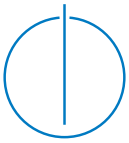
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Modelling and Implementation of Access Control Mechanisms in Ethereum Smart Contracts

Thomas Hain



DEPARTMENT OF INFORMATICS

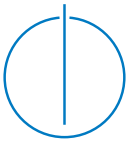
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Modelling and Implementation of Access Control Mechanisms in Ethereum Smart Contracts

Modellierung und Implementierung von Zugriffskontrollmechanismen in Ethereum Smart Contracts

Author:	Thomas Hain
Supervisor:	Professor Dr. Florian Matthes
Advisor:	Ulrich Gallersdörfer, M.Sc.
Submission Date:	December 16th, 2019



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

Munich, December 16th, 2019 Thomas Hain

Acknowledgments

First, I would like to thank my advisor Ulrich Gallersdörfer. By providing advice and ideas he served as a constant stream of input in the process of directing and shaping this thesis. He was always open for discussing crucial parts concerning my research and often provided valuable insights and perspectives concerning the Blockchain technology. In addition he promoted an environment which allowed me to envision and describe my own understanding of the topic.

Additionally I want to thank Professor Dr. Florian Matthes for his help in the process of formulating the thesis title and its goals and for the opportunity to write this thesis at the Software Engineering for Business Information Systems (SEBIS) chair.

Furthermore I want to thank my family and friends for supporting and encouraging me throughout the whole process of conducting my research.

Thank you.

Abstract

The Ethereum Blockchain has gained a lot of popularity within the recent years. While it is often mentioned alongside Bitcoin, it receives an increasing amount of attention by itself. One of the reasons for the rising interest lies in systems lies in its capability of encoding enforceable code on its Blockchain. This allows an automatic and transparent transfer of funds between the network's participants and eliminates the need for a Trusted Third Party. Further it is currently being the subject of diverse researchers as its potential use cases expand beyond the financial domain. In the past there have been several successful attacks however. Oftentimes they might have been prevented by a more sophisticated model of access control. As Smart Contracts are being exposed to the complete network this leaves many possible attack vectors. This is underlined by the fact that these programs can hold big amounts of currency. This makes it necessary to carefully evaluate one's security model and to assess each component's required degree of public exposure. This is being further complicated by the fact that Ethereum's most popular programming language Solidity suffers from a lot of shortcomings compared to more established ones oftentimes leading to programming errors which can only be debugged by rather primitive means. As a consequence the thesis includes an evaluation of different implementations of access control in Ethereum and derives their commonalities in order to derive desirable features of a new implementation. In addition it provides insights about the topic of data privacy in Blockchain systems by describing the permissioned Blockchain Quorum. It provides a perspective on how to approach the decision making process behind publicly exposing one's system to a network of multiple competing nodes and gives warnings about related implications. The findings are then being accumulated and formulated into a model finally being transferred into the implementation of an own modified XACML based system. This prototype provides a reusable and flexible framework for future implementations.

Contents

1 Introduction

1.1 Motivation

With the rise of Authorization frameworks like "XACML"[87] and "OAuth"[90] there's an increasing amount of flexible authentication systems. At the same time monolithic architectures are slowly being replaced by cloud services to ensure a higher availability and scalability. Managing access to these distributed systems is an increasingly complex task. While over the years multiple solutions to these problems emerged many Blockchain applications are still resorting to very basic ways of protecting their functionality. Thus a unified access control solution could potentially serve as a starting point for increasing the overall security of decentralized applications. Thus it poses an interesting target for research. Access Control is strongly linked to the currently on-going debate about data privacy on Blockchains which is even being under observation by Vitalik Buterin the inventor of the Ethereum chain[86]. As private data often poses a valuable target to attacks topics like information security and cryptography play a major role in designing a secure system for regulating access on the Ethereum Blockchain. Because Ethereum allows storing small programs also known as "Smart Contracts"[41] on its chain they can be analyzed for their capability of provisioning and managing access rights. Many of these contracts solve relatively small tasks and are therefore rather simple compared to bigger software developed in older programming languages such as Java or C++. On the other hand Ethereum's limits regarding computation are being an obstacle in finding a common solution for the problem of regulating access. Incorporating them into the design of newly developed systems introduces several adaptations of existing solutions. While historical systems often rely on dividing their network infrastructure into individually secured servers the interaction with these publicly distributed programs can be is transparent to the public[96]. This introduces a whole new level of complexity to the current understanding of evaluating access rights. Many of the traditional concepts rely on the establishment of trust between multiple parties.[98] Because Ethereum is being highly transparent to its participants the public visibility of the decision making process behind the revocation or granting of rights can be made publicly visible[25]. Still there are many different scenarios where such a degree of transparency is not required or even strictly opposed to the intended functionality of a system[96]. Therefore careful considerations have to be taken in order to determine whether a Blockchain solution can be a reasonable choice for a given scenario.

Thus the thesis tries to provide information about which limitations and benefits a Blockchain based access control system offers by presenting a prototype implementation of multiple Smart Contracts working together in order to form an extendable decentral-

ized application. This directly contributes to the development of Ethereum and similar Blockchains supporting Smart Contracts. Thus the thesis' aim is to provide a flexible approach which can be adapted to a wide range of existing problems.

