} checks the Endorsement Repository if a valid endorsement for the Registry is included. Given success, it checks in the Root Store whether the endorsing certificate's Root certificate is trusted by the

Application. Finally, the PIP_{edb} submits a response with the ID of the endorsing SSL/TLS certificate to the PDP_{app} at the Application.

(c) **Attribute & Policy Check**
The PDP_{app} contacts the PDP_{cst} at the Certificate Database and provides the to-be-evaluated policy of the Application, as well as the ID of the endorsing SSL/TLS certificate. The PDP_{cst} requests the to-be-checked attribute type and its value as specified in the policy from the PIP_{cst}. It retrieves the attribute type and its value from the respective certificate in the Certificate Repository. Finally, the PDP_{cst} checks whether the attribute type and value match and returns the outcome to its counterpart at the Application.

If any of the three steps fail, the evaluation is aborted and a negative response returned to the PEP. Given that all three steps evaluate successfully, a positive response is returned.

3. **Access Decision**
Depending on the response of the PDP_{app}, the PEP either grants or rejects access to the functionality (i.e object) of the Application.

4 System Implementation

In this chapter we describe and explain the implementation of the prototype for blockchain-based ABAC for smart contracts, based on the system design we introduced in the last chapter. We use the Truffle framework to develop our prototype for the Ethereum blockchain and use the Solidity programming language for implementation. The core component of our prototype is the ABAC framework, that evaluates the access request of a user account at an *Application* smart contract. However, as the user account needs to leverage trust from SSL/TLS certificates, we first need to describe the implementation of the **Endorsement Framework** in section 4.1 before we focus in section 4.2 on the **ABAC Framework**. As our system leverages *On-Chain AuthSC*, we need to consider its design and architecture during implementation. We use its existing functionality but also add some new functions to components of *On-Chain AuthSC*. In this chapter we focus on explaining the functions that we added to the system. Nevertheless, we sometimes refer to functions implemented by Groschupp and provide a brief summary if a functionality is crucial to understand a design decision in our implementation. However, in general we do not explain the implementation of *On-Chain AuthSC*, as it is not part of our contribution. In case of questions regarding the *On-Chain AuthSC* implementation, we refer to section 2.4 and [33].

4.1 Endorsement Framework

The endorsement framework of our prototype is based on our system design in section 3.3. Hence, it comprises a *Registry* smart contract, *On-Chain AuthSC* for the endorsement of the *Registry* smart contract and a sub-endorsement framework for the sub-endorsement of user accounts. It defines the functionality to create and manage a link between SSL/TLS certificates and the *Registry* smart contract, as well as between the *Registry* smart contract and user accounts - a trust endowment from the SSL/TLS certificate PKI to the user accounts. As we apply *On-Chain AuthSC* as specified by Groschupp without any modifications, we do not describe the implementation in this chapter and refer to [33] for a detailed elaboration. Thus, in this section we take the endorsement of the *Registry* smart contract for granted and focus on the implementation of the sub-endorsement framework and the configuration for the *Registry* smart contract.

4.1.1 Sub-Endorsement Framework

The sub-endorsement framework is a set of functionalities that allows to create, obtain, update and delete a sub-endorsement for a user account. It is the central component that allows a user to indirectly obtain trust from SSL/TLS certificates in order to invoke functions protected

by our ABAC mechanism at an *Application* smart contract. Hence, it also offers functionality for *Application* smart contracts to check if a user account is sub-endorsed by the *Registry* smart contract. The sub-endorsement framework is implemented in the *Registry* smart contract. Its most relevant functions and definitions are specified in Figure 4.1.

Sub-endorsements are represented by a `SubEndorsement` struct, that contains an arbitrary `userID` and a boolean `endorsementStatus` that specifies if the sub-endorsement is active or revoked. Each sub-endorsement struct is stored in a central `subEndorsements` mapping at each *Registry* smart contract. As every user account should only be sub-endorsed by one sub-endorsement no key for the mapping, which is unique to each endorsement, is necessary. Hence, we do not directly store the account address of the to-be-sub-endorsed user account in the struct, but use it as the key for the `subEndorsements` mapping. Thus, each sub-endorsement can easily be retrieved from the mapping and associated to the respective user account. Furthermore, it ensures extensibility for future implementations of our system, as other attributes can be assigned to the sub-endorsement struct of each user account.

Managing Sub-Endorsements

The owner of the *Registry* smart contract can create and add a sub-endorsement to the mapping by invoking the `addSubEndorsement()` function. Furthermore, we implement a function for the owner to revoke a sub-endorsement, as well as a getter function that allows a user to retrieve his or her sub-endorsement. In order to limit the accessibility of some functions to the contracts owner, we leverage the `Owned.sol` contract which is part of *On-Chain AuthSC* and drawn from the Open Zeppelin Ownable contract [1]. In the following we describe the different functions in more detail:

- **`addSubEndorsement(address _address)`**:
  A function that can only be invoked by the owner of the *Registry* smart contract to add a new sub-endorsement to the `subEndorsements` mapping. It requires the `address` of the to-be-sub-endorsed user account to create and add a `SubEndorsement` struct to the `subEndorsements` mapping. Once the sub-endorsement is added, the user account is sub-endorsed and may use the *Registry* smart contracts trust to access a protected function at an *Application* smart contract.
- **`revokeSubEndorsement(address _address)`**:
  Revocation of sub-endorsements is needed, as the user might loose access to its sub-endorsed account or the trust of the sub-endorsing *Registry* smart contract's owner. In order to revoke a sub-endorsement, the function needs to retrieve the `SubEndorsement` struct from the `subEndorsements` mapping. Hence, as sub-endorsements are mapped to their account's address, the *Registry* smart contract's owner must pass the account address to the `revokeSubEndorsement()` function. Then the sub-endorsement can be

[1] https://docs.openzeppelin.com/contracts/2.x/api/ownership, visited 13/12/2020

```
1  pragma solidity >0.5.12;
2  import "../Owned.sol";
3
4  contract Registry is Owned{
5      mapping(address => SubEndorsement) public subEndorsements;
6      mapping(string => bool) public config;
7
8      struct SubEndorsement {bool subEndorsementStatus};
9
10     function addSubEndorsement(address _address) public onlyOwner;
11     function getSubEndorsement(address _address) public view returns(...);
12     function revokeSubEndorsement(address _address) public onlyOwner;
13     function checkSubEndorsement(address _address) public view returns(...);
14
15     function updateConfig(string memory _ruleType ,boolean _ruleValue)
           public onlyOwner;
16     function getConfig(string memory _ruleType) public view returns(...);
17 }
```

Figure 4.1: Shortend interface of the Registry smart contract

revoked by updating the `endorsementStatus` in the struct of the sub-endorsements to *false*.

- **`getSubEndorsement(address _address)`**:
  To increase the usability of the sub-endorsement framework, we also allow accounts on the blockchain to easily check the content of sub-endorsements with a `getSubEndorsement()` function. As the data stored on the public Ethereum blockchain can be read by any user, we can not limit access to only the owner or the respective sub-endorsed account. To retrieve a sub-endorsement, the invoker needs to provide the account address of the sub-endorsed account. Subsequently, the function returns the `endorsementStatus` of the respective `SubEndorsement` struct from the `subEndorsements` mapping. However, this function is not invoked by the *Application* smart contract to check the sub-endorsement in an access request, as the evaluation of the request additionally requires to check the `config` mapping of the *Registry* smart contract. We explain its functionality at the end of this section and describe the function `checkSubEndorsement()` that allows *Application* smart contracts to check the status of a sub-endorsement in section 4.2.

4.1.2 Configuration of the Registry Smart Contract

Beyond CRUD operations for a sub-endorsement, we also consider a configuration for the *Registry*, where the owner can configure how the *Registry* can be used. For example, to configure the sub-endorsement framework.

In particular, we are facing the challenge of how the owner of the *Registry* smart contract can quickly deactivate all the endorsements and prohibit access control at the *Application* smart contract. A possible, but resource intensive implementation, is to revoke every single sub-endorsement via a function call of the `revokeSubEndorsement()` function. However, this is neither time nor Gas cost efficient, especially if the timeframe for the deactivation only is intended to be limited.
Another approach is to introduce a configuration variable, that specifies whether sub-endorsements are currently supported. If activated, sub-endorsements can be leveraged to check access control. Once it is deactivated, each request of an *Application* smart contract to check a sub-endorsement is rejected. Using this implementation, it allows the owner of the *Registry* smart contract to efficiently activate and deactivate sub-endorsements without actually revoking sub-endorsements. Therefore, we pursue the second approach. Moreover, during our interviews to evaluate the design of our system we learned that owners would like to have the opportunity to add other individual configuration rules to the *Registry* smart contract. Hence, we implement a `config` mapping, that maps the type of the configuration rule to its value. A new rule can be added and an existing one can be updated by invoking the `updateConfig()` function. Furthermore, any interested account can obtain the current configuration of the *Registry* smart contract with the `checkConfig()` function.

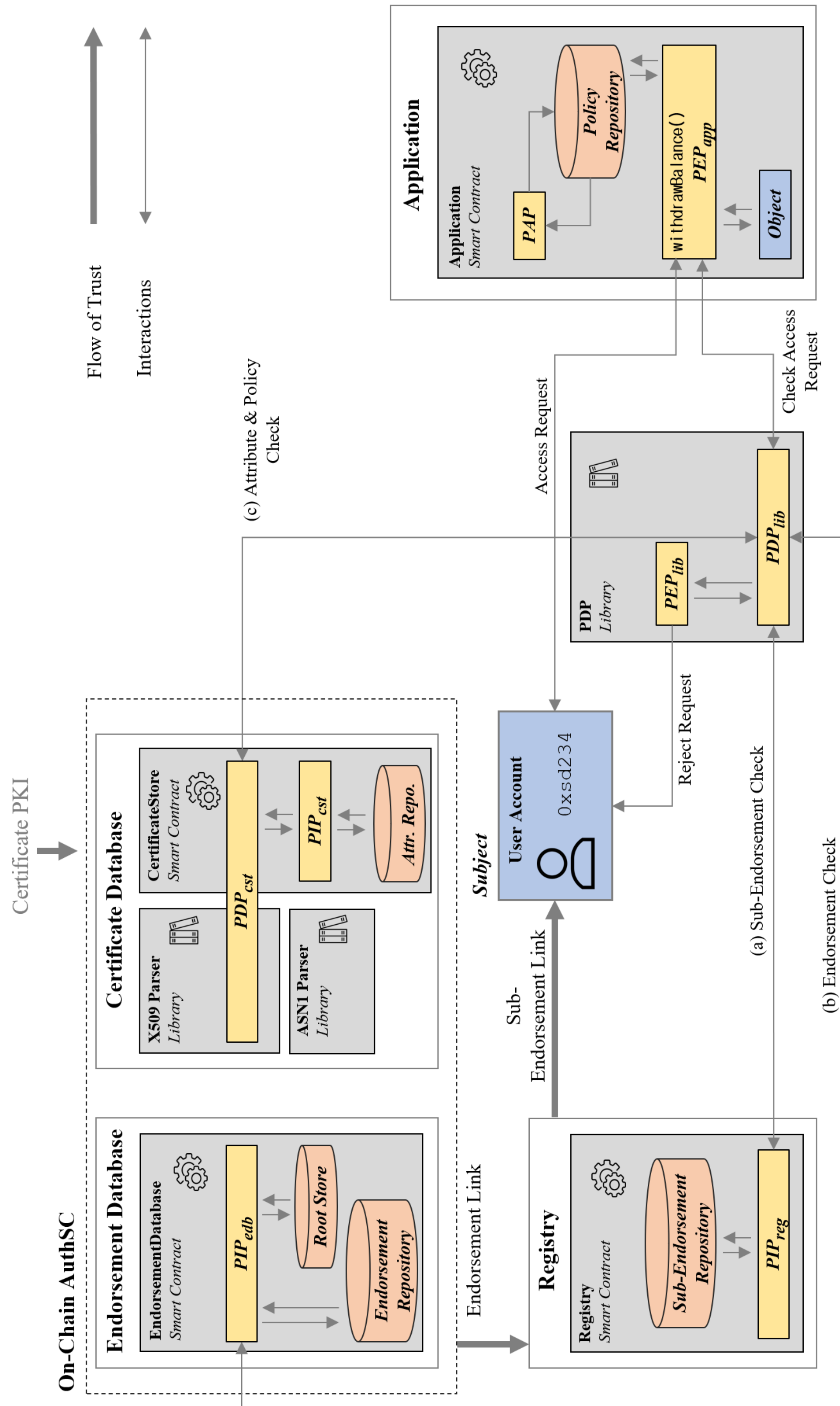


Figure 4.2: Model of prototype implementation

4.2 ABAC Framework

Our prototype implements the ABAC framework for smart contracts and leverages the sub-endorsement framework of the *Registry* smart contract, we described in the previous section, and *On-Chain AuthSC* by Groschupp [33]. It is based on the system design in section 3.5, hence comprises smart contracts and libraries that implement the central components that an ABAC architecture consists of: *Policy Enforcement Point (PEP)*, *Policy Decisions Point (PDP)*, *Policy Information Point (PIP)*, *Policy Administration Point (PAP)* and relevant repositories. As we bootstrap an existing prototype and want to reduce code duplication, the components can not always be implemented by only one smart contract or library, hence represent functionality from multiples ones. An overview of the systems components, smart contracts and libraries, as well as their interactions are depicted in Figure 4.2.

The PAP is implemented in the *Application* smart contract, the PDP is resembled by a *PDP* library and the PEP is shared between both - the PEP_{app} and the PEP_{lib}. The *Application* smart contract uses the *PDP* library to check the sub-endorsement of the requesting user account with the *Registry* smart contracts PIP_{reg} and the endorsement of the *Registry* smart contract with the PIP_{edb} of the *EndorsementDatabase* smart contract. Furthermore, the *PDP* library is used to evaluate the attributes of the endorsing SSL/TLS certificate together with a PDP_{cst} and PIP_{cst} at the *CertificateStore* smart contract and *X509Parser* library. The relevant helper functions to parse the SSL/TLS certificate data are provided by the *ASN1Parser* library.
In the following we discuss design decisions that affect the general architecture of our prototype in section 4.2.1, before we discuss the implementation of its components in sections 4.2.2 to 4.2.6.

4.2.1 ABAC Prototype Architecture

In this section we explain major design decisions that affect the structure and components of our prototype implementation. At first we discuss how and in which components to implement the PAP, PEP and PDP. Subsequently we elaborate the implementation of the PIP. The structure of the prototype is depicted in Figure 4.3.

Implementation of the PAP, PEP and PDP

Our ABAC mechanism is applied to restrict access to a function at a smart contract. We specified the name of such a smart contract already in the system design as Application. However, the Application we described in the System Design chapter in 3.5.2 differs from the actually implemented *Application* smart contract in our prototype.