As the subject of access control is based on multiple disciplines including network architectures, protocols and cryptography it also provides an overview about the degree of inter connectivity between these topics. As there is no common consensus in the Blockchain community on how a possible implementation of an access control system could look like the thesis offers a common ground for further discussion by providing a fully implemented system including the documentation of the underlying design choices. This extends previous research in this area as at the time of writing there are few published works in which include full implementation details. In addition the thesis branches off into the controversial topic of "private" and "permissioned"[42] Blockchains and discusses their usefulness regarding varying requirements of data privacy. This way it provides guidelines for any company exploring Ethereum's capabilities or researchers interested in expanding their understanding of access control systems. As a consequence the thesis implementation tries to accommodate features to allow an easy adaptation to individual security requirements.

1.2 Research Questions

During the starting phase of conducting the research multiple questions arose. As a consequence they are being included in this chapter to give an overview about the questions the thesis answers in subsequent chapters.

Q1: What are current challenges regarding the implementation of access control on a Blockchain?
To be able to understand existing implementations of access control systems it is necessary to evaluate the motivation behind them first. What problems do they try to solve and how are they approaching them? This information can be aggregated and evaluated in order to provide possible perspectives on common obstacles and derive re occuring patterns. What measurements can be taken in order to overcome them and to advance current research?

Q2: What is the current state of implementations regarding access control in Solidity?
There are different existing implementations of access control systems implemented in Solidity. They can be found in literature as well as online repositories such as GitHub. Therefore a general evaluation of the various sources is required in order to assess the community's understanding of the subject. These findings could prove to be valuable in the design process of an own prototype. Motivated by this idea the thesis analyzes code samples for similarities in order to find possible improvements.

Q3: Which advantages does using Blockchain technology provide for access control?

Making use of the Blockchain technology comes with various implications. As Blockchains are often open and highly transparent systems[29] they might provide advantages over existing non-Blockchain solutions. How can the technology be leveraged to maximize its benefits and how do they compare to traditional systems?

Q4: How can an extendable access control system be modelled and implemented?
The thesis tries to aggregate the results of the previous research questions in order to derive its own implementation of a reference prototype. The modelling process is being influenced by each of the previous research questions. The provided system is then being evaluated in order to find out whether existing approaches could be improved. Both the implementation and modelling can provide potentially valuable insights for other researches trying to understand the design decisions. This is motivated by the possibility of providing a point of reference for future discussion on the subject.

1.3 Approach

The research conducted in this thesis is based on an iterative narrowing down of collected information. As the early stages of conducting research resulted in a big collection of data it had to be grouped multiple times according to its categories. By repeatedly contextualizing the accumulated data the underlying principles became clearer and their connections became more apparent. These findings ultimately led to the formulation of concrete design goals and led to the decision to implement a prototype. As the thesis is inspired by multiple sources of literature its discovery and analysis was the first step throughout research. In addition the sources served as constant reference points during orientation processes.

- **Literature Research** In order to answer the stated research questions it was necessary to first understand the current state of several different research areas ranging from modern web applications to Blockchain technology. As each of the analyzed disciplines evolved by developing its own philosophies and best practices the primary concern was dissecting them for the purpose of understanding their most essential parts and the problems they try to solve.

- **Analysis of Smart Contracts** In addition to the literature research itself the implementation of existing access control libraries was examined. Oftentimes this included manual inspection of their source codes as the documentation was rather limited. This included multiple contracts hosted on different repositories. As a result of this analysis the thesis conducts two code analyses. Their findings helped in shaping the system reach its final state.

- **Testnetworks and Implementation Tests** After explaining some commonalities of current strategies for enforcing access restrictions the thesis branches off into an actual implementation of a prototype system. In order to prove its stability the thesis provides a suite of unit tests based on a publicly available JavaScript testing framework for Smart Contracts. Each of these tests were run on multiple test networks both locally and public. This ensures an easy reconstruction of the stated results making them provable. The testing also included experiments with publicly available projects such as "zk-SNARKS"[35] and node-Casbin[**casbi**].

1.4 Outline

The thesis is structured in multiple chapters. Each of them provides information which ultimately led to the implementation of a prototype. In order to follow the decisions involved in its design process the fundamentals of networks, encryption and Blockchain's need to be understood first. Therefore each of these topics is being described in chapter 2: "Fundamentals". Chapter 3 then proceeds by giving an overview of existing publications and how they relate to the subject. Then chapter 4 proceeds by explaining basic concepts of access control and includes an evaluation of three different existing systems. In addition it provides an introduction in Solidity programming because two of the evaluations require a basic understanding of the language. As a result of the final evaluation chapter 5 begins by outlining a proposal for improving an existing system and then concludes with a finished implementation of a prototype. Afterwards the prototype is both being compared with a related project and being tested in chapter 6. The final chapter contains a conclusion of the findings and a discussion about future work regarding the subject.

Disclaimer: The thesis' text makes use of terms in the generic masculine in order to ensure an easier readability. However both female and male readers are meant equally.

2 Fundamentals

As the presented technologies all rely on a solid understanding of securing the propagation of messages through networks it is a necessity to lie the groundwork first. Securing messages in a network is often done without a user's knowledge. Each time a user opens a website like PayPal or Google its connection is secured in order to protect sensitive parts of the exchange of data. As Blockchain technology implies a perspective on desirable infrastructure it is explained and compared to traditional web servers. Both Blockchains and traditional approaches make use of cryptography to protect data and prove one's identity. Thus this chapter starts off by giving a brief introduction into the topic of network architectures including the common approach of separating different functionality into "layers"[51] and then continues by describing multiple types of cryptography. Each of these topics are relied on in later chapters of the thesis and are crucial in the context of exercising access control on Blockchains.

2.1 Networks, Requests & Resources

Networks are the foundation of the current degree of intercommunication humans face in the modern world. Whenever a user interacts with a website messages are exchanged between the user also known as "client"[65] and the host commonly referred to as "server"[65]. The interaction between a client and a server follows strict protocols and architectural choices to ensure an efficient transmission of data between them[43]. These design principles are constantly evolving since the internet surfaced. However its core technologies remain mostly unchanged. With the goal of formulating desirable properties of communication the following sections therefore explain the implications of traditional web-based software.

2.1.1 Client-Server & Peer To Peer

The most basic form of network communication comes in the form of "Client-Server Architecture"[60]. In a pure client-server architecture there is one single central server simultaneously handling the requests of multiple different clients[60]. A concrete example is a chat server. It maintains a list of all connected clients while listening to each client's messages at the same time. Whenever it receives a message from a client it distributes it to all other connected clients. This architectural style has the advantage that each participant only needs to know how to connect to the server instead of requiring each client to be able to directly connect to everyone else. On the other hand this approach is considered to be rather vulnerable due to being centered around

a "Single Point of Attack"[89]. The term highlights the negative impact of basing one's architecture around a central server. If it fails or crashes the downfall of the complete infrastructure follows. As a consequence modern infrastructures rely on a distribution of their services to multiple servers instead[90]. By doing so an attacker gaining access to one of the servers is not automatically having control over the other server as well. Due to an increase in complexity of modern websites application servers are added as another layer operating behind traditional web servers. A single website often makes use of multiple application servers to provide a higher degree of scalability as each of them is potentially linked to its own database server.

The understanding of how to scale infrastructures is crucial to big software companies trying to reach a large customer base. As a consequence big companies like Netflix invest lots of effort in the maintenance of their infrastructures. However there is an alternative. The technology "Peer to Peer"[39] allows the connection of a possibly infinite amount of computers thus presenting a flexible and scalable alternative without relying on a central server[39]. This is achieved by establishing connections between all the network's participants known as "nodes"[1]. In case a node disconnects the system remains intact as the transfer of its messages doesn't follow a strict path. In addition the absence of a Single Point of Attack makes it more resilient. This is underlined by the online piracy community using a Peer to Peer based protocol and file-sharing system known as "Torrents"[68]. If a user illegally uploads a movie to the network it can only hardly be taken down via copyright claims because the files are distributed between multiple participating nodes[68]. The reason for this is that such file-sharing protocols often automatically turn participants who downloaded parts of the movie into distributors themselves.[68] Consequentially this results in multiple coexisting copies of the movie on each participant's hard drive. In order to enforce copyright claims each participant would be required to delete that file. This circumstance is of such importance that it still poses a problem to current legal institutions.[68] But there are not only advantages of using Peer to Peer. Because there is no guarantee that two participants have an adequate connectivity data transfer varies greatly.[17] Commonly this is known as "propagation speed"[17].

2.1.2 HTTP Requests

Both of the previously presented infrastructures are based around messages. In order for a network participant to understand how to participate in their exchange it requires additional information on how to behave. This knowledge is referred to as "protocol".[65] In the previous section BitTorrent was already introduced as an example of a popular Peer to Peer protocol. One commonly used protocol in the client-server domain is called "Hypertext Transfer Protocol" (HTTP)[65]. The underlying principle is rather simple: The client sends a request message and the server responds to it.[65] The actual content of these messages can vary depending on the use case. As HTTP is defined as "stateless"[49] each request is handled individually without being affected by previous requests.[49] The server responsible for handling incoming HTTP requests is called "web

server"[49].