A question, that arises when implementing the Application in a smart contract, is how and where to best implement the PAP, PEP and PDP. A first solution is to include all three components in the *Application* smart contract - hence implement the *Application* smart contract with every functionality described in the system design section. This is the most straight

forward approach, however it creates a large amount of duplicate code that needs to be stored on the blockchain in each deployed instance of an *Application* smart contract. Furthermore, it leaves room for errors and security issues, as each owner has to implement the code individually.
A second option is to split the functionality of the Application into an *Application* smart contract and an abstract smart contract. The *Application* smart contract can implement functions of the PAP and PEP, which are specific to each individual instance of the Application, while the abstract smart contract, from which each *Application* smart contract inherits, defines the common code. As the owner of the *Application* smart contract does not need to implement most of the logic, security issues and errors are mitigated. Nevertheless, since Solidity copies the code from an abstract contract into the inheriting one, duplicate code and increased Gas costs still are an issue.
Therefore, we explore a third solution, where we apply a similar separation of functionality, but use a library instead of an abstract contract. The library stores the common code of the PEP and PDP and allows multiple *Application* smart contracts to use its functionality without duplication of code. Yet a library can not store Ether, does not support modifiers and state variables. Nevertheless, we still implement a library, as we can use function calls instead of modifiers and store the state variables in the *Application* smart contract. Thus, we define and store the PAP state variables and implement its functionality directly at the *Application* smart contract. Furthermore, we implement some of the functionality of the PEP (PEP_{app}) at the *Application* smart contract and the common code (PEP_{lib}) together with the PDP_{lib} at the *PDP* library.

Implementation of the PIP and Implications for the PDP

In order to evaluate an access request, the *PDP* library needs to check the validity of the sub-endorsement and endorsement, as well as the relevant attribute values from the SSL/TLS certificates. The PDP depends on the PIP to retrieve the relevant information from the *Registry*, *EndorsementDatabase* and *CertificateStore* smart contracts. Hence, in this section we elaborate how and where to implement the PIP, in particular under consideration of its design in section 3.5.2, where we defined the PIP_{reg}, PIP_{edb} and PIP_{cst}.

The PIP_{reg} retrieves the information whether the access requesting user account is sub-endorsed by the *Registry* smart contract from the `subEndorsement` mapping at the *Registry* smart contract. First we consider an implementation of the PIP's$_{\text{reg}}$ functionality at the *PDP* Library. Such implementation reduces duplicate code and decreases Gas costs. However, as we described in section 4.1.2, we want to enable the owner of a *Registry* smart contract to implement an individual configuration for the *Registry* smart contract. Depending on the specification of the configuration rules defined by the owner, information retrieval from the PIP is affected. For example, if the owner adds a rule, that deactivates sub-endorsements for a certain timeframe, the PIP needs to consider the rule and return that the user account is not sub-endorsed. Hence, the PIP's implementation depends on the individual configuration of each *Registry* smart contract. Therefore, it needs to be implemented at the *Registry* contract.

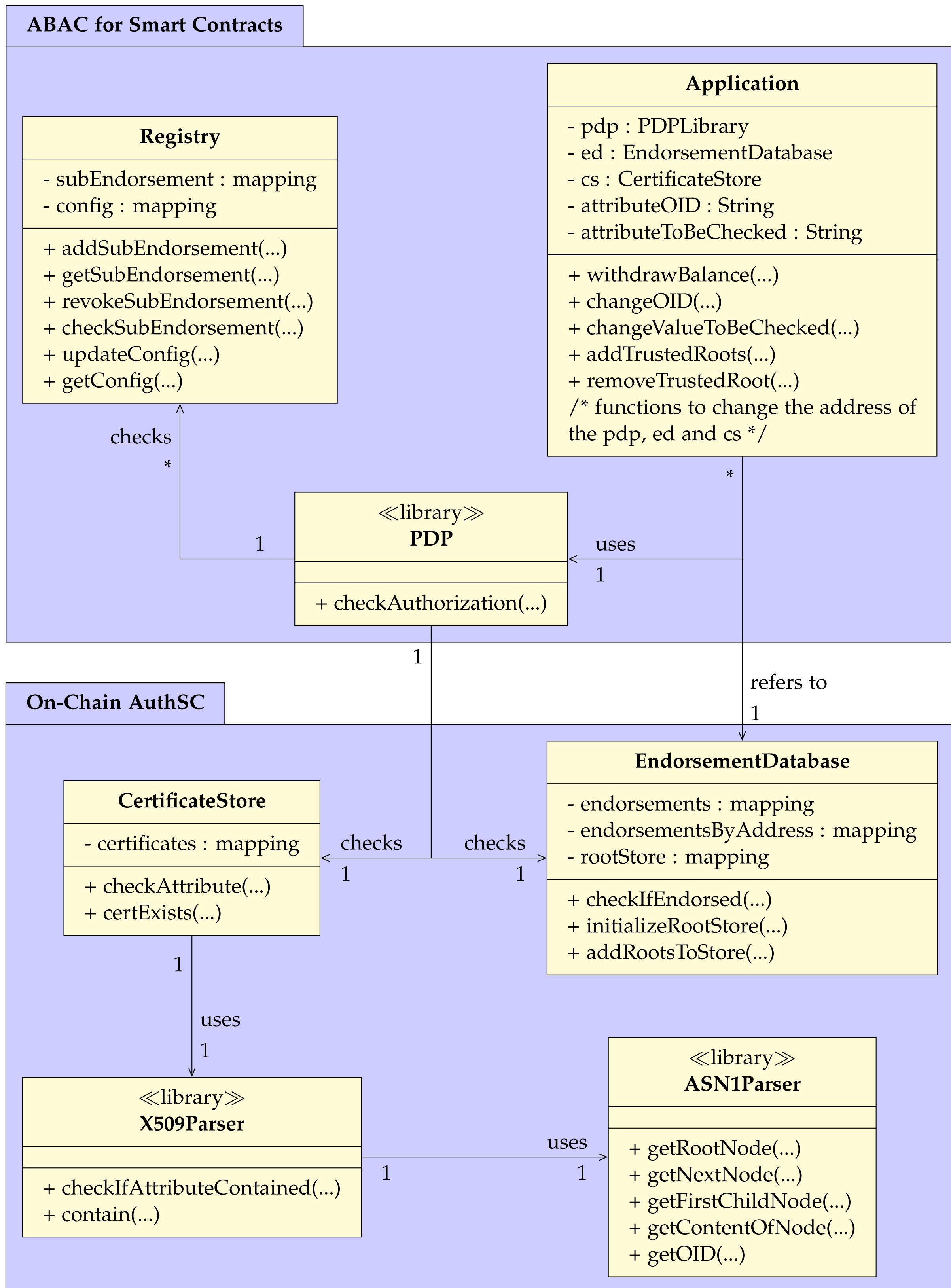


Figure 4.3: Structure of the prototype implementation with components from [33]

For the evaluation of the *Registry* smart contract's endorsement and the SSL/TLS certificate attributes, we bootstrap *On-Chain AuthSC*. Hence, we need to consider its architecture in our implementation to ensure functionality and efficiency of the prototype. As the *PDP* library relies on the Endorsement Database to check the endorsement of a *Registry* smart contract and on the Certificate Database to check the attributes of the endorsing SSL/TLS certificates, the PIP_{edb} and PIP_{cst} need to be developed in accordance with the implementation of these two entities from *On-Chain AuthSC*.

The Endorsement Database comprises the *EndorsementDatabase* smart contract and a *SignatureValidation* library. However, the *SignatureValidation* library is not relevant for this section, because it provides enabling functionality which we assume as given since it does not directly affect our implementation. The *EndorsementDatabase* smart contract does not implement a function which allows to retrieve an accounts endorsement status with its address. Therefore, we are confronted with the decision to either implement such functionality in the *PDP* library or directly at the *EndorsementDatabase* smart contract. As the mapping, that stores the endorsements is set private, it can only be accessed from a function within the *EndorsementDatabase* smart contract. Hence, to check the endorsement from outside the *EndorsementDatabase* smart contract, we need to change large parts of the existing implementation of *On-Chain AuthSC*. This most likely leads to duplicate code, hence increases complexity and deployment costs. Thus we do not significantly change the implementation of *On-Chain AuthSC* and implement the functionality of the PIP_{edb} directly at the *EndorsementDatabase* smart contract.

The Certificate Database comprises the *CertificateStore* smart contract, the *X509Parser* library and the *ASN1Parser* library. Once again we are confronted with the challenge where to implement the PIP, more specifically the PIP_{cst}. In order to retrieve the attributes from the SSL/TLS certificate and evaluate them under the policy of the *Application* smart contract, the ABAC mechanism needs to check first whether the specified SSL/TLS certificate exists. Given that the SSL/TLS certificate is stored in the *CertificateStore* smart contract, its DER-encoded subject field needs to be parsed with the *X509Parser* library to extract the attribute information. Consequently, we can either directly evaluate the policy with the *X509Parser* library at the *CertificateStore* smart contract, hence move some of the PDP functionality away from the *PDP* library, or retrieve the attribute value from the *CertificateStore* smart contract and evaluate it under the policy at the *PDP* library. However, the decoding of the DER-encoded subject field and the extraction of the stored data is very complex. Therefore, we do not extract and decode the field, but only search for the attribute type and value specified in the policy to determine if both are included in the subject field. Hence, we are forced to check the policy requirements with the *X509Parser* library at the *CertificateStore* smart contract and not at the *PDP* library. Consequently, the attribute retrieval and evaluation is implemented in a PIP_{cst} and a PDP_{cst} at the *CertificateStore* smart contract and the *X509Parser* library. As for each access request evaluation the subject field needs to be parsed, this implementation has a negative influence on the performance and Gas costs of the prototype. Nevertheless, given the complexity to develop a sophisticated DER-Decoder and to deploy it on the blockchain,

we are certain that this solution is most feasible for the scope of this thesis. However, future developments should consider to develop such decoder to extract and store each attribute type and its value once during submission of the SSL/TLS certificate to the *CertificateStore* smart contract. Thus, Gas cost can be significantly reduced for the evaluation of an access request, as no parsing and decoding is required anymore.

4.2.2 Application Smart Contract (PEP_{app}, PAP)

The Application functionality is implemented by each Application owner. We only provide a reference implementation of the ABAC mechanism, the Application owner has to complement with desired functionality. The reference implementation for an *Application* smart contract comprises functionality for the PEP_{app}, that manages the access request from the user account and the PAP, that allows an owner to manage policies and the configuration of the ABAC mechanism. In order to limit accessibility of some functions to the *Application* smart contract's owner, we once again leverage the `Owned.sol` smart contract.
For demonstration purposes of the ABAC mechanism in this prototype we implement and protect a simple function `withdrawBalance()`, that allows an authorized user to withdraw all Ether from the *Application* smart contract. An overview of the functions and variables of our reference implementation of the *Application* smart contract is listed in Figure 4.4. In the following we discuss the reference design in more detail and explain our design choices.

```
1  pragma solidity >0.5.12;
2  import "../Owned.sol", "../PDP.sol", "../EndorsementStore.sol", "../
       CertStore.sol";
3
4  contract Application is Owned{
5      PDP pdp;
6      EndorsementDatabase es;
7      CertificateStore cs;
8      string public attributeOID;
9      string public valueToBeChecked;
10     /* Omitted variables for addresses */
11
12     function withdrawBalance(address _addressEndorser) public;
13     function changeOID(string memory _attributeOID) public onlyOwner;
14     function changeValueToBeChecked(string memory _valueToBeChecked) public
           onlyOwner;
15     function addTrustedRoots(...) public onlyOwner;
16     function removeTrustedRoot(...) public onlyOwner;
17     /* Omitted functions to change addresses and enable Ether use-case */}
```

Figure 4.4: Shortend and simplified interface of the Application smart contract

The PEP_{app} - Receiving an and Responding to an Access Request

A possible implementation of the PEP_{app} at the *Application* smart contract is to implement it in a separate function. The user has to invoke the function that the ABAC mechanism can check whether he or she is permissioned to access a protected function. Given success, the user is added to a whitelist and may then invoke and access the to-be-protected function. However, this solution requires two interactions of the user with the *Application* smart contract. A second approach is to design and implement the access request evaluation in a non-interactive way. Hence, execute it when the to-be-protected function is invoked, but before its actual content is executed. The function head is part of the PEP_{app}, as by invoking the function and providing the address of the user's sub-endorsing *Registry* smart contract, the user account indirectly submits an access request. Therefore, the only extra effort that the user needs to make is to provide the address of the sub-endorsing *Registry* smart contract. Furthermore, this also allows the *Application* smart contract to use our ABAC mechanism for multiple functions, which apply different policies, as each request is individually managed at the respective function. After considering the benefits and challenges we implement the second approach in our prototype at the `withdrawBalance()` function.

The evaluation of the access request is conducted at the *PDP* library and not directly at the *Application* smart contract. Hence, the PEP_{app} forwards an access request, including the address of the user account and its sub-endorsing *Registry* smart contract, to the *PDP* library. Once the *PDP* library evaluated the access request, it returns a boolean, that specifies whether access to the functionality is granted or denied. The PEP_{app} then either approves or aborts the execution of the protected functionality by the user.

The PAP - Policy, Root Store and Update Management

In this section we describe the functionality supported by the PAP at the *Application* smart contract: Administration of the policies for access request evaluation and management of trusted SSL/TLS Root certificates.

In our prototype the currently applied policy for the *Application* smart contract is specifed by an `attributeOID` and a `valueToBeChecked` variable stored in the state of the *Application* smart contract. As this is a first prototype, it is only possible to define one policy for all functions of an *Application* smart contract.
Given a policy is defined, the ABAC mechanism checks at the *PDP* library, whether the attribute type (i.e. `attributeOID`) specified by the owner of the *Application* smart contract matches the attribute type in the endorsing SSL/TLS certificate and if the value (i.e `valueToBeChecked`) assigned to both attribute types is equal. However, an owner of an *Application* smart contract might want to change the policy that specifies the access requirements for a to-be-protected function. Hence, we implement a `changeOID()` function to change the attribute type (i.e. `attributeOID`) and a `changeValueToBeChecked()` function that allows to change the attribute value (i.e `valueToBeChecked`).

Beyond policy management, we are confronted with the question, whether it is beneficial to add functionality to the *Application* smart contracts PAP, that allows the initialization and management of its Root Store. The Root Store is a component of *On-Chain AuthSC* that stores the SSL/TLS Root certificates that are trusted by the *Application* smart contract. Only if the SSL/TLS Root certificate of the user account endorsing SSL/TLS certificate is included in the *Application* smart contracts `rootStore` mapping at the *EndorsementDatabase* smart contract, the attribute and policy evaluation is conducted.
On-Chain AuthSC already provides functions at the *EndorsementDatabase* smart contract with which a Root Store of an account can be created and managed. Nevertheless, to improve usability, we initialize a Root Store through a function call in the constructor of the *Application* smart contract. Furthermore, we implement the two helper functions `addTrustedRoots()` and `removeTrustedRoot()`, that allow the owner to add and remove trusted SSL/TLS Root certificates without the need to manually execute functions at the *EndorsementDatabase* smart contract.

During the implementation of our prototype we also elaborate how to deal best with updates to and the replacement of smart contracts that are used by our ABAC mechanism. Smart contracts are immutable, therefore in case of an update a new smart contract with a new address has to be deployed on the blockchain. To ensure that smart contracts in our system can be updated, we need to make sure that the variable that store the addresses of the to-be updated contract can be updated.
As the PDP_{lib} is implemented in a library, it can not store any state variables. Hence, the addresses of the *EndorsementDatabase* smart contract and the *CertificateStore* smart contract - both are needed for access request evaluation - have to be passed as payload of the `checkAuthorization()` function to the *PDP* library. As we do not want the user to provide those addresses to the *Application* smart contract, due to usability and security reasons, the two addresses have to be stored together with the address of the *PDP* library in the state of each *Application* smart contract. In order to ensure that the *Application* smart contract still works in case any of the three smart contracts is replaced by a new one, we implement the `changeLibraryAddress()`, `changeEndorsementDatabaseAddress()` and `changeCertificateStoreAddress()` functions with which the owner can update the addresses of the respective smart contracts.