Whenever a client sends an HTTP request it encodes its "method"[65] and a header to include additional information fields.[49] In modern Access Control frameworks like "OAuth2"[90] a header can include information about a client's authorization to execute a request[90]. HTTP is based on the "four basic operations"[85]:

- **HTTP GET** for fetching information from the server
- **HTTP POST** for sending information to the server
- **HTTP PUT** for updating server data
- **HTTP DELETE** for deletion of information from the server

Similar to the way the HTTP Protocol is responsible for defining a common behavior the "REST"[85] (Representational State Transfer) standard revolves around addressing "resources"[85]. Resources can be thought of as references to any type of data (e.g. text, movie,...). Combining HTTP with REST is the foundation of request-based retrieval and alteration of data stored in various sources like databases. While there are varying definitions of REST it can be described based on a set of five different constraints:

- Resources are identified by a resource identifier
- Access methods have the same semantics for all resources
- Resources are manipulated by exchanging their representations
- Representations are being manipulated via messages
- The application state is handled via Hypertext

In other words: the access to a resource is carried out via HTTP. In case the resources are access restricted they follow a common convention. The referenced data can be altered via messages carrying the information about which alterations to make. By making use of identifiers resources data can be handled equally and independently of their type. They are commonly being referred to as "Uniform Resource Identifier"[91] or "URI"[91]. Often they are linked to database indices referring to data sets identified by a unique id field.

Web servers often expose their functionality via a so called "API"[61] (Application Programming Interface). Combining this approach with the principles of REST results in a a concept commonly referred to as "REST API"[61]. As its name indicates a REST API provides a central point responsible for answering incoming requests. Making use of APIs allows a wide range of possible applications because there are many publicly accessible APIs. The naive approach of defining a resource identifier is to combine a resource's type with its index in the database. As viewing a website does nothing else than sending an HTTP GET request to a webserver and receiving its HTML content a response to a fictional URL including a referenced resource might therefore look like:

```
https://www.myserver.com/users/1
```

This way the server is directly able to lookup said resource via its index in the corresponding table by mapping this request to a corresponding database query. Due to the simpleness of this approach modern web frameworks such as "Laravel"[50] allow the definition of "Resource Routes"[50] which by default include a predefined set of standard operations. This quickly provides multiple publicly accessible points. Because their individual functionality can be defined by the programmer of the web application its use cases are unlimited. As an example of a definition of end points responsible for varying requests involving the storage, viewing and updating of a photo resource the following routes could be the result of an automatic generation via Laravel's command line interface.

Method	**URI**	**Controller Action**
GET / HEAD	api/photos	index
POST	api/photos	store
GET / HEAD	api/photos/{photo}	show
PUT / PATCH	api/photos/{photo}	update
DELETE	api/photos/{photo}	destroy

Table 2.1: Example result of automatic API route generation

The variable "photo" indicated by the parentheses has to be provided by the request. It is also known as "query string". One way a user can set it is to access the interface by supplying a corresponding value to the URL:

```
https://www.myserver.com/api/photos/1/show
```

As the table already indicates the resources' URIs are assigned a controller's action. The code in the controller then defines how to process the received request including its parameters.[50] This could include setting the content of the response to a data set retrieved from a database by its id. Other types of requests might be altering referenced data instead. This can be seen by the "DELETE" entry within the table. Its controller code could invoke the removal of data from the database. Other operations such as updating or creating new objects can potentially require multiple parameters. Consequentially query parameters are not the only possible way of passing additional data. In addition the controller can use information supplied by the request in order to check a user's authorization status. In case the user is not allowed to access a resource can then be responded to with an error code, such as 401.

While the previous examples were mostly based on the understanding that a user accesses a website via a browser this is by no means a necessity. As the name API already suggests it is being used as an interface for applications instead. In addition it allows the communication between different web services via messages. Therefore

the publicly accessible routes of APIs often define a required data format for incoming requests. A common format also used within the context of Blockchains[36] is "JSON" [74] (JavaScript Object Notation). The name already reveals two facts about it: first it is used by the scripting language JavaScript and second it provides a way of serializing objects. This allows an easy transfer between interacting web servers. A JSON-formatted object including multiple students could have the following representation:

```
{
   "students":[
     {"firstName":"John", "lastName":"Doe"},
     {"firstName":"Anna", "lastName":"Smith"},
   ]
}
```

Listing 2.1: JSON File Structure

Another rather common way of serialization is to encode data in "XML"[74] (Extensible Markup Language). Its explanation is part of the thesis' chapter about XACML.

2.1.3 Layered Design

The separation between data storage and operations is often solved by a "layered architecture". According to Richards[73] a rather common solution is to structure one's software into the following layers: presentation, business, persistence and database. Each layer only communicates with others via interfaces.[73] In other words the persistence layer does not require any information about how a business layer internally handles incoming requests. Instead it only needs to know how to interact with it. Likewise the presentation layer (e.g. an Android client) does not need to be able to execute operations on the server's database directly. It only needs to know how to request functionality exposed by an API's routes.

In the domain of web servers where data is often stored on a separate machine or cloud service applying this architecture is rather straight-forward. It is important to note however that the architecture's use cases are in no way limited to web services. While the previous examples implied using a database as a target of the persistence layer there are other options as well. The actual storage of data can be as simple as a single file serializing all the information (e.g. JSON). Because access control often requires the storage of "policies"[54] in addition to the data itself the chapter about the Ethereum Blockchain discusses to which degree a Blockchain can be used as a persistence layer and how Ethereum's functionality can be exposed to users via a layered approach similar to traditional web applications.

2.2 Basics of Cryptography

The communication within networks relies on the transmission of messages. Depending on which participants receive the message they might be able to acquire information

about its content or even modify it. In a public Blockchain system like Bitcoin messages are transmitted transparently[29] to every participant of the network making it an easy target for attacks.[1] Likewise HTTP requests are transferred in plain-text and are therefore considered insecure and vulnerable. In order to ensure that data is only accessible by the intended audience encryption serves as a common tool. Applying encryption is a common means of ensuring confidentiality of data.[15] HTTP requests are therefore often guarded by "TLS"[43] encryption. This way their data is being protected from being publicly readable. HTTP's encrypted counterpart is known as "HTTPS"[43] (Hypertext Transfer Protocol Secure). Later chapters include information about how a Blockchain called "Quorum"[42] includes HTTPS messages in order to send private data. Understanding the importance of ensuring confidentiality directly leads to an understanding of access control. Further the following chapters include multiple approaches limiting a users' access to resources solely by cryptographical measurements. In addition Blockchain technology makes heavy use of the concepts explained in the following sections. Thus this chapter includes a short summary of the basic technologies involved in the context of Blockchains and Client-Server architectures.

2.2.1 Hash-Functions & Merkle Trees

Hash-Functions map input text of any length to an output string of a fixed length.[81] A typical example is the so called "SHA-256" function. It is classified as being a "one-way hash-function"[81] meaning that its near impossible to reconstruct the original input value from a given output value. This allows representing any given file by its respective hash value. As the name suggests SHA256 always produces an output of 256Bits independent of the input's length.

```
1 SHA-256Hash("StringWhichIsBeingHashed") =
2 "3CF4A6B8D9DFBD75A8F0A3EBE91F605054D45A0718E462211B3FC7539AF1B7BB"
```

Listing 2.2: Example of applying a SHA256 Hash Function

While the definition of hash functions which map multiple inputs to the same output is possible the subcategory of cryptographical hashes tries to minimize the probability of such an occurrence known as "collision"[22] to an extreme. SHA256 is one representative of this group. This makes it a suitable unique identifier for files because there are practically no overlaps. This technique is often used in so called "Content Addressable Storage"[28] (CAS) systems which handle the storage and retrieval of data sets by using their hash values as references instead of a traditional continuous index known from a typical database system[28]. One example of such a system is the distributed network "IPFS"[28] (Interplenetary File System). It is often used in conjunction with a common software pattern used in Blockchain context known as "Content-Addressable Storage"[28]. This term will reoccur at later parts off the thesis as it plays a role in many different applications.

Another consequence of the low probability of collisions is that whenever its input (e.g. the binary representation of a file) changes its hash value changes as well. Thus it

is possible to find out whether data has been changed.

Because a network and its messages are often exposed to manipulation attempts hash values serve as a protection mechanism ensuring that data remains unaltered between its sender and its receiver[71]. Thus hash values are often used in order to ensure the "integrity"[71] of data. One prominent example of a hash-based mechanism used in order to ensure the integrity of messages is known as "HMAC"[71] (Keyed-Hash Message Authentication Code). Because using the same input for a hash value results in the same output they are also used to store passwords securely in databases. Instead of storing the actual password its hash representation is stored. This way whenever a user tries to authenticate himself at a later stage the web server only needs to calculate the hash value of the supplied password and compare it with the stored value. If this technique is applied correctly it offers an important advantage over storing the actual data: If an attacker is able to hijack the server he can't reconstruct the actual password.

By combining multiple hash values into a data structure known as "Merkle Patricia Tree"[88] a Blockchain network is able to protect the integrity of messages its participants regard as valid. A Merkle tree is a recursively constructed tree of hash values. Each node is obtained by concatenating its leaves' values and hashing them afterwards. The leaves are hash values themselves. As previously stated whenever the represented data changes its hash value requires recalculation. As a consequence this step is recursively executed for all parent nodes whenever an alteration of one of its children occurs ultimately resulting in an update of the root node. As the root node is being calculated as the hash-value of the concatenation of all its leaves' hash values it serves as a single value ensuring the integrity of all its children. This is why it serves as an efficient way of protecting multiple messages at once.

2.2.2 Symmetric & Asymmetric Encryption

The importance of encryption on today's communication is observable in multiple forms. In June 2017 the whistleblower Edward Snowden even posted a message on twitter stating that the US government is unable to decrypt PGP encrypted documents[79]. While PGP itself is not especially relevant in Blockchain scenarios, this example showcases the impact these technologies have on modern communication.

On the other hand the governments themselves make use of encryption for concerns of confidentiality.[63] This property is especially important in the light of access control systems as they often require the transmission of passwords or similar secret data. Such data is often being used to initially authenticate a user to a web server in order to determine whether he can access a resource or not. This can be regarded as the most basic form of access control in web scenarios. Because authentication is one of the essential components of access control this thesis includes a dedicated section about it. Therefore further explanation about the details of the various authentication mechanisms are omitted here. Instead the underlying types of encryption are explained.