4.2.3 PDP Library (PDP_{lib}, PEP_{lib})

The *PDP* library only defines a single function that implements the functionality of the PDP_{lib}: The `checkAccess()` function, that retrieves and evaluates information from other smart contracts PIPs. More specifically, in our system it checks the three requirements that a user account needs to satisfy in order to be granted access to the protected function: A valid sub-endorsement at the *Registry* smart contract, a valid endorsement of the *Registry* smart contract stored in the *EndorsementDatabase* smart contract and attributes of the endorsing SSL/TLS certificate that successfully evaluate under the policy defined by the owner of the

Application smart contract.

A potential implementation of the `checkAccess()` function that executes the described functionality first retrieves the information from each of the three smart contracts, before jointly evaluating it. However, if the check of the first of the three conditions turns out to be negative, resources are wasted retrieving data that is no longer needed. Therefore, we propose a sequential structure as depicted in Figure 4.5, in which the retrieved data is immediately evaluated. Furthermore, we suggest to start with the validation of the user accounts sub-endorsement, or the *Registry* smart contracts endorsement, as the parsing of the SSL/TLS certificate is much more computation intensive than the other two functionalities.

Validation of the Sub-Endorsement

As we previously explained, in the `checkAccess()` function we first implement the functionality that checks whether the access requesting user account is sub-endorsed by the specified *Registry* smart contract. To do so, the PDP_{lib} invokes the `checkIfEndorsed()` function at the respective *Registry* smart contract and provides the address of the access requesting user account to the *Registry* smart contract. After evaluation the function returns a boolean value, that specifies if a sub-endorsement is valid and if the user account is allowed to use it for access control purposes. Next the PDP_{lib} immediately evaluates the value. If it returns true, the endorsement of the *Registry* smart contract is evaluated in the next step. However, if it returns false the evaluation is aborted and the user account informed via an error message that his or her request is rejected. As the error message is directly submitted from the *PDP* library, the PEP is shared between the *Application* smart contract and the *PDP* library.

Validation of the Endorsement

The evaluation process of the *Registry* smart contracts endorsement at the *PDP* library is very similar to the one of the sub-endorsment. However, this time the `checkIfEndorsed()` function at the *EndorsementDatabase* smart contract is invoked. In case of success the endorsing SSL/TLS certificate's ID is returned instead of a boolean value. It is required to identify the endorsing certificate of the user account for the policy evaluation in the next step.

Attribute and Policy Evaluation

The attribute and policy evaluation is the most complex step of our ABAC mechanism, as we need to parse the SSL/TLS certificate's subject field for the to-be-evaluated attribute type and the attribute value. Furthermore, we need to implement the evaluation in compliance with the structure of *On-Chain AuthSC*. Hence, as we already explained in section 4.2.1, we can not conduct the evaluation at the *PDP* library and therefore have to implement a PDP_{cst} at the *CertificateStore* smart contract and the *X509Parser* library. Nevertheless, the *PDP* library already checks if the attribute type's OID, specified by the *Application* smart

```
1  pragma solidity >0.5.12;
2  import "../Registry.sol", "../EndorsementStore.sol", "../CertStore.sol";
3
4  library PDP {
5
6      function checkAuthorization(address _addressRequester, address
           _addressEndorser, address _addressEndorsementStore, address
           _addressCertificateStore, uint256 _rootStoreIndex, string memory
           _attributeOID, string memory _valueToBeChecked) public view returns(
           bool){
7
8      Registry reg = Registry(_addressEndorser);
9      EndorsementDatabase ed = EndorsementDatabase(_addressEndorsementStore);
10     CertificateStore cs = CertificateStore(_addressCertificateStore);
11     bytes32 certID;
12
13     /* Sub-endorsement validation */
14     require(
15         reg.checkSubEndorsement(_addressRequester),
16         "Access Denied: You are not endorsed by the contract you specified!"
17     );
18
19     /* Endorsement validation */
20     certID = ed.checkIfEndorsed(_addressEndorser, _rootStoreIndex);
21     require(
22         certID != 0x00,
23         "Access Denied: The endorsement of your endorsing contract is not
               valid!"
24     );
25
26     /* Attribute and policy evaluation, this example checks "commonName" */
27     if(_attributeOID == "2.5.4.3"){
28         require(
29             cs.checkAttribute(certID, _valueToBeChecked, _attributeOID),
30             "Access Denied: The commonName of your endorsing contract's
                   certificate does not authorize you to use this Application!"
31         );
32         return true;
33     }
34     /* Omitted else if cases, that check the other supported OIDs */
35     }
36 }
```

Figure 4.5: Shortend and simplified implementation of the PDP library

contract, is supported by the ABAC system. We consider the following design choices for our implementation.
A first possible implementation is to store a mapping of supported attribute types at the *PDP* library. For each access request the *PDP* library checks whether the attribute type, desired by the *Application* smart contract's policy and passed as parameter of the `checkAccess()` function to the *PDP* library, is included in the mapping. This approach is cost-efficient, however as libraries can not store state variables we are not able to store the mapping of the supported attribute types at the *PDP* library. Hence, we compare each supported attribute type with the requested one in a separate if-block. We are aware that this creates duplicate code and hence increases cost for the deployment of the *PDP* library. However, as only one instance of the *PDP* library is deployed, the costs are limited.
If in any of the cases a supported attribute type matches the requested one, the OID and the desired value of the attribute type are submitted as payload of the `checkAttribute()` function to the PDP_{cst} at the *CertificateStore* smart contract. After the evaluation is finalized, the function once again returns a boolean variable. Given a successful evaluation, the PEP_{app} at the *Application* smart contract is ordered to grant access. In case of failure an error message is submitted to the user account.

4.2.4 Registry Smart Contract (PIP_{reg})

In section 4.1 we already discussed the sub-endorsement framework and the *Registry* smart contract. Yet, we did not describe the implementation of the PIP_{reg} in the *Registry* smart contract, as it is a component of the ABAC framework. The PIP_{reg} at the *Registry* smart contract allows the PDP_{lib} at the *PDP* library to retrieve the status of a sub-endorsement from the `subEndorsement` mapping at the *Registry* smart contract. As the `subEndorsement` mapping's visibility is not set to private, in theory the implementation of the PIP_{reg} can reside at the `PDP` library. Thus, the amount of duplicate code at the multiple instances of *Registry* smart contracts can be decreased. However, as we described in section 4.1.2 and 4.2.1, we want to enable the owner of the *Registry* smart contract to implement an individual configuration to better manage the *Registry* smart contract. Yet, depending on the specification of the configuration rules defined by the owner, the information retrieval from the PIP is affected. Hence, to consider individual implementations that enforce the configuration rules at the *Registry* smart contract, the PIP_{reg} needs to be implemented at the *Registry* smart contract.

The PIP_{reg} is implemented in the `checkSubEndorsementFunction()` of the *Registry* smart contract and comprises two components.
The first checks the requirements specified by the owner in the configuration of the *Registry* smart contract, i.e `config` mapping. As its content differs at each *Registry* smart contract, the functionality needs to be implemented by the owner of the *Registry* smart contract. However, we provide a reference implementation in our prototype, that allows the owner to activate or deactivate all sub-endorsements, i.e. approve or disapproves access control checks for sub-endorsed user accounts.
In the second component of the function the ABAC mechanism checks whether an active sub-

endorsement is stored in the `subEndorsement` mapping (i.e. Sub-Endorsement Repository). In order to conduct the check, the *PDP* library needs to pass the address of the to-be-checked user account to the `checkSubEndorsementFunction()` at the *Registry* smart contract.
If any of the checks at the two components fail, a negative response is returned to the *PDP* library. In case of success, a positive response is submitted.

4.2.5 EndorsementDatabase Smart Contract (PIP_{edb})

The *EndorsementDatabase* smart contract is the central component of *On-Chain AuthSC*, where external endorsements are stored and managed. It implements functionality, that allows to add an endorsement for an account on the blockchain, as well as functions for updates and revocation. Hence, as we leverage *On-Chain AuthSC* to create a trust-link between a SSL/TLS certificate and the *Registry* smart contract, the endorsement of the *Registry* smart contract is created with and stored at the *EndorsementDatabase* smart contract.
The validity of the *Registry* smart contracts endorsement is one of the three conditions that need to be checked by the ABAC mechanism during the evaluation of a user accounts access request. Thus, we need to implement a PIP_{edb} at the *EndorsementDatabase* smart contract that checks whether the endorsement of the *Registry* smart contract is stored at the *EndorsementDatabase* smart contract and is valid. Furthermore, it needs to check if the SSL/TLS Root certificate of the endorsing SSL/TLS certificate is included in the respective `rootStore` mapping of the *Application* smart contract, i.e if it is trusted by the *Application* smart contracts owner.

As the *EndorsementDatabase* smart contract does not offer a function, that allows to check every described condition, we complement it with the PIP_{edb} implemented in a `checkIfEndorsed()` function. The design of the function is predefined by the structure and components of the *EndorsementDatabase* smart contract. Furthermore, as we previously discussed in section 4.2.1, the mapping that stores the endorsements is set private, hence we need to implement the function at the *EndorsementDatabase* smart contract.
At first we retrieve every endorsement of the respective *Registry* smart contract from the `endorsementsByAddress` mapping (i.e Endorsement Repository). The key of the mapping is the account address, that maps to the accounts respective `EndorsementStore` struct. Hence, we need to pass the *Registry* smart contract's account address as payload of the `checkIfEndorsed()` function from the *PDP* library to the *EndorsementDatabase* smart contract to retrieve the required `EndorsementStore` struct. The `EndorsementStore` struct stores each endorsement of the respective *Registry* smart contract, no matter if it is still valid or already revoked or expired. Therefore, we only retrieve the most recently added endorsement, that is neither revoked nor expired, and check if its SSL/TLS Root certificate is included in the *Application* smart contract's `rootStore` mapping. Given the `checkIfEndorsed()` function identified a matching endorsement, it returns the endorsing certificates ID to the *PDP* library. In case of failure, it returns an ID that only contains zeros.

4.2.6 CertificateStore Smart Contract and X509Parser Library (PDP_{cst}, PIP_{cst})

On-Chain AuthSC stores SSL/TLS certificates in a central *CertificateStore* smart contract. However, the SSL/TLS certificates are not stored as a single file. Once submitted, the most relevant information is extracted by the *X509Parser* library with support of the *ASN1Parser* library. Unfortunately, the attributes in the subject field of the SSL/TLS certificate, which our *PDP* library aims to retrieve from the *CertificateStore* smart contract, are not parsed yet and are still DER-encoded. As we previously explained in section 4.2.1, as the certificates mapping as well as other functions in the *CertificateStore* smart contract and *X509Parser* library are not accessible from an external smart contract, we have to complement both with functionality to retrieve and parse the subject field data. Fortunately, we can leverage many functions, that are already part of *On-Chain AuthSC*.
Moreover, as we explained in section 4.2.1, we do not extract and store each attribute type and its value in the *CertificateStore* smart contract during submission of the certificate, but instead search if the to-be-checked attribute type and its value are included in the DER-encoded subject field during each evaluation of an access request. Therefore, as we implement the policy evaluation at the *X509Parser* library, we thus not only implement the PIP_{cst} at the *CertificateStore* smart contract, but also a PIP_{cst} and PDP_{cst} at the *X509Parser* library.

Given the design decisions we previously discussed and the predefined structure of the *CertificateStore* smart contract and the *X509Parser* library, the design of our functions is predefined. However, as we face implementation decisions we discuss them in the description of the implemented functionality below.

To retrieve the relevant attribute for the evaluation under the policy, we have to retrieve first the SSL/TLS certificate at the `checkAttribute()` function, before we can parse its subject field and check whether the SSL/TLS certificates attribute successfully evaluates under the policy.
We implement the `checkAttribute()` function at the *CertificateStore* smart contract as the first touchpoint for the *PDP* library. Given the *PDP* library passed a valid `certID` (i.e. certificate ID), the function retrieves the respective `X509certificate` struct and passes it together with the OID of the to-be-checked attribute type and its desired value to the `checkIfAttributeContained()` function of the *X509Parser* library. At this point we are facing the decision to either use and complement the already implemented `checkNameContained()` function or to implement a new `checkIfAttributeContained()` function.

The `checkNameContained()` function already searches the subject field for the domain name. However, the function checks for wildcard domains, hence does not allow to limit the search to a specific attribute type. Moreover, its search is not only limited to the subject field as it also includes the subject alternative name field of the certificate. Therefore, it is necessary to rewrite the whole function and add multiple checks to cover each case. As this also adds significant amount of code, we do not modify it but implement a new `checkIfAttributeContained()` function.

In the `checkIfAttributeContained()` function we iterate over the subject field using the *ASN1Parser* library. It partitions the DER-encoded data in nodes, according to the nested tree-like structure of the data. Hence, the subject field itself is the root node, that contains nested child nodes. The direct child nodes are the sequence nodes, that each store a object identifier node (i.e. OID attribute type) and a string node (i.e. attribute value). In order to determine whether an attribute type is included in the subject field, the function retrieves for each sequence node the respective object identifier node and checks whether it is equal to the attribute types OID from the policy. Furthermore, it checks whether the string nodes value matches the attribute type value of the policy. We implement the `contain()` function that leverages the *BytesUtils* library[2] for the comparison of byte values.

If a matching attribute is identified by the `checkIfAttributeContained()` function or if the last node was parsed without success, it notifies the PDP_{lib} at the *PDP* library via the PIP_{cst} at the `checkAttribute()` function.

[2]https://github.com/ensdomains/dnssec-oracle/blob/master/contracts/BytesUtils.sol, visited 13/12/2020

5 System Evaluation

The objective of this chapter is to evaluate our system design and prototype implementation. To conduct a sophisticated evaluation and identify an appropriate strategy we select a scientific research methodology, the Framework for Evaluation in Design Science Research (FEDS) from Venable et al. [67]. We first explain and apply the framework in section 5.1. During the application of the framework we identify the *Quick & Simple* evaluation strategy as best match for the evaluation of our system design and prototype implementation. In the following two sections we execute the evaluation strategy in two episodes. Section 5.2 describes the **ex ante evaluation**, where we conduct the system design evaluation prior prototype implementation. The **ex post evaluation**, in which we conduct a requirement and a performance evaluation after the implementation of the prototype, is presented in section 5.3.

5.1 Evaluation Approach

We conduct our evaluation in accordance with the *FEDS framework* for design science research introduced by Venable et al. in [67]. Being compliant with this scientific framework specifically developed for the evaluation of design science research, allows us to increase the reliability and validity of our system evaluation. The framework proposes a four step process, that helps to identify which strategy is the best match for the evaluation of a research project, as well as which properties of the to-be-evaluated system should be evaluated in which setting. In the first step "Explicate the Goals" the framework helps to formulate the goals of the evaluation with respect to rigour, uncertainty & risk reduction, ethics and efficiency. Next in "Choose a Strategy or Strategies for Evaluation", it suggests four different evaluation strategies among which the evaluating researchers can chose from. Once a strategy is selected the framework provides guidance to determine the to-be-evaluated properties of the research in the "Determine the Properties to Evaluate" step. Finally, in the fourth step "Design the Individual Evaluation Episode(s)", researchers are supported in defining the individual evaluation episodes. Below, we briefly explain how we apply the four-step framework and describe the resulting strategy and implications for the evaluation of our blockchain-based ABAC system for smart contracts.

1st Step: Explicate the Goals

For each of the four key goals for evaluation from [67] - rigour, uncertainty & risk reduction, ethics and efficiency - we choose respective objectives for the design of the evaluation of our research. Furthermore, we evaluate how important those goals are for the evaluation of our

system, to pick an appropriate strategy in the next step of the framework.

With regards to *rigour*, it is important for us to evaluate the effectiveness of our system, hence the evaluation component needs to make sure that our ABAC mechanism actually is effectively working in the context of blockchain. Another goal is to evaluate the design of our system and prototype to identify *uncertainty & risk*. More particularly, to identify risks that threaten our systems functionality (i.e. technical risk) or its acceptance among users (i.e. human social risk). However, as we are developing a small-sized first prototype for research purposes and not a large system for productive deployment, both risks are rather limited. Furthermore, as we expand *On-Chain AuthSC*, an already existing and working system that already expands the trust from SSL/TLS certificates to the blockchain, with generally known technology, we believe that the technical risks are low. Yet, the human social risks that our system is not accepted in a decentralized community due to the use of a centralized PKI, is higher and thus should be focus of our evaluation.
An evaluation needs to adhere to *ethical* standards and also reveal ethical concerns, especially if the evaluation method itself or the system might have a negative impact on people or organizations. According to Venable et al. in [67] that is especially important in the context of systems critical to safety. Access control is concerned with protecting resources, hence our system might fall into that category. However, our system is a proof-of-work and not a system that can be used without further evaluation and improvements. Hence, we estimate the ethical implications as limited and do not include an ethical analysis. Yet we obviously adhere to ethic standards when conducting the expert interviews and the performance analysis.
Finally, the *efficiency* of a system is a central goal of an evaluation component [67]. As we are conducting a master's thesis, our time, human and monetary resources are limited. Hence, our goal is to create a simple evaluation component that conducts a high-level evaluation to determine the most relevant risks and the system's general performance, but also creates a foundation for future research.

2nd Step: Choose a Strategy or Strategies for Evaluation

Venable et al. propose four different strategies for evaluation in [67]: Quick & Simple, Human Risk & Effectiveness, Technical Risk & Efficacy and Purely Technical Artefact. The strategy for the evaluation of *Purely Technical Artefact* does not meet our needs, as it is designed to evaluate systems that are very technical and not used or only used by people in the far future. Our system's technical complexity is not extraordinary and use-cases for application by users are set today or in the near-future. Regarding technical risk and human social risk we already identified in step one that both are rather limited, yet human social risks prevail. Hence, we can already eliminate the *Technical Risk & Efficacy* strategy, which focuses on the evaluation of systems where technical risks are high. With regards to human social risks we face the challenge that our ABAC system might not be adopted, as we are using centralized technology in a decentralized community and costs for access request evaluation are high, while execution speed is low. However, as our research project is small and development costs are limited, risks are in general limited. Therefore, we do not conduct a *Human Risk*

& Effectivenes strategy, but a *Quick & Simple* one. It is the strategy of our choice, as it aligns with our efficiency goal. Furthermore, our system design and prototype implementation are not excessively complex and rather simple. Although some human social risks prevail they are still limited as the time and capital invested is small. Quick & Simple is the strategy for the evaluation of such rather small projects with limited risk involvement, where only few episodes are sufficient for evaluation. Nevertheless, although evaluation in the Quick & Simple strategy is limited, we still aim to evaluate our system with regards to the discussed challenges and risks we described above. However, before future deployment in large projects a Human Risk & Effectiveness strategy is recommended.

3rd Step: Determine the Properties to Evaluate

In the third step of the framework, Venable et al. suggest to identify the use-cases, requirements and components of the system design and prototype implementation that shall be the subject of evaluation [67]. During the selection process, the choice of such evaluands shall align with the goals of the evaluation component defined in the first step and the strategy in the second. Hence, we identify the following potential evaluands:

- **Use-Cases**: We include one reference use-case and three exemplary use-cases in our work. To identify *uncertainty & risk*, especially human social risks we can test the use-cases with a limited number of experts (cf. *efficiency* goal) and rate peoples interest in our system.
- **Functional and Non-functional system requirements**: In the system design chapter 3 we defined functional and non-functional-requirements for our system development. Evaluating and improving the requirements with experts allows to increase *rigour* of our evaluation component and thus the effectivity of our system. Furthermore, it allows to identify *uncertainty & risk*: Human social risks, as we can determine which requirements are especially important to potential users and technical risks as we identify design errors before implementation.
- **System Design**: Evaluation of a high-level system design architecture with experts to increase *rigour* of our evaluation. Therefore, it allows us to design the architecture more effectively and identify *uncertainty & risk*, i.e technical and human social risks before prototype implementation.
- **Final Prototype**: The final prototype can be analyzed to determine our system's effectiveness and related *uncertainty & risk* factors. More specifically, prototype execution cost and time are relevant evaluands for our prototype.

4th Step: Design the Individual Evaluation Episode(s)

The final fourth step of the framework is concerned with the number of evaluation episodes, their design and time of execution. Important during this step is to consider resource constraints, prioritize and focus [67].

For the design of our evaluation component, we pick the *Quick & Simple* strategy from [67] and conduct two rounds of evaluation. As we want to identify risks and design errors before implementation, we conduct one episode **ex ante prototype implementation**. Furthermore, to evaluate the efficiency and adherence to requirements we conduct another episode **ex post implementation**.

The first episode of evaluation (**ex ante prototype implementation**) aims to evaluate the early design of our system. Potentially relevant evaluands are the use-cases, requirements and a high-level system design architecture, as they can be created and evaluated ex ante prototype implementation. Furthermore, they already provide insights regarding system effectiveness and potential human and technical risks. For the setting of the first evaluation episode we pick *Experimental*, more specifically controlled experiments as defined by [36]. Among the experimental methods we considered to conduct a survey, as it allows us to reach a large audience with maximum efficiency. However, as the combination of blockchain, access control and SSL/TLS certificates can not easily be explained in the context of a survey, we are concerned that this approach decreases the rigour of our evaluation component. Expert interviews are more rigorous controlled experiments, given a successful identification of a sufficiently large number of experts. Hence, we conduct expert interviews. However, to overcome the challenge of identifying a sufficient number of expert participants, we can leverage the German academic blockchain community. Furthermore, we combine the evaluation of the three different evaluands in one interview to maximize efficiency and the amount of feedback received from each session with an expert.
The second episode of evaluation (**ex post prototype implementation**) aims to evaluate the prototype implementation of our system. Hence, the final prototype is the subject of evaluation. On public blockchains efficiency is especially important, as inefficient applications are much more expensive and slower to use. Hence, the execution time and costs are the evaluands of our choice to evaluate performance and determine the level of human social risk involved. Furthermore, regarding our efficiency goal, both can be analyzed relatively efficiently with limited resources. A design evaluation method that allows such evaluation is a *Dynamic Analysis* as defined by [36], in our case a performance analysis in the Truffle development environment. Beyond performance, we also want to evaluate the alignment of the prototype implementation with the requirements we tested during the first episode. This allows us identify potential human social and technical risk for the future development of our system. To do so, we leverage the *Analytical Optimization* design evaluation method, where optimal properties (i.e. tested requirements) are compared to the actual characteristics of the system [36].

5.2 Episode 1: Ex Ante Implementation Evaluation

During the first evaluation episode we conduct eight expert interviews with researchers from blockchain academia and developers from the blockchain community. The interviews comprised four sections, the first to introduce the topic, concept and background, the second

to evaluate the system design and its high level architecture, the third to discuss the use-cases and the fourth to evaluate the system requirements. After conducting the interviews, we analyze the feedback and introduce some adjustments to our system. In the following we discuss the insights from the interviews, as well as the related implications for our work structured according to our evaluands.

System Design

In the interviews we presented a high-level architecture of our system, comprising of the systems core components: The Registry, its endorsing SSL/TLS certificate and the endorsement, as well as the user account, the respective sub-endorsement and the Application. Furthermore, we specified interactions between the components during an access request. Overall, we received positive feedback for the design of our system. Mr. Knobloch and Mr. Beinke liked the idea of leveraging an established PKI and expanding its trust infrastructure to the blockchain for access control. Especially, as Mr Knobloch mentioned, because the effort to obtain a certificate and hence trust has already been accrued during the certification process of a website. Hence, it would be good to use the trust infrastructure for a different system like our access control mechanism. However, to increase compatibility many interview candidates as Mr. Kurrle, Mr. Buchwald and Mr. Knobloch recommended to expand our blockchain-based ABAC system beyond SSL/TLS certificates and support other types of certificates, as well as Decentralized Identifiers. Nevertheless, regarding the use of a centralized source of trust, Mr. Precht agreed that using a centralized PKI is appropriate, as in order to expand trust to the blockchain a trust anchor would always be needed. However, Mr. Paixão noted, that the centralized approach might face some resistance from the Ethereum community. Hence, the human social risk prevails that some users will not use the system because of its centralized nature.

With respect to the components of our architecture we received some feedback to improve the Registry and the Application. Mr. Paixão highlighted that it would be important to enable individual configuration of the Registry and support functionality with which the owner of the Registry could add additional attributes to sub-endorsements. Based on the feedback we add the configuration mapping, that allows owners to individually configure their Registry. Yet, we do not implement additional attributes at the sub-endorsements for the reasons we previously described in section 3.5.1.
With regards to security, we received the feedback from Mr. Burkhardt, that although Registries are decentralized, large Registries might be subject to (Distributed-Denial-of-Service (DDOS)) attacks. To mitigate this issue we suggest large organizations to create multiple Registries that are endorsed by the similar SSL/TLS certificate. Hence, incentives for such attacks can be decreased. However, to evaluate the security aspect of our system, a detailed analysis is needed. Regarding the design of the Application, Mr. Kurrle recommended to minimize the frequency of access request evaluations for each user account in order to reduce Gas costs. More specifically, he recommended to add a user account for a certain timeframe to a whitelist at the Application once the user was successfully evaluated. During that timeframe

the user can access the service of the Application without the need for another request to be evaluated. This especially is important for future optimization of our system, as it decreases the human social risk that users do not use our system due to performance restrictions.

During the evaluation of the system design we did not identify any significant lack of effectivity or technical risks beyond the discussed security aspect. However, we were able to identify the human social risks we discussed above. Nevertheless, in general our system design and the architecture was well received and no significant problems were identified. Therefore, we develop our prototype based on the presented design, considering the suggestions of our interviewees.

Use-Cases

We presented the reference university use-case to every interviewee, and the exemplary consortia use-case to interviewees that are members of consortia blockchain. In general the feedback for both use-cases was positive and interviewees agreed that the general concept bears potential. Regarding the university use-case Mr. Precht highlighted, that the approach might be interesting to extent a blockchain system that transparently tracks the use of data from multiple research organizations and its members on the blockchain. More specifically, our mechanism could be used to only allow members of a certain organization or organizational unit to access a record keeping smart contract application, that tracks and provides access to research data. With regards to the consortia use-case, Mr. Paixão confirmed the manual work required to approve new members to the blockchain. He agreed that some of this work could be automated with our system, as most approvals are just based on the evaluation of some standard attributes. However, he also highlighted that this might differ for each consortia, depending on their access requirements and the amount of blockchain external information required to review the access request. He continued, that using external data is possible but sometimes challenging, as it requires the use of Oracles - another entity that members need to trust.
In general many interviewees agreed that our access control mechanism could be beneficial for any use-case where a blockchain internal resource requires protection. Examples of suggested application areas include Internet of Things (IoT), business applications, financial services and health care. However, as health care systems require strong security and privacy standards, Mr. Burkhardt highlighted that our system requires additional modification to match theses standards.

While conducting the evaluation of use-cases and discussing other potential application areas of our ABAC mechanism, we noticed interest in our system and its application. Although few challenges prevail, interview candidates confirmed our use-cases, but also proposed new ones in multiple application areas. Hence, with our evaluation we minimize uncertainty and learn that only few human social risks with regards to the general concept of our system exist. Nevertheless, in the next sections, we still have to evaluate whether further human social risks arise from the system design and prototype performance issues.

System Requirements

In the last section of the interview we presented the functional and non-functional system requirements to the candidates. We included their feedback to revise our initial proposal and create our final system requirements specified in section 3.2. The interviewees did not recommend to remove any of our drafted system requirements, however they provided feedback that helped to improve formulation. Moreover, they proposed to add the following four requirements for our system.
Mr. Precht advised us that the user account should be able to check its sub-endorsement at the Registry, before attempting an access request that is subject to a charge in Ether. Additionally, Mr. Knobloch recommended to not only support adding and revocation of sub-endorsements, but also to allow the owner of the Registry to update a sub-endorsement. As both functional requirements include significant functionality without increasing our system's complexity, we add them to our system requirements.
Beyond the two functional requirements, interviewees also recommended to add two non-functional requirements. Mr. Precht highlighted the importance of GDPR compliant programming, hence advised to include a requirement for GDPR compliance. As our system design does not intend to store any personal data on the blockchain that is not already publicly available at SSL/TLS certificates, compliance with GDPR can be achieved. Another requirement that focuses on the optimization of a system was proposed by Mr. Muth. More specifically, he suggested a requirement that demands a specific availability and security level from the ABAC mechanism. Although we believe that both requirements are highly relevant for the acceptance of a system in production, we do not include them in the requirements for our prototype. Particularly because the goal of our prototype development is to first test the feasibility of the concept, before we conduct optimization in future research.

The evaluation helps us to improve and complement the requirements we drafted, such that human social risks are minimized. As we only develop a prototype we do not include requirements suggested by interviewers that demand optimization of performance and security, however we include two suggested requirements that add functionality for users of the system.

5.3 Episode 2: Ex Post Implementation Evaluation

In the second evaluation episode, we focus on the evaluation of our prototype, to determine its performance and how well its implementation meets the requirements from episode one. Hence, we first conduct a performance analysis of our prototype with regards to computation intensiveness and speed in section 5.3.1 before we discuss its alignment with the requirements in section 5.3.2.

Operation	Slow execution (51 Gwei) ETH	USD	MTTC	Fast execution (84 Gwei) ETH	USD	MTTC
Deployment Registry SC	0.0559	28.86	458	0.0920	47.01	27
Deployment Cert. Chain	0.1368	69.90	1797	0.2253	115.06	1366
Creation End.	0.0214	11.05	321	0.0353	18.24	27
Creation Sub-End.	0.0033	1.71	321	0.0055	2.84	27
Revocation Sub-End.	0.0009	0.47	321	0.0016	0.83	27
Update Config	0.0007	0.36	321	0.0012	0.61	27

Table 5.1: Cost and speed of execution for operations at the Registry smart contract

5.3.1 Performance Analysis

In Ethereum the amount of computation required to execute a transaction determines the cost of execution. Furthermore, as the creator of the transaction can specify the price he or she is willing to pay for each computational step (i.e. Gas price), the time until execution of a transaction, i.e. speed is depending on the price. In our performance analysis we want to consider this trade-off between execution cots and execution time. Hence, we pick a Gas price for evaluation that ensures cheap execution at a slow speed and another one for a fast, but expensive execution. We display the price for the execution of a transaction in Ether and USD, as well as its execution time in seconds.