- **Symmetric Encryption** uses a single key to both encrypt and decrypt informa-

tion.[16] A common example is the encryption algorithm "AES"[63] (Advanced Encryption Standard). It is defined as a "block cipher"[45] processing the source data in equally sized blocks of multiple Bits[45]. Oftentimes these blocks use a size of 128 Bits.[45] Following the naming convention of appending the key length in Bits multiple AES implementations exist. This includes "AES-192"[7] and "AES-256"[7]. AES is explicitly approved by the US government and therefore is considered to be secure and suitable for highly confidential data[8]. In addition it can be used during the communication via HTTPS.

- **Asymmetric Encryption** Other than symmetric encryption its counterpart asymmetric encryption requires the usage of two keys.[16] These keys are called "private"[16] and "public"[16] key. They both work together in the process of encrypting and decrypting data.[16] In a scenario where each user holds his own private-public key pair a sender can encrypt its data by using another user's public key.[16] As the user's public key is linked to its private key by its generation the data can only be decrypted by it.[16] This scenario assumes that the private key is only known to him. In the Ethereum Blockchain each user is identified by a hashed version of a public key.[92] This key is used in order to link his messages to an account.[48] In Ethereum these keys are derived from a common representative of asymmetrical encryption known as "elliptic-curve cryptography".

2.2.3 Digital Signatures

Various problems in everyday life require signing a form. Procedures such as buying a house and signing a marital contract even require the presence of a notary. As such signatures serve a multiple purposes. However in the virtual world there is no equivalent to a handwritten signature. Instead a combination of asymmetric cryptography and hash values are used to both prove the connection between a private - public key pair and to link a signature to an integrity protected state of a document.[2] The signing process of a document or any other type of data works the following way: Let's assume that user A owns a private key (prK) and its corresponding public key (puK) and wants to sign a message.

- First he creates a hash-value of that message known as "Hash-Digest"
- Then he encrypts the hash-digest with his private key
- Finally he appends the encrypted hash-digest to the message itself

This serves the purpose of ensuring the integrity of a given document and also lets other users link the document to the signee's identity. Verifying the signature can be achieved by decrypting the hash digest with the signee's public key and then comparing the hash digest's value with a self generated hash value of the clear text message. If both hash values match the signee's identity can be inferred and the integrity of the message is confirmed. Digital signatures are appended to every message entering a Blockchain

network.[92] This way its sender can be verified and the message remains integrity protected. This is essential because Blockchains are based on Peer To Peer technology. If a member's message is potentially being redistributed to every other participant this could otherwise lead to possibly infinite manipulation attempts.

2.2.4 Securing HTTP

The previous chapter outlined HTTP and its requests. However there is one major shortcoming of it. It is being transmitted as plain text. Because Client-Server connections are rarely direct connections but instead rely on different relayers such as the Internet Service Provider (ISP) and other physical machines this circumstance poses a potential issue. The message can be intercepted at any point in between the intended receiver and the sender. This is also referred to as "Man-in-the-Middle"[95] attacks. In order to prevent this its counterpart "HTTPS" (Hypertext Transfer Protocol Secure) uses asymmetric cryptography. The server has a public-private key pair. Its public key is then being linked to certain attributes such as the server's domain and then being signed by a trusted authority.[95] This document is called "certificate".[95] Because the signature can't be forged without having access to the authority's private key this practice is being considered rather secure. Because these institutions provide certificates they are also called "Certification Authorities" or "Certifying Authorities"[95]. A customer explicitly needs to request certification for their servers. Depending on the type of certification it requires filling out multiple forms which are being proved by the authority. In case the customer provided valid information the authority will then send him a signed certificate. This way whenever a client sends a HTTPS request to the server it can answer with its certificate.[95] Because many clients browsers contain a list of trusted certificate authorities it can then decide whether it should establish a connection to the server or to show a warning message concerning the certificate's validity first.

Having a trusted entity within this system therefore allows trusting a server's provided public key. Because the certificate includes the server's public key a secure connection can then be established. This is achieved by using a technique known as "TLS"[43] (Transport Layer Security). During the establishment of a TLS connection multiple steps occur. It is initiated whenever a client requests a server's website via HTTPS. Because the server responds with its certificate the client can verify its validity.[80] Then both parties securely negotiate the TLS version they intend to use and a symmetric key algorithm to establish a "session key".[80] Because this communication is being secured by asymmetric cryptography the newly generated key is only known to the server and the client. Therefore it can then be used for further communication between the two parties. In addition to HTTPS being used for Client-Server communication it is also applied to communicate privately between nodes in the Quorum Blockchain[42].

2.3 Blockchains

Blockchains gathered a lot of public interest in the last decade. Especially a cryptocurrency called "Bitcoin" became increasingly popular. In fact its implications are so powerful that whole governments actively pursue its regulation in order to stop an uncontrolled flow of currency.[10]

Bitcoin's proposal was originally published in a white paper by Satoshi Nakamoto.[56] Until today his identity is still unconfirmed. However the paper led to multiple scientific implications and questions important parts of everyday life, including our understanding of the importance of banks.

As Blockchain technology is often referred to being a "distributed ledger"[59] the term "ledger" requires explanation first. In financial institutions such as PayPal the recording of their clients' balances is written to a structure known as "ledger". Essentially it can be regarded as a type of database listing all the clients' balances in a financial institution such as a bank. By handing over the management of one's money to a bank or another institution a client inherently places its trust in it.[98] More precisely in a scenario where there is only a one bank exercising control over a single ledger holding all the users' account balances the bank can alter the state arbitrarily.

In addition to the fact that said institution might have malicious intentions their systems often revolve around IT-based communication, i.e. programs. Naturally this exposes additional attack vectors to hackers. There are wide-ranging examples where placing trust into banks or other payment providers backfired due to the fact that their systems were not adequately secured and hacked by often unknown entities. As a consequence the user not only needs to trust the institution itself but also its systems.

The cryptocurrency Bitcoin and the platform Ethereum are developed with the intention of removing third parties (banks, etc.) from this equation by distributing trust.[98] Both projects are based on software applications which are maintained by the Open-Source community. This fact already indicates that the control over these systems is partially or fully handed over to the the public.

Similar to traditional payment providers both of these Blockchains prevent relying on a bank[18] which prevents the installment of a Single Point Of Failure by not relying on single ledger and instead distributing their ledger to a network of nodes within a Peer to Peer network. Basically each node then keeps its own copy and is therefore able to verify its validity.

Due to the nature of varying propagation speeds within Peer to Peer networks there are multiple competing ledgers at any point in time making it necessary to determine the ground truth, i.e. the "real" ledger within the network.[57]

In addition to the question about how to determine this "ground truth" the following sections answer questions about how a Blockchain reaches agreement on a shared state and how Ethereum is extending the characteristics of traditional crypto currencies like Bitcoin by storing executable programs known as "Smart Contracts"[77] on its chain.

2.3.1 Commonalities

As Bitcoin and other Blockchain networks are using Peer to Peer technology to pass messages between their nodes. In a public Blockchain like Bitcoin each node is able to distribute a message to the network. In the terminology of Bitcoin these messages are called "transactions".[64]

Thinking in terms of a currency Bitcoin is based around the idea that each participant's balance in a ledger can be constructed by a flow of currencies within a chain of transactions.[27] This is reflected by the way Bitcoin handles transactions. Each of them specifies a number of "inputs"[27] and "outputs"[27] . The inputs specify where the included amount of currency stems from while the outputs state its destination. In order to make Bitcoin's participants addressable they are identified by a public key known as their "address"[32]. The only exception to the necessity of declaring a transaction's inputs is when new currency is generated. The detailed workings of generating currency is described in the next section about mining. However there is still another potential issue. The message could've been forged by another participant in order to fake a user's intention to transfer his money. To prevent this each transaction is signed by the input's account to prove the owner's will.[56]

In addition to the transfer of the currencies BTC in Bitcoin and ETH in Ether transactions can also include a payload of data.[92][56] In order to keep the chain's total amount of data relatively low Ethereum links costs to the amount of data their transactions contain[55].

A "Block"[29] as in the name "Blockchain" is a data-structure responsible for aggregating multiple transactions.[56] It consists of a header and a body.[56] Both the structures of Ethereum and Bitcoin Blocks are similar. Therefore an explanation of Ethereum's Block is given by a header containing information about the the previous Block's hash value and the root hash of the Merkle Tree spanned by all its included transactions.[92] The Block's body contains the concrete representation of all its transactions making it rather big compared to the header itself.[92] By making use of hash-values and Merkle Trees an integrity protection of all the previous Blocks and transactions is given recursively. This mechanism is implemented to prevent a participant from cheating the system by forging a different chain of events or manipulating Blocks themselves in order to alter his or others' account balances. The motivation behind such an action could be to alter his own holdings in his favor.

The separation into header and body offers another advantage. While the full Block includes all its transactions the body only stores their Merkle root. By only storing a single hash value instead of all the transactions participants can still contribute to the stability of the network without storing all the transactions' data. While this might not seem relevant at first it is important to note that Bitcoin's full chain requires roughly 250GB of storage at the current date (Nov 2019).[11] This is an interesting fact as Bitcoin's official website states that a Block has a maximal size of [9]. Because a Block consists of multiple transactions it is safe to assume that each included transaction is less than 1MB in size. This already indicates that its intended purpose is not to store much data on the

chain. This is an understandable decision as transactions need to be replicable by any peer in the network. Thinking of storing high quality video footage on the Blockchain containing multiple gigabytes of data can therefore be considered bad practice.