We develop our prototype in Solidity and test it in the Truffle development environment. To conduct a structured analysis, we create test cases that describe likely scenarios from the perspective of the owner of a *Registry* smart contract, owner of an *Application* smart contract and sub-endorsed user account. For each transaction submitted during the test case we retrieve the required computational steps in Gas from Truffle. Together with the current Gas prices for a cheap or a fast execution from "ETH Gas Station"[1], we are able to determine the cost of execution. Furthermore, as "ETH Gas Station" also provides an estimate of the Mean Time To Confirm (MTTC) for each execution, we use the value to determine how fast functionality in our system can be executed. For the following analysis we assume the following market data as of 26/11/2020: A Gas price of 51 Gwei (cheap) and 84 Gwei (fast), as well as an Ether to USD conversation ratio of 510.72 USD for 1 ETH[2].

Registry Smart Contract Deployment and Management

This section describes scenarios where a new *Registry* smart contract is deployed and endorsed, a sub-endorsement for a user account is created and revoked and one where the configuration is updated. The process and the Gas used for each step is depicted in Figure 5.1. Furthermore, the respective Gas costs in Ether and USD are listed in Table 5.1. As Groschupp created a

[1] https://ethgasstation.info/index.php, visited 26/11/2020
[2] https://coinmarketcap.com/converter/eth/usd/, visited 26/11/2020

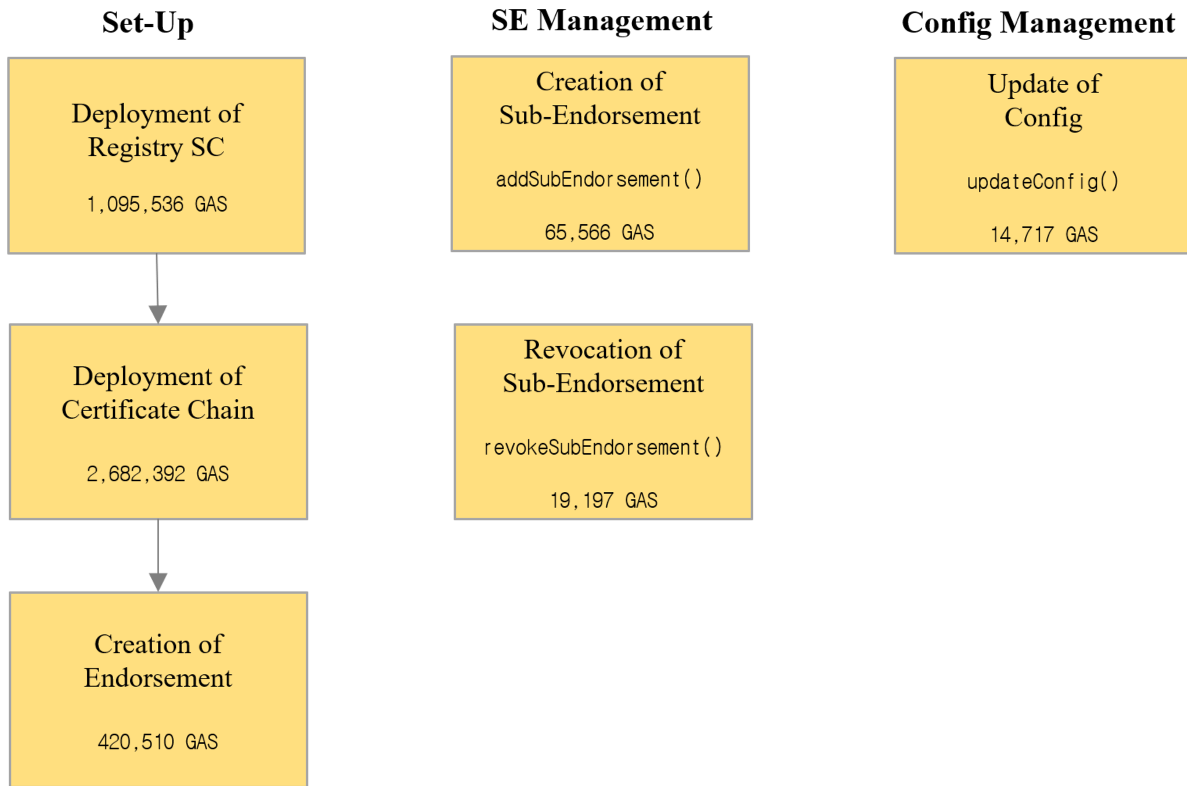


Figure 5.1: Scenarios at the Registry smart contract

sophisticated analysis of the required Gas for the "Deployment of a Certificate Chain" and the "Creation of Endorsement" operations, we refer to their median values from [33]. All the other values are based on our prototype in Truffle.

An organization that wants to allow its members to use the ABAC mechanism on the blockchain, first needs to set-up the *Registry* smart contract and its endorsement. Hence, computational costs accumulate for the deployment of the *Registry* smart contract (1,095,536 Gas), uploading the endorsing SSL/TLS certificate chain (2,682,392 Gas) and creating an endorsement for the *Registry* smart contract (420,510 Gas). In total, this leads to 4,198,438 Gas and costs of 28.86$ + 69.90$ + 11.05$ = 109.81$ given a slow execution. We conclude that the deployment of the *Registry* smart contract requires a relatively large investment for a private person. However, in the use-cases we introduced the *Registry* smart contract is usually owned by organizations with many members and larger funding. Furthermore, as these are deployment costs, they only have to be paid once and execution time is not critical.
Regarding the sub-endorsement management, much less computational steps are required for creating (65,566 Gas) and especially revoking (19,197 Gas) a sub-endorsement. Once again we consider slow execution costs, as a confirmation time of 321 seconds seems reasonable in most cases. Hence, the costs for creating (1.71$) and for revoking (0.47$) a sub-endorsement are much lower compared to the set-up. However, the costs accumulate, if an organization

regularly adds and removes sub-endorsements. Hence, it depends on the user behaviour and the use-case, whether the system is profitable. Finally, we also determine the computational efforts for updating the configuration (14,717 Gas) of the *Registry* smart contract. Costs are 0.36$ for a MTTC of 321 seconds and 0.61$ for a MTTC of 27 seconds. Even if changes to the configuration are time critical, e.g. in order to prohibit every access control request for any sub-endorsed user account when malicious behaviour is detected, speed and costs are still reasonable.

Application Smart Contract Deployment and Management

In this section we evaluate the speed of and the cost related to the deployment and management of the *Application* smart contract. The scenarios are depicted in Figure 5.2 and the related costs in Table 5.2. In order to only measure the performance of our ABAC mechanism, we do not include any to-be-protected functionality. Hence, the *Application* smart contract only includes the required functionality of our system.

At first in the set-up, the *Application* smart contract has to be deployed and the owner needs to specify which SSL/TLS Root certificates he or she trusts. Thus, the set-up requires 1,578,342 Gas for deployment and 96,598 Gas to add a trusted SSL/TLS Root certificate to the *Application* smart contract's Root Store. For the set-up we assume a slow execution with a MTTC of 779 seconds. Hence, cost accumulate to 40.98$ + 2.53$ = 43.51$. However, the MTTC can also be as low as 54 seconds, given the willingness to pay 71.8$. As in the previous case, the set-up of the system requires up-front investment. Nevertheless, as costs for deployment only have to be paid once, and as it is sufficient to only add 13 Root certificates to the Root Store in order to trust 98% of the Domain certificates [33], we conclude that the costs are acceptable and neglectable in the long-run. In a second test scenario, we update the policy of the *Application* smart contract. More specifically, we change the attribute type (30,012 Gas) and the attribute value (30,068 Gas). That results in costs between 0.77$ and 1.29$ for each operation depending on the desired MTTC (321 seconds or 27 seconds). Finally, in the third scenario, we also determine the cost associated to changing the address of the *PDP* library, the *EndorsementDatabase* smart contract or the *CertificateStore* smart contract. Each operation requires about the same amount of Gas, hence costs between 0.72$ and 1.24$ depending on the MTTC. As we assume that addresses and policies are not frequently updated, costs are relatively low. Therefore, we conclude that cost and speed in both scenarios are reasonable.

Access Request Evaluation

The goal of this last scenario is to evaluate the performance of the core functionality of our ABAC system for smart contracts: The evaluation of an access request from a sub-endorsed user account at an *Application* smart contract. In order to do so, we endorse a *Registry* smart contract with a SSL/TLS certificate and create a sub-endorsement for our user account. We furthermore use the *Application* smart contract we tested in the previous section. It supports every functionality of our system, but does not include any to-be-protected functionality, as

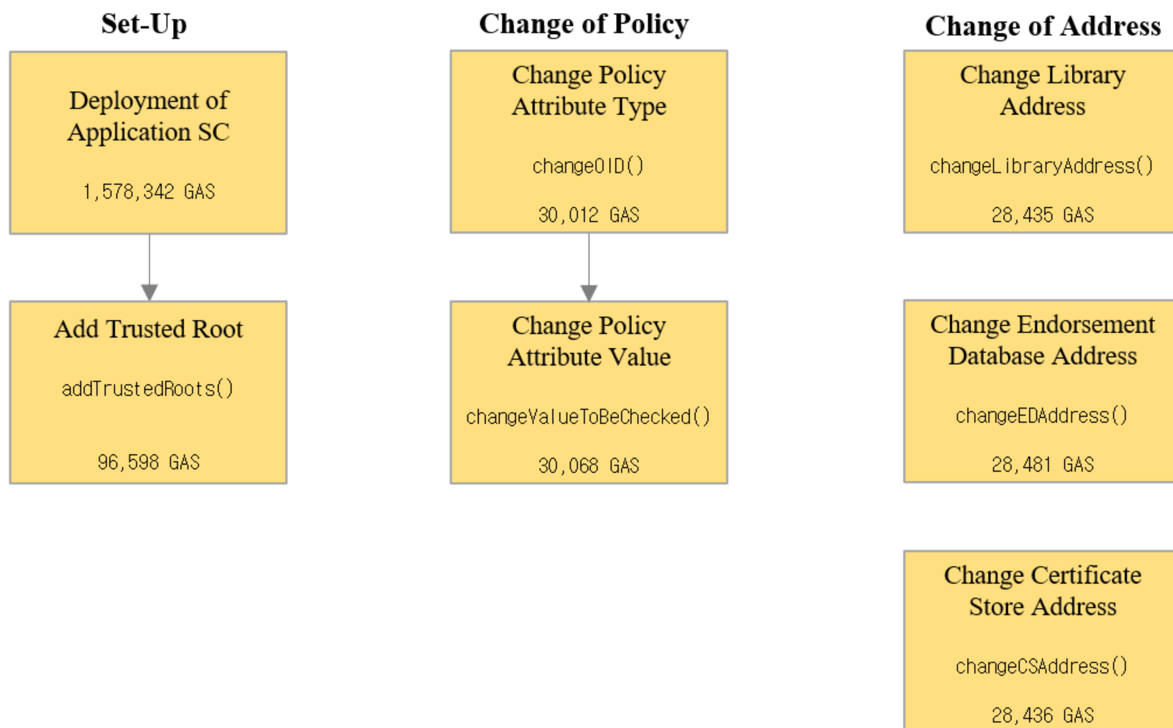


Figure 5.2: Scenarios at the Application smart contract

we only want to determine the performance of the ABAC system.
During the evaluation we learn that the required computational power highly depends on the selected policy, more specifically whether a policy is tested or not. Hence, in Table 5.3, we display the cost for an access request check with regards to the selected attribute. However, we exclude the MTTC as it is similar for each observation: 321 seconds for the slow execution and 27 seconds for the fast one.

The access request check comprises three core components, the check of the sub-endorsement, the check of the *Registry* smart contract's endorsement and the retrieval and evaluation of the SSL/TLS certificate's attributes under a policy. If we conduct a check of all the three components, hence use the full functionality of our ABAC system, the required Gas varies between 102,000 and 129,358. Thus, costs for a slow execution vary between 2.68$ and 3.40$ and between 4.43$ and 5.52$ for a fast one. Although the process of the access request check is similar, no matter which attribute type is selected, the computational intensity differs depending on the attribute type. We conclude that this is due to the fact that we iterate from start to end through the DER-encoded subject field of the SSL/TLS certificate to search an attribute type and its respective value. Therefore, we have to search longer for attribute types that are stored at the end of the DER-encoded data string. As *Country* is stored in the front, only 102,000 Gas is needed. The attribute types *Business Unit* and *Domain* are stored in the very end, thus the required Gas to retrieve both is significantly higher. Consequently,

Operation	Slow execution (51 Gwei) ETH	USD	MTTC	Fast execution (84 Gwei) ETH	USD	MTTC
Deployment Application SC	0.0805	40.98	458	0.1326	67.62	27
Add Trusted Root	0.0049	2.53	321	0.0081	4.18	27
CP Attribute Type	0.0015	0.77	321	0.0025	1.29	27
CP Attribute Value	0.0015	0.77	321	0.0025	1.29	27
Change Library Address	0.0014	0.72	321	0.0024	1.24	27
Change EDB Address	0.0014	0.72	321	0.0024	1.24	27
Change CST Address	0.0014	0.72	321	0.0024	1.24	27

Table 5.2: Cost and speed of execution for operations at the Application smart contract

we claim that the computational steps, i.e. Gas required to retrieve the respective attribute depends on the attribute types position in the subject field.
To even better understand the magnitude of the computation intensity required for the attribute retrieval and policy evaluation, we determine the Gas needed just for this type of check. Hence, we compute the magnitude by subtracting the Gas required to only check the sub-endorsement and the endorsement from the Gas required for a check of all three components. As the Gas needed for the complete execution varies depending on the attribute types, we use the median of the Gas consumption among all attribute types. Given a median consumption of 115,633 Gas for the complete execution and a consumption of 56,151 Gas for the execution of the mechanism, that checks the sub-endorsement and the endorsement only, the difference amounts to 59,482 Gas. Hence, the attribute retrieval and policy evaluation accounts for more than 50% of the Gas required.

We conclude that our attribute retrieval and policy evaluation mechanism still requires major performance improvements, as the search algorithm executed for each user access request evaluation is computation intensive. Therefore, a complete check of an access request currently is twice as expensive as without attribute retrieval. This is especially problematic, as in most use-cases, access requests are submitted by users very frequently. However, a potential solution that should decrease the computation intensity was already proposed by us in section 4.2.1 and should be implemented in the future. The idea is to retrieve and decode the data stored in the subject field only once when a SSL/TLS certificate is added to the *CertificateStore* smart contract. The attribute type and value can then be stored in variables that can easily be accessed for the access request check. By doing so, it is no longer necessary to always iterate through the subject field to retrieve the attributes. However, as we already stated previously, we do not implement this mechanism as its implementation is very complex and goes beyond the scope of this work.

OID	Attribute Type	Gas	Slow execution (51 Gwei)		Fast execution (84 Gwei)	
			ETH	USD	ETH	USD
2.5.4.3	Domain	126,943	0.0065	3.35	0.0107	5.51
2.5.4.6	Country	102,000	0.0052	2.68	0.0086	4.43
2.5.4.7	State	112,625	0.0057	2.94	0.0095	4.90
2.5.4.8	Locality	107,998	0.0055	2.84	0.0091	4.69
2.5.4.10	Business Name	118,640	0.0061	3.15	0.0100	5.15
2.5.4.11	Business Unit	129,358	0.0066	3.40	0.0109	5.62
NULL	No attribute	56,151	0.0029	1.50	0.0047	2.43

Table 5.3: Cost and speed of execution for the evaluation of an access request

Summary and Implications

During our analysis we learned, that the up-front deployment costs and the MTTC for the deployment of our system are within a reasonable range. The set-up for the *Registry* smart contract costs 109.81$ given a slow execution with a MTTC of about 43 minutes (2576 seconds). For the deployer of the *Application* smart contract costs amount to 43.51$ at a MTTC of about 13 minutes (779 seconds). In general, we believe that deployment costs are neglectable in the long-run, but operational costs and speed of execution decide whether our system works profitable in a specific context. Hence, it is important to identify operations which are time-sensitive and/or frequently executed in a certain use-case and thus accumulate cost over time. In our system, there are three such operations: Adding and revoking a sub-endorsement, as well as the check of an access request.

With regards to managing sub-endorsements, creating a sub-endorsement uses 65,566 Gas (1.71$) and revoking 19,197 Gas (0.47$). While both operations are not especially expensive in terms of required Gas, costs may add up if the owner of the *Registry* smart contract has to frequently add or remove users. As the frequency depends on the use-case, it is hard to determine whether costs are reasonable in general. However, we can conclude that in use-cases, where the user base of the *Registry* smart contract is relatively consistent, costs for both operations are moderate. Nevertheless, in cases where user fluctuation is high, our system and probably other public blockchain-based system, might not be the best fit. Regarding the speed of both operations, currently user accounts have to wait between 27 and 321 seconds, before their sub-endorsement is added to the *Registry* smart contract. Depending on the use-case the waiting time might constitute a human social risks, especially as users are used to fast registration online. Yet, blockchain users are used to a delay and the registration at an organizations business system usually also is not instantaneous. Therefore, and as the waiting time still is reasonable, we perceive the risk as being rather low.

The computational costs of the access request check vary between 102,000 (2,68$) and 129,358 Gas (3,40$) depending on the to-be-checked attribute. The MTTC varies between 27 and

321 seconds. In the respective section of this chapter we already concluded that costs are relatively high for an operation that is very frequently used. Hence, human social risks that our system may not be adopted by users still prevail. Therefore, in future work it is significant to improve the performance. For example, by improving the mechanism that retrieves the attributes from the SSL/TLS certificate or by implementing a solution where granted access requests are valid for multiple interactions during a specific time frame. With regards to the speed of execution, we perceive the human social risk to reject our system as low. The user of the blockchain system has to wait anyway for the execution of the protected functionality, hence already expects a delay. Furthermore, as the access request check is executed together with the protected functionality, the delay is not significantly increased.

We conclude that it is challenging to evaluate, whether the cost and speed of the system is reasonable, as the long-term profitability depends on the use-case, especially on how often it requires to add or revoke sub-endorsements and how often an access request needs to be checked. Furthermore, it also depends on the blockchain system. With regards to performance and costs, our system is constraint by the specifications of the public Ethereum blockchain. Thus, a system based on a private blockchain or external ABAC systems is cheaper and faster in most cases, but also lacks the benefits of a public blockchain. Nevertheless, independent from the type of blockchain our implementation still is a first prototype. Hence, the implementation is not optimal with regards to performance and should be improved in the future.

5.3.2 Requirement Analysis

This analysis evaluates whether our prototype implementation meets the system requirements we defined in section 3.2 and refined during the interviews. An overview of the requirements is depicted in Table 5.4. In the following we briefly go through the functional and non-functional requirements and discuss how and to which degree compliance is achieved.

Functional Requirements

At the *Registry* smart contract, our prototype supports functionality to manage sub-endorsements. By invoking `addSubEndorsement()`, the owner can add a sub-endorsement for a user account to the *Registry* smart contract (FR1). The functions `updateSubEndorsement()` and `revokeSubEndorsement()` enable the update (FR3) and revocation (FR2) of a respective sub-endorsement. As the user might be interested in the current status of its sub-endorsement, he or she can check the sub-endorsement (FR4) at the sub-endorsing *Registry* smart contract by invoking the `checkSubEndorsement()` function. The sub-endorsement can be leveraged to access a to-be-protected function of an *Application* smart contract, that supports blockchain-based ABAC for smart contracts. The access request is submitted indirectly by the user account (FR5), by invoking and passing the address of its sub-endorsing *Registry* smart contract to the protected function it desires to access. The authentication of the user account is inherent in Ethereum. Hence, during the function call the user account is authenticated (FR6),

#	Requirement	Status
FR1	Add sub-endorsement at Registry	✓
FR2	Revoke sub-endorsement at Registry	✓
FR3	Update sub-endorsement at Registry	✓
FR4	User account may check status of sub-endorsement at Registry	✓
FR5	Request access to Application	✓
FR6	Authenticate user account at Application	✓
FR7	Check authorization of user account at Application	✓
NFR1	Leverage attributes of SSL/TLS certificates	✓
NFR2	Use On-Chain AuthSC	✓
NFR3	On-chain access control decisions	✓
NFR4	Decentralized sub-endorsement allocation	✓
NFR5	Access control without a direct trust relationship with the Registry	✓
NFR6	Access control without pre-provisioning of the subject at the Application	✓
NFR7	Minimal costs of user management, authentication and authorization	X

Table 5.4: Compliance of the prototype with regards to our requirements

as the signature of its invoking transaction is validated by the EVM. Finally, the prototype also supports the evaluation of the user account's authorization (FR7). More specifically, it checks the sub-endorsement (FR7.1) of the user account, the endorsement of the *Registry* smart contract (FR7.2) and whether the attributes of the endorsing SSL/TLS certificate successfully evaluate under the *Application* smart contract's policy (FR7.3). As for the submission of the access request and the authentication of the user account, the authorization evaluation is indirectly initiated after the user account invokes the protected function at the *Application* smart contract.

Non-Functional Requirements

The goal of our research is to leverage SSL/TLS certificates and a SSL/TLS-based identity assertion and verification system to enable authentication and access control of user accounts at smart contracts. Hence, in our prototype we use *On-Chain AuthSC* (NFR2) to expand trust and create a link to the attributes of SSL/TLS certificates. More specifically, by linking a user account with a sub-endorsement and an endorsement to a SSL/TLS certificate, we allow user accounts to leverage the trust and attributes of SSL/TLS certificate (NFR1) for access control at an *Application* smart contract. In order for our prototype to conduct access control, it needs to access attribute data from the endorsing SSL/TLS certificates. However, as all the relevant data in our prototype, the certificate chain is stored on-chain at the *CertificateStore* smart contract of *On-Chain AuthSC*. Therefore, our prototype does not need to rely on blockchain external data and can execute access control completely on-chain (NFR3).
Decentralization is a key characteristic of blockchain systems. Hence, also in our prototype

we aim to increase decentralization. In the prototype any owner of a SSL/TLS certificate can create a *Registry* smart contract and issue sub-endorsements for user accounts. Thus, sub-endorsements are not allocated at a central entity, they can be distributed among multiple *Registry* smart contracts (NFR4). Moreover, in our prototype a user account that wants to access a function at an *Application* smart contract only has to provide the address of its sub-endorsing *Registry* smart contract. No previous trust relationship between the *Registry* smart contract and the *Application* smart contract is required (NFR5), as well as no pre-provisioning of the user at the *Application* smart contract (NFR6). This is due to the fact that all required trust is endowed by the SSL/TLS certificates and can be checked with the PDP library. Finally, NFR7 is to minimize the cost of user management, authentication and authorization. While the implementation of the attribute retrieval is not optimal, the costs of other operations are reasonable. However, as the performance of this operation is highly significant NFR7 is the only requirement that is not satisfied.

6 Related Work

The goal of this chapter is to describe research and technology that is related to our work, as well as to structure and classify the access control research environment. Moreover, we aim to identify and specify research gaps, which can be filled by the contributions of our work.
All in all, there is no other research which attempts the design and the development of a prototype for an on-chain mechanism for real-world entities at smart contracts, based on a SSL/TLS identity assertion and verification system. However, there is research that elaborates access control and blockchain in many different settings with varying focus. Most research or applications leverage blockchain as underlying technology for authentication and access control. Only few research focuses on internal access control for functions of smart contracts, hence those resources to-be-protected are usually located off-chain. The applied access control mechanisms are usually attribute-based, only some are role-based and discretionary. However, there is a lack of trust in attributes or roles assigned to accounts in public blockchains, as the credibility of the attribute or role issuer usually can not be validated. A solution to overcome this trust issue is proposed by research in self-sovereign identity, where the requester's attributes are endowed with trust by a decentralized Web-of-Trust. However, research still is limited, mainly focused on Identity Management in general and still has not identified a way to successfully build a sufficiently large and trusted decentralized Web-of-Trust. Consequently, there is still room for other solutions as ours, which suggest new approaches to fill the research gap.

In the following we provide examples and analyze the related research to underpin our statements in the summary above. To conduct a clear and comprehensive analysis, we classify the related work according to its focus: Blockchain-based authentication in section 6.1 and blockchain-based access control in section 6.2. However, sometimes the line between systems that conduct only authentication and systems that allow authentication and access control is blurry. Especially since some authentication systems already implement important components of access control, but not the complete system. In such cases, we assign the work to the category where we think it might match best. Furthermore, our goal is not to create a sophisticated analysis of each and every blockchain-based authentication and access control system. Such research would need the scope of a master's thesis for itself. Hence, we mainly focus on Ethereum-based systems and ledger-independent concepts.

6.1 Authentication

Authentication is inherent in blockchain systems, as accounts are represented by public private key pairs. Based on public key cryptography accounts can sign data (e.g. messages) with their private key, which can be validated using their public key. There are multiple unreviewed proposals of the blockchain community, that propose to leverage the blockchains cryptographic characteristic to enable improved and more user-friendly authentication at blockchain external resources. For instance, for authentication at web applications in [49] Miller suggests a system which uses a wallet browser plug-in (e.g. MetaMask) that stores the users account keys and is able to interact with the web-app with Web3 injections. To authenticate at a web-app, the user needs to submit the account address as username and sign an arbitrary message from the web-app with the accounts private key via the wallet plug-in. Given a successful validation of the signature with the user's public key, the web-app creates an access token and sends it to the user. Multiple sources propose similar solutions: [11], [54], [56], [60], [34]. Some of them differ slightly, as from Shumikin et al. in [60] and Gruener et al. in [34] where the developers use a wallet app that scans QR codes which include a to-be-signed message and a callback to communicate with the web-app. However, all of them use public and private keys to authenticate an account in a challenge and response protocol. Authentication at internal blockchain resources, as smart contracts is automatically conducted when a user account invokes a function via a transaction (cf. section 2.2.5). Authereum [5] and UniLogin [65] aim to improve the user-experience of the authentication process at blockchain internal resources as smart contracts, i.e. DApps. Currently users often can already access a DApp via a web-app, nevertheless are still required to use a wallet browser plug-in as MetaMask to log-in and interact with the DApp. To decrease barriers of entry, for users which are unfamiliar with blockchain technology, Authereum and UniLogin promise a common authentication with username and password.

The related work either aims to leverage blockchain technology to improve authentication at external resources or to increase the user-experience of authentication at internal resources, i.e smart contracts. As we focus on neither of the two, no similar research which enables on-chain authentication and authorization at smart contracts based on SSL/TLS certificates was identified.

6.2 Blockchain Access Control Systems

As authentication is inherent in blockchain systems, most access control systems leverage the characteristic of the blockchain, but differ in terms of mechanism type. The majority of the proposed blockchain access control systems are attribute-based. Nevertheless, there are also sophisticated systems which are role-based and one system which is based on discretionary access control. As we develop an ABAC system for smart contracts, we first discuss the respective related work in depth in section 6.2.1, before we provide a brief summary of related discretionary and role-based blockchain access control systems in section 6.2.2.

6.2.1 Attribute-Based Access Control on Blockchain

ABAC is among the most common access control mechanisms that are transferred to and applied on the blockchain. Multiple private blockchains as Hyperledger offer implementations of ABAC as in [43] and [20]. However, as their architecture significantly differs from Ethereum, we mainly focus on Ethereum-based systems and ledger-independent concepts.

Maesa et al. develop a blockchain-based ABAC based on the XACML framework in [23], [22] and [21]. In general they elaborate how to exploit blockchain technology to develop a decentralized ABAC. They discuss which components of an ABAC should be located and executed on the blockchain and also propose different system architectures with different level of blockchain involvement. In [21] they propose a system that can be easily integrated in already existing ABAC systems. It only stores the Attribute Managers (AM), the part of the system which stores and manages the attributes, on the blockchain. Hence, the use of blockchain technology is very limited. Furthermore, the access control mechanism is intended to protect blockchain external resources. In [23] Mesa et al. describe an ABAC, where the AM, as well as the PIP and the PDP reside on a blockchain. Each policy is stored as smart policy in a smart contract, which resembles the PIP and the PDP. As also the attributes are stored on the blockchain, the evaluation of a policy in context of an access control request is completely performed on-chain. A third paper by Mesa et al. [22] proposes an ABAC, where the protected resource is a smart contract and/or its respective functions and not a blockchain external resource as in the two papers we discussed earlier. Furthermore, the PEP and parts of the PAP are now stored within the to-be-protected smart contract. Hence, the user that requests access to the resource, can submit the request directly to the resource from the users blockchain account. The smart contract ABAC in [22] shares strong commonalities with our system. It is an on-chain attribute-based system, that protects access to blockchain internal resources. However, a key difference is that the attributes in [22] are only endowed with trust from AMs. Opposed to the certificates, that store attributes in our system, AMs are not endorsed by a globally trusted PKI. Furthermore, AMs are also not endorsed by any other source of trust as a decentralized PKI.

In [3] Almakhour et al. describe a token-based system that manages access to Ethereum service smart contracts among members of a federation. The members of the federation define the access control policies and attributes of services. Then each member loads all the users of the federation on a user smart contract. To get access to services, the users first need to obtain their token from the user smart contract on the blockchain. The contract looks up the user and its respective attributes from a pre-loaded list and returns a unique token, with which the user can request access to a service smart contract. Given a successful verification of the token and its attributes, the user is granted access. As Mesa et al. in [22], Almakhour et al. in [3] shares the same strong commonalities, but also lacks trust endowment of the central attribute providing entities, i.e user smart contract. Hence, both systems might be sufficiently endowed with trust in a private blockchain, but not on public blockchains where association of accounts to real-world identities is cumbersome.

Beyond the two papers of Mesa et al. [23][21], there is other work where the to-be-protected resources are stored externally and only (parts of) the ABAC mechanism resides on the blockchain: [45], [35], [61], [12]. Siris et al. in [61] and Buccafurri et al. in [12] apply ABAC to manage access to IoT devices. In [45] Jemel et al. proposes a distributed ledger for ABAC of blockchain external data. The blockchain manages the policies to access the external data and evaluates the users access requests. To add a new policy, the resource owner creates a transaction that contains the policy, a tree structure that stores the attributes required to be granted with access. The nodes of the blockchain validate the transaction and add it to a block. If a users wants to access an external resource, a transaction with attributes needs to be created and submitted to the blockchain. Given successful verification of the user's attributes and conditional attributes, the user receives an access key of the respective resource. Another blockchain-based ABAC that allows a data owner to share data with a user is described in [35] by Guo et al. In order to get access to the encrypted data, the user needs to acquire an access key from the data owner with attribute tokens. The user obtains these tokens from pre-verified Ethereum nodes called Attribute Authorities. These nodes manage and validate user attributes, and submit tokens to authorized requesters. If a user obtained sufficient tokens from respective Attribute Authorities, he or she may request the access key from the data owner to decrypt the external data. Although in [35] and [45] user attributes are critical to protect data, none of the two papers elaborate attribute trust endowment.

Self-sovereign blockchain-based identity systems are not particularly focused on access control systems. Nevertheless, some indirectly support limited ABAC, as Uport [66] or Sovrin [63]. For example, Uport, an identity management system on the Ethereum blockchain, allows users to obtain a claim from another identity within the ecosystem. The claim is stored in the users DID document and can be used to authenticate and be granted access to services offered by other identities. A Proof-of-Concept of such a solution was conducted in Zug, Switzerland [47]. Users can create an identity with the uPort App and assign the passport number and date of birth to their identity. After in-person verification at the city clerk, the city signs the claim. From then on, citizens are authorized to use city services. However, current implementations mostly focus on authentication and blockchain external resources, no self-sovereign blockchain-based work we identified explicitly implements ABAC. Only Gruener et al. in [34], suggest future work to augment the current authentication system based on DIDs with an authorization component that leverages the attributes in the DID document.

In general we conclude that blockchain-based decentralized identity provider, based on self-sovereign identity offer a strong foundation for the implementation of blockchain-based ABAC. In particular, because attributes are already assigned to identities and theoretically already are endowed with trust from a decentralized Web-of-Trust. However, we did not identify any sufficiently large Web-of-Trust for turst endowment and furthermore no implementation that explicitly implements a blockchain-based ABAC mechanism for smart contracts.

6.2.2 Other Access Control on Blockchain

Blockchain-based RBAC is elaborated in research publications, but also is already applied in explicit, widely-used implementations for different types of blockchains.
A RBAC implementation, that leverages X.509 certificates to authenticate and authorize users, is a component of the Hyperledger Fabric blockchain [42]. As previously stated, in general we focus on Ethereum-based or ledger-independent concepts, but as the Hyperledger RBAC system leverages x.509 certificates, it is very similar to our ABAC implementation and related to our work. In Fabric, to interact with restricted services in the network, users need to be assigned to respective roles by a Membership Service Provider (MSP). Each MSP defines a Fabric member organization and maintains a record of its approved members. To be authenticated and authorized, a new user has to present a valid X.509 certificate from the organization, that is linked to a Root certificate trusted by the MSP. Once validated, the user is assigned to a role, which is specified in an attribute of the certificate. As in our system design, X.509 certificates are used to endow entities on the blockchain with trust and the access control id focused on blockchain internal resources. Nevertheless, the systems is different as it is only geared towards a private blockchain and access control is based on roles and not on the specific attributes in the SSL/TLS certificate.
OpenZeppelin provides a contract module for RBAC on the Ethereum blockchain [51]. Developers who want to implement RBAC in their smart contract can inherit from Open-Zeppelin's *AccessControl* smart contract and use pre-defined functionality. It allows to define different types of roles and provides utility functions to manage admins, users and roles. Thus, functions of a smart contract can be protected, such that they can only be revoked by users assigned to a specific role.
Besides the two RBAC systems, that are geared towards access control of blockchain internal resources, there is the RBAC-SC system from Cruz et al. [19]. First specified as RBAC for Bitcoin in [18], later in 2018 as a RBAC for Ethereum in [19]. RBAC-SC on Ethereum leverages smart contracts on the blockchain to manage access to blockchain external resources. Cruz et al. describe a use-case, where a student is registered at a smart contract of the university and assigned to a "student" role in a role contract. Based on the endorsement the student is able to successfully request access to an external service provider (e.g. a university cafeteria), given the provider successfully validates the students role on the blockchain. However, Cruz et al. do not discuss in their paper how trust in the service provider is established.

There is one paper from Zheng et al. where Discretionary Access Control [69] is applied. More specifically, Zhang et al. use smart contracts to store links to access control lists for IoT devices. Requests to use an IoT device are evaluated based on the subject (user) object (IoT device) relationship in the respective ACL and based on previous misbehaviour of the user with the respective IoT device. Another IoT related paper by Ouaddah et al. [52] designs a framework called "FairAccess", that specifies a generic system design for blockchain-based IoT access control, but leaves the implementation of a specific access control mechanism to other researchers. Moreover, there are two token-based approaches to blockchain-based authorization of identities in a federated systems. One from Fotiou et al. that ports parts of

OAuth 2.0 Authorization protocol to the blockchain [29], and another from Bendiab et al. that enables federated access for users of multiple Cloud Service Provider [9].

7 Discussion

In this chapter we conclude this research. We discuss our contribution, recapitulate our research questions and illustrate limitations of our system in section 7.1. Finally, we provide suggestions for future work in section 7.2.

7.1 Conclusion

In this research we explore how authentication and access control at smart contracts can be achieved for real-world entities by leveraging trust and attributes from SSL/TLS certificates. We conduct a survey of access control models and select Attribute-Based Access Control as fundamental concept for our work. Moreover, we choose *On-Chain AuthSC*, a SSL/TLS-based identity assertion and verification system, to expand trust from the SSL/TLS certificate public key infrastructure to the Ethereum blockchain. We complement *On-Chain AuthSC* with an endorsement framework, that allows entities with a SSL/TLS certificate to create and manage sub-endorsements for accounts of real-world entities on the blockchain, such that they can leverage trust and attributes from the SSL/TLS certificate for authentication and access control at smart contracts. Furthermore, we transfer the concept of ABAC to the blockchain by creating an ABAC system, that allows owners of smart contracts to evaluate access requests of real-world entities considering their sub-endorsement, i.e association with a SSL/TLS certificate and its attributes. To test the system design we implement a prototype, including a library and two reference smart contracts for fast and easy adoption of our system, in Solidity for the Ethereum blockchain.

Our research is aligned by the three overarching research questions we introduced in section 1.2. During the course of this work we provide answers for each of the questions, nevertheless we summarize in this chapter to conclude our contribution.

RQ1: Which are the major access control practices and technologies?

We conduct a survey of the major access control models that are predominant in the literature in section 3.4 and evaluate how well each model matches to the goals our research. Thereby we identify four major mechanisms: Mandatory Access Control, Discretionary Access Control, Role-Based Access Control and Attribute-Based Access Control. Among those we chose ABAC as fundamental concept for our system, as it meets best the requirements of our work. Furthermore, we also review related literature and unstructured work in chapter 6, hence identify authentication systems, as well as few attribute and role-based access control

mechanisms for the blockchain. However, there is no system that creates trust by leveraging the SSL/TLS certificate PKI.

RQ2: How can a SSL/TLS-based identity assertion and verification system contribute trust to authentication and access control on the blockchain?

We elaborate the contribution of the SSL/TLS certificate PKI, as well as *On-Chain AuthSC*, and design an endorsement framework in section 3.3.2. SSL/TLS certificates act as trust provider in our system. They are endowed with trust from SSL/TLS Root certificates, which are issued by Certificate Authorities. The two key properties that create an increased level of trust, are the trust of the society in and the wide acceptance of the endorsing Certificate Authorities and the practically unforgeable link that distributes trust among SSL/TLS certificates. Moreover, especially important for our system is that SSL/TLS certificates comprise confirmed and validated attributes that describe the certified entity. However, we would not be able to leverage the SSL/TLS certificate PKI without *On-Cahin AuthSC*, a SSL/TLS-based identity assertion and verification system. It contributes to our system, as it allows to securely expand the required trust to the blockchain and provides an on-chain Certificate Database from which we can retrieve the attributes of SSL/TLS certificates. Yet, *On-Chain AuthSC* does not support endorsements for real-world entities that do not own a SSL/TLS certificate. Hence, we expand the endorsement framework by sub-endorsements as described in section 3.3.2 to allow the participation of other real-world entities. Furthermore, *On-Chain AuthSC* is a standalone system, consequently some changes to its Certificate and Endorsement Database are required to enable the retrieval of attributes and the evaluation of endorsements (cf. section 3.5.2).

RQ3: How can we achieve on-chain authentication and access control of real-world identities at smart contracts considering the constraints of blockchain?

In order to conduct authentication and access control based on attributes (i.e ABAC), trusted attributes need to be associated to accounts of real-world entities. However, as no sophisticated centralized or decentralized trust infrastructure resides on the public Ethereum blockchain, we expand trust from the SSL/TLS certificate PKI with the help of *On-Chain AuthSC*. We furthermore design (cf. section 3.3.2) and implement (cf. section 4.1) a sub-endorsement framework to expand trust and associate attributes to accounts of real-world entities that do not own a SSL/TLS certificate. Together with *On-Chain AuthSC* and the sub-endorsement framework we create an endorsement framework that provides the trust needed for our ABAC mechanism. The ABAC mechanism allows an Application to authenticate and control access of real-world entities, by checking their endorsement, sub-endorsement and the attributes of the endorsing SSL/TLS certificate. Hence, we achieve on-chain authentication and access control of real-world identities at smart contracts by combining an endorsement framework, that comprises *On-Chain AuthSC* and the sub-endorsement framework, with the ABAC mechanism we design in section 3.5 and implement in section 4.2. We explain our design and implementation choices, their advantages and disadvantages as well as the

application lifecycle in chapter 3 and 4.
The major constraints of the blockchain are the previously described lack of trust and the limited access to blockchain external resources, as we need to find a way to access the attributes of SSL/TLS certificates. Furthermore, to allow updates to components of the system, we also consider the immutability of smart contracts once they are deployed. Finally, another constraint is the rather slow and expensive execution of access requests on a public blockchain compared to traditional authentication and access control systems.

The key contribution of our system is, that it allows access control with attributes within the trustless environment of the public Ethereum blockchain, by leveraging SSL/TLS certificates. Hence, with this work we complement the still young research for authentication and access control at digital resources that reside on public blockchain systems, but also contribute to the advancement of authentication and access control in general. Furthermore, we help to increase the maturity of public blockchain systems, as our research complements functionality that is important for consumer and business applications and widely supported by non-blockchain systems. The evaluation of our expert interviews show that there is general interest in authentication and access control systems on the blockchain for and beyond the use-cases we presented. Furthermore, interviewees like the idea of using the existing SSL/TLS certificate PKI to overcome a lack of trust on public blockchains and enable authentication and access control, instead of creating a new one. Last but not least, with our sub-endorsement framework we offer a solution with which *On-Chain AuthSC* can also be used by real-world entities that do not own a SSL/TLS certificate.

However, we are also aware of the limitations of our system. As we bootstrap the SSL/TLS certificate PKI, our system inherits its security issues and the trust in our system indirectly depends on the credibility of central Certification Authorities. Yet, most authentication and access control systems that are not located on the blockchain rely on central entities, while most systems on the blockchain neither have a centralized nor a decentralized source of trust. Furthermore, our sub-endorsement framework only allows indirect endorsements for real-world entities without SSL/TLS certificates. Hence, an intermediary entity that owns a Registry and is willing to sub-endorse the account of a real-world entity is needed. Consequently, the attributes of the Registry might not always perfectly describe a sub-endorsed user account. Moreover, the current implementation of the prototype still lacks complex access control policy evaluation, hence currently only policies with one attribute and the equality operator are supported. Finally, our prototype is slower and more expensive than most traditional blockchain-external authentication and access control systems. However, speed is a general constraint of most public blockchains, hence our system does not significantly influence it, but yet is affected by it. Nevertheless, as the implementation of the access request evaluation is not optimized yet with regards to speed and cost and we already propose solutions that increase performance and reduce cost, we are confident that this and other constraints can be mitigated and thus not hinder adoption of the system.

7.2 Future Work

The goal of our research is to elaborate a first system design and prototype implementation that allows to test the general concept of authentication and ABAC of real-world entities at smart contracts. As the general concept seems promising, future research can contribute by creating a literature review to further structure and explore the research field, or by improving our system with regards to performance and functionality. Furthermore, new applications, use-cases and sources of trust can be elaborated.

One major goal of future research should be to improve the performance of our system with regards to cost, speed and security. A starting point can be the access request evaluation in our ABAC framework, as it is frequently used and still is computation intensive. Currently the mechanism iterates through the subject field of a SSL/TLS certificate every time an access request is evaluated to retrieve the attributes. As the iteration is very performance intensive, we rather recommend to retrieve and decode the data stored in the subject field only once when a SSL/TLS certificate is added to the Certificate Database. The attribute type and value could be stored in variables that can easily be accessed during the access request check. An additional solution, that further reduces the required computation intensity, decreases the frequency of access control checks. The idea is to add the account address of the requester for a limited time to a whitelist at the Application, given its successful evaluation. Thus, the access request of an account only needs to be evaluated once within a certain timeframe. In this timeframe the account owner can use the Application as often as desired without any additional costs.
Our evaluation identifies areas to improve the performance, however it does not cover an analysis of the system's security. Therefore, after implementing the suggested improvements we recommend future research to conduct a holistic study that covers security and performance. Furthermore, as highlighted in chapter 5, future research should also conduct an evaluation with a Human Risk & Effectiveness strategy, to better assess the human social risks we identify in our evaluation.

In chapter 6 we introduce relevant structured and unstructured related work for blockchain-based authentication and access control. However, our literature review is mainly limited to Ethereum-based and ABAC systems, the research which is most related to our work. Future research should conduct a literature review of blockchain-based authentication and access control systems independent from the underlying blockchain system and the access control mechanisms. Such analysis structures the research field, identifies research gaps and allows researchers to identify related work which might significantly advance their research.
We also suggest future research to add further functionality to the system. Interesting for potential users might be the support of complex access control policies with multiple attributes and operators (e.g. unequal). This allows the owner of the Application to specify multiple desired attributes and different conditions, the endorsing SSL/TLS certificate of the user account needs to satisfy.
Currently in our system we assume that the owner of the endorsing SSL/TLS certificate and

the owner of the to-be-endorsed Registry are the same person. However, in some cases this is not the case. Hence, we suggest to include flags in the endorsement of the Registry with which the owner of the SSL/TLS certificate can specify how the endorsed trust and attributes of the SSL/TLS certificate can be used.

Finally, future research should explore support for other sources of trust than SSL/TLS certificates. Multiple sources of trust may increase the credibility of endorsements and access control decisions. Furthermore, it might increase the user base and improve usability, as it allows users to pick their favourite source of trust. Potential sources of trust to start the exploration with are certificates that are used within organizations or decentralized trust infrastructures as Decentralized Identifiers. Furthermore, the system could be tested in private blockchains, where trust could be provided by the owner of the blockchain.

List of Figures

1.1 Design Science Research Process from [53, pg.93] 5

2.1 Structure of a blockchain 11
2.2 Exemplary SSL/TLS certificate PKI hierarchy 17
2.3 Structure of a X.509 certificate [17] 19
2.4 Simplified architecture of On-Chain AuthSC adapted from [33, pg.56] 23
2.5 Selected attributes of certificates stored on the Certificate Database from [33, pg.63] 28

3.1 Functional requirements 35
3.2 Endorsement framework 38
3.3 Exemplary Access Matrix 44
3.4 Exemplary Access Control List 44
3.5 Schematic of Role-Based Access Control from [44, pg.4] 46
3.6 Schematic of functional entities of Attribute-Based Access Control from [41, pg.15] 48
3.7 Architecture of our blockchain-based ABAC mechanism for smart contracts . 55

4.1 Shortend interface of the Registry smart contract 61
4.2 Model of prototype implementation 63
4.3 Structure of the prototype implementation with components from [33] 66
4.4 Shortend and simplified interface of the Application smart contract 68
4.5 Shortend and simplified implementation of the PDP library 72

5.1 Scenarios at the Registry smart contract 85
5.2 Scenarios at the Application smart contract 87

List of Tables

2.1 Attribute types of subject and issuer fields [17] with respective OIDs [37] . . . 20

3.1 Attribute types [17] supported by our ABAC system with respective OIDs [37] and certificate types . . . 52

5.1 Cost and speed of execution for operations at the Registry smart contract . . . 84
5.2 Cost and speed of execution for operations at the Application smart contract . 88
5.3 Cost and speed of execution for the evaluation of an access request 89
5.4 Compliance of the prototype with regards to our requirements 91

Bibliography

[1] C. Allen, A. Brock, V. Buterin, J. Callas, D. Dorje, C. Lundkvist, P. Kravchenko, J. Nelson, D. Reed, M. Sabadello, G. Slepak, N. Thorp and H. T. Wood. *White paper: Decentralized Public Key Infrastructure*. Tech. rep. 2015, pp. 1–21.

[2] C. Allen, K. H. Duffy, R. Grant and D. Pape. *BTCR DID Method*. 2019. URL: `https://w3c-ccg.github.io/didm-btcr/` (visited on 24/10/2020).

[3] M. Almakhour, L. Sliman, A. E. Samhat and W. Gaaloul. 'Trustless blockchain-based access control in dynamic collaboration'. In: *CEUR Workshop Proceedings*. 2018, pp. 27–33.

[4] A. M. Antonopoulos and G. Wood. *Mastering Ethereum: Building Smart Contracts and DApps*. O'Reilly Media, Incorporated, 2018. ISBN: 9781491971949.

[5] *Authereum*. 2020. URL: `https://authereum.com/` (visited on 03/11/2020).

[6] B. Ballad, T. Ballad and E. Banks. *Access Control, Authentication, and Public Key Infrastructure*. Jones & Bartlett Publishers, 2010, pp. 1–398. ISBN: 9780763791285.

[7] I. Bashir. *Mastering Blockchain: Distributed ledger technology, decentralization, and smart contracts explained, 2nd Edition*. Packt Publishing, 2018. ISBN: 9781788839044.

[8] M. Benantar. *Access Control Systems: Security, Identity Management and Trust Models*. Springer, 2005. ISBN: 0387004459.

[9] K. Bendiab, N. Kolokotronis, S. Shiaeles and S. Boucherkha. 'WiP: A novel blockchain-based trust model for cloud identity management'. In: *Proceedings - IEEE 16th International Conference on Dependable, Autonomic and Secure Computing, IEEE 16th International Conference on Pervasive Intelligence and Computing, IEEE 4th International Conference on Big Data Intelligence and Computing and IEEE 3rd Cyber Science and Technology Congress, DASC-PICom-DataCom-CyberSciTec 2018*. 2018, pp. 724–729.

[10] K. Bosworth, M. G. Gonzalez Lee, S. Jaweed and T. Wright. 'Entities, identities, identifiers and credentials - What does it all mean?' In: *BT Technology Journal* 23 (2005), pp. 25–36.

[11] D. Bryson. *Simple authentication using your Ethereum address*. 2016. URL: `https://github.com/davebryson/ethauth-js` (visited on 03/11/2020).

[12] F. Buccafurri, C. Labrini and L. Musarella. 'Smart-contract Based Access Control on Distributed Information in a Smart-City Scenario'. In: *DLT@ITASEC*. 2020.

[13] V. Buterin. *Ethereum Whitepaper: A Next Generation Smart Contract & Decentralized Application Platform*. Tech. rep. 2013, pp. 1–36.

[14] CA/Browser Forum. *Baseline Requirements for the Issuance and Management of Publicly-Trusted Certificates v. 1.7.3*. Tech. rep. 2020, pp. 1–96.

[15] CA/Browser Forum. *Guidelines For The Issuance And Management OfExtended Validation Certificates v. 1.7.4*. Tech. rep. 2020, pp. 1–53.

[16] Y. Cao and L. Yang. 'A survey of Identity Management technology'. In: *Proceedings 2010 IEEE International Conference on Information Theory and Information Security, ICITIS 2010*. 2010, pp. 287–293.

[17] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley and W. Polk. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. Tech. rep. 2008, pp. 1–151.

[18] J. P. Cruz and Y. Kaji. 'The Bitcoin Network as Platform for Trans-Organizational Attribute Authentication'. In: *IPSJ Transactions on Mathematical Modeling and Its Applications* 9.2 (2016), pp. 41–48.

[19] J. P. Cruz, Y. Kaji and N. Yanai. 'RBAC-SC: Role-based access control using smart contract'. In: *IEEE Access* 6 (2018), pp. 12240–12251.

[20] W. Dai, C. Wang, C. Cui, H. Jin and X. Lv. 'Blockchain-Based Smart Contract Access Control System'. In: *25th Asia-Pacific Conference on Communications (APCC)*. 2019, pp. 19–23.

[21] D. Di Francesco Maesa, A. Lunardelli, P. Mori and L. Ricci. 'Exploiting Blockchain Technology for Attribute Management in Access Control Systems'. In: *Economics of Grids, Clouds, Systems, and Services*. 2019, pp. 3–14.

[22] D. Di Francesco Maesa, P. Mori and L. Ricci. 'A blockchain based approach for the definition of auditable Access Control systems'. In: *Computers and Security* (2019), pp. 93–119.

[23] D. Di Francesco Maesa, P. Mori and L. Ricci. 'Blockchain Based Access Control Services'. In: *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. 2018, pp. 1379–1386.

[24] O. Dib, K.-L. Brousmiche, A. Durand, E. Thea and E. B. Hamida. 'Consortium blockchains: Overview, applications and challenges'. In: *International Journal On Advances in Telecommunications* 11.1&2 (2018), pp. 51–64.

[25] T. Dierks and E. Rescorla. *RFC 5246 - The transport layer security (TLS) protocol - Version 1.2*. Tech. rep. 2008, pp. 1–104.

[26] *Ethereum Docs*. 2020. URL: `https://ethereum.org/en/developers/docs` (visited on 10/10/2020).

[27] *Ethereum Docs - EVM*. 2020. URL: `https://ethereum.org/en/developers/docs/evm/` (visited on 10/10/2020).

[28] Eurostat. *List of the recognized research entities*. 21st Aug. 2020. URL: `https://ec.europa.eu/eurostat/documents/203647/771732/Recognised-research-entities.pdf` (visited on 09/11/2020).

[29] N. Fotiou, I. Pittaras, V. A. Siris, S. Voulgaris and G. C. Polyzos. *OAuth 2.0 authorization using blockchain-based tokens*. 2020. arXiv: `2001.10461 [cs.CR]`.

[30] Friederike Kleinfercher, Dr. Sandra Vengadasalam, James Lawton. *Bloxberg - The trusted Research Infrastructure - Whitepaper 1.1*. Tech. rep. Bloxberg, 2020, pp. 1–26.

[31] U. Gallersdörfer, P. Holl and F. Matthes. *Blockchain-based Systems Engineering – Lecture Slides*. Oct. 2019.

[32] U. Gallersdörfer and F. Matthes. 'TeSC: Mind the Gap between Web and Smart Contracts'. unpublished.

[33] F. Groschupp. 'Exploring the Use of SSL/TLS Certificates for Identity Assertion and Verification in Ethereum'. MA thesis. Technical University of Munich, 2020, pp. 1–83.

[34] A. Gruener, A. Mühle, T. Gayvoronskaya and C. Meinel. 'Towards a Blockchain-based Identity Provider'. In: *SECURWARE 2018: The Twelfth International Conference on Emerging Security Information, Systems and Technologies*. 2018, pp. 73–78.

[35] H. Guo, E. Meamari and C.-C. Shen. *Multi-Authority Attribute-Based Access Control with Smart Contract*. 2019. arXiv: `1903.07009 [cs.CR]`.

[36] A. R. Hevner, S. T. March, J. Park and S. Ram. 'Design science in information systems research'. In: *MIS Quarterly: Management Information Systems* 28.1 (2004), pp. 75–106.

[37] P. Hoffman and J. Schaad. *New ASN.1 Modules for the Public Key Infrastructure Using X.509 (PKIX)*. Tech. rep. 2010, pp. 1–117.

[38] 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 ACM SIGCOMM Internet Measurement Conference, IMC*. 2011, pp. 427–444.

[39] V. C. Hu, D. R. Kuhn and D. F. Ferraiolo. 'Access Control for Emerging Distributed Systems'. In: *Computer* 51.10 (2018), pp. 100–103.

[40] V. C. V. Hu, D. F. Ferraiolo and D. R. Kuhn. *Nistir 7316: Assessment of access control systems*. Tech. rep. 2006, pp. 1–51.

[41] V. C. Hu, D. Ferraiolo, R. Kuhn, A. Schnitzer, K. Sandlin, R. Miller and K. Scarfone. *NIST Special Publication 800-162 - Guide to Attribute Based Access Control (ABAC) Definition and Considerations*. Tech. rep. 2014, pp. 1–37.

[42] *Hyperledger Fabric Docs - Membership Service Provider (MSP)*. 2020. URL: `https://github.com/hyperledger/fabric/blob/release-2.2/docs/source/membership/membership.md` (visited on 06/11/2020).

[43] IBM. *Fabric-contract-attribute-based-access-control*. 2020. URL: `https://github.com/IBM/fabric-contract-attribute-based-access-control` (visited on 05/11/2020).

[44] International Committee for Information Technology Standards. *ANSI INCITS 359-2004*. Tech. rep. 2004, pp. 1–47.

[45] M. Jemel and A. Serhrouchni. 'Decentralized Access Control Mechanism with Temporal Dimension Based on Blockchain'. In: *Proceedings - 14th IEEE International Conference on E-Business Engineering, ICEBE 2017 - Including 13th Workshop on Service-Oriented Applications, Integration and Collaboration, SOAIC 207*. 2017, pp. 177–182.

[46] Y. Jiang, C. Lin, H. Yin and Z. Tan. 'Security analysis of mandatory access control model'. In: *Conference Proceedings - IEEE International Conference on Systems, Man and Cybernetics*. 2004, pp. 5013–5018.

[47] P. Kohlhaas. *Zug ID: Exploring the First Publicly Verified Blockchain Identity*. 2017. URL: `https://medium.com/uport/zug-id-exploring-the-first-publicly-verified-blockchain-identity-38bd0ee3702` (visited on 05/11/2020).

[48] N. Leavitt. 'Internet security under attack: The undermining of digital certificates'. In: *Computer* 44.12 (2011), pp. 17–20.

[49] A. Miller. *Never Use Passwords Again with Ethereum and Metamask*. 2017. URL: `https://hackernoon.com/never-use-passwords-again-with-ethereum-and-metamask-b61c7e409f0d` (visited on 03/11/2020).

[50] S. Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. Tech. rep. 2008, pp. 1–9.

[51] OpenZeppelin. *OpenZeppelin Docs - Access Control*. 2020. URL: `https://docs.openzeppelin.com/contracts/3.x/access-control` (visited on 06/11/2020).

[52] A. Ouaddah, A. Abou Elkalam and A. Ait Ouahman. 'FairAccess: a new Blockchain-based access control framework for the Internet of Things'. In: *Security and Communication Networks* 9.18 (2016), pp. 5943–5964.

[53] K. Peffers, T. Tuunanen, C. E. Gengler, M. Rossi, W. Hui, V. Virtanen and J. Bragge. 'The Design Science Research Process: A Model for Producing and Presenting Information Systems Research'. In: *1st International Conference, DESRIST 2006 Proceedings*. 2006, pp. 83–106.

[54] *QuantumProductions - Eth-Auth*. 2018. URL: `https://github.com/quantumproductions/eth-auth` (visited on 03/11/2020).

[55] D. Reed, M. Sporny, C. Allen, L. Dave, R. Grant, M. Sabadello and J. Holt. *Decentralized Identifiers (DIDs) v1.0*. 2020.

[56] M. Røed. *ethauth*. 2017. URL: `https://github.com/martolini/ethauth` (visited on 03/11/2020).

[57] P. Samarati and S. d. C. di Vimercati. 'Access control: Policies, models, and mechanisms'. In: *Foundations of Security Analysis and Design*. 2001, pp. 137–196.

[58] R. S. Sandhu, E. J. Coyne, H. L. Feinstein and C. E. Youman. 'Role-based access control models'. In: *Computer* 29.2 (1996), pp. 38–47.

[59] R. S. Sandhu and P. Samarati. 'Access Control: Principles and Practice'. In: *IEEE Communications Magazine* 32.9 (1994), pp. 40–48.

[60] M. Shumikhin, S. Wooders and R. Senanayake. *6.858 Final Project: Distributed Authentication on the Ethereum Blockchain*. 2018.

[61] V. A. Siris, D. Dimopoulos, N. Fotiou, S. Voulgaris and G. C. Polyzos. *OAuth 2.0 meets Blockchain for Authorization in Constrained IoT Environments*. 2019. arXiv: 1905.01665 [cs.CR].

[62] *Solidity Docs v0.7.3*. 2020. URL: https://github.com/ethereum/solidity/blob/v0.7.3/docs/index.rst (visited on 14/10/2020).

[63] Sovrin Foundation. *Sovrin: A Protocol and Token for Self- Sovereign Identity and Decentralized Trust*. Tech. rep. 2018, pp. 1–42.

[64] V. Suhendra. 'A survey on access control deployment'. In: *Security Technology*. 2011, pp. 11–20.

[65] *UniLogin*. 2020. URL: https://unilogin.io/ (visited on 03/11/2020).

[66] *uPort Specs*. 2020. URL: https://github.com/uport-project/specs (visited on 05/11/2020).

[67] J. Venable, J. Pries-Heje and R. Baskerville. 'FEDS: a Framework for Evaluation in Design Science Research'. In: *European Journal of Information Systems* 25.1 (2016), pp. 77–89.

[68] G. Wood. 'Ethereum: A secure decentralised generalised transaction ledger'. In: *Ethereum Project Yellow Paper* (2014), pp. 1–39.

[69] Y. Zhang, S. Kasahara, Y. Shen, X. Jiang and J. Wan. 'Smart Contract-Based Access Control for the Internet of Things'. In: *IEEE Internet of Things Journal* 6.2 (2019), pp. 1594–1605.

[70] P. Zimmermann. *PGP User's Guide, Volume I: Essential Topics*. 1994.