2.3.2 Mining, Validation and Consensus

Because of propagation effects there are multiple competing chains at any given time.[99] Whenever a node considers a Block valid it is appended to the chain and distributed to connected nodes.[99] This process runs recursively leading to the independent verification of multiple nodes. The goal behind this approach is to prevent abuse of malevolent participants. Since whoever holds the power of convincing the majority of the network about appending a new Block is able to manipulate the global state of the network there are many possible competitors.[99] Whenever most of the network's participants consider the same chain to be valid the network reaches "consensus"[5]. The two most common consensus mechanisms are known as "Proof of Work"[5] and "Proof of Stake"[5]. Both operate under the assumption that the longest chain is the "real" chain.[99]

- **Proof of work** is based on guessing a "nonce"[56] (number only used once) in such a way that the new Block's hash value is ending with a specified amount of zeros.[56] Considering that a one way hash function is used there is no way to construct the required nonce instead of guessing. This procedure makes the inclusion of a new Block a difficult task. To alter the problem's difficulty the required format of the nonce can be altered.[56] In Bitcoin it contains an amount of zeros which can be adjusted.[56] After finding a correct nonce a node publishes its result to other nodes which can easily confirm its validity by checking whether the contained transactions are possible according to the state of their individual ledger and by confirming the nonce's format by calculating the Block's hash value themselves.[56]

  Because each guessing attempt requires computational effort the execution of the protocol is being additionally incentivized.[69] Each time a node publishes a Block it can append a transaction with his address to it which output includes Bitcoin without the requirement of being linked to a valid input. This leads to the issuance of currency also known as "mining". If the node's Block is propagating the network the fastest i.e. being successfully validated and redistributed by other multiple other nodes it becomes the network's ground truth due to the fact that it is being included in the network's longest chain. Due to the inclusion of the generation transaction the network then agrees that the miner owns the Block reward.

  The amount of financial value a Block Reward itself carries already hints the high amount of competition a mining node faces within the realm of Bitcoin. In fact, it serves as such a high incentive that there are mining services consisting of computer farms who distribute acquired Block Rewards to all its paying users.[99] Bitcoin is of such high relevance for regulated economies (e.g. the Republic Of China) try to impose restrictions on its execution. This poses a problem since

acquiring the majority of the network's computational power leads to having high influence on determining the validity of Blocks. This in turn leads to a centralization of the network. In this context oftentimes the term "51% attack" surfaces. Its name is based on the idea that an entity holding more than half of the network's computational power is able to manipulate the chain according to its liking. However there is research stating that this scenario might be more complex than previously assumed.

- **Proof of Stake** presents an alternative approach of finding consensus.[2] It is currently planned to be included in the upcoming Ethereum 2.0 release. Other than in Proof Of Work verifying nodes are called "validators"[5] instead of miners. Whenever a validator wants to participate in the process of Block inclusion it is required to "freeze" a certain amount of their Ether (ETH) known as "stake". Multiple validators then vote about the validity of the Block. The amount of ETH each of the nodes provide is considered during the decision process increasing the likelihood of a node successfully verifying the Block. In case the account successfully verifies a Block its balance is unlocked and the validator is rewarded by gaining a portion of the transaction fees. In Ethereum these fees are known as "gas"[55]. Other than in Proof of Work Ethereum's documentation states that the punishment for declaring wrong Blocks as valid is being "explicit"[69]. Therefore it provides mechanisms which removes funds from a misbehaving party.[69] It explains that this is necessary as Proof Of Stake does not provide any "implicit"[69] cost like the amount of required electricity[69] in a mining-based system. This statement already shows of its main benefits. As it is based around staking instead of raw computational power it is often considered to be a more sustainable alternative.

- **Raft-based consensus** is different to the other presented types of consensus due to its limitation on closed membership Blockchains (e.g. private / consortium chains). While a more detailed description of these Blockchain types is found in the next section intuitively reaching consensus on a predefined network of assumed-to-be trustful nodes is not as difficult as in public and open Blockchains. In Raft nodes are organized in "clusters"[70]. Each cluster contains a single elected "leader"[70] which is responsible for ensuring that all nodes share the same state.[70] To achieve this he accumulates data and distributes them in packages known as "log[s]"[70]. This makes the leader the only entity able to append new Blocks to the chain.[70] However there are possible reelections where a leader can lose its status.[70]

  Each other node in Raft assumes a role of either being a "verifier"[70] or a "learner"[70]. The leader continually sends its data to all the verifiers in order to prove its liveness and to distribute Blocks.[70] Each of them responds with a confirmation that they included the new Block in its chain.[70] If a verifier doesn't receive a message of the leader for a certain amount of time it starts a new election process and turns into a candidate for potential leadership.[70] This process allows

a verifier to become a new leader.[70] Both leaders and verifiers are able to add and remove verifiers and to promote learners to verifiers.[70] Learners are simply appending the Blocks of the leader to their own Blockchain.[70] Except the ability of removing themselves from the chain they have basically no additional rights. In order to actively participate in the process of forming the chain they have to wait for their promotion.[70] This imposes a hierarchical structure on the networks nodes. This is an important feature because Raft's consensus can be used by the Quorum Blockchain which is being an important part of the thesis proposed solution for issues regarding data privacy.

As the example of Raft-based consensus already indicated the determination of consensus is also strongly linked to the question of who is able to participate within the network. Thus the differences between private, public and permissioned Blockchains are being described in the following.

2.3.3 Private, Public and Permissioned Blockchains

The explanations in the previous sections were mostly based on the assumption that anyone can participate. While a cryptocurrency like Bitcoin benefits from such a property the understanding of who is able to participate (i.e. send and receive transactions) and who is hosting the nodes strongly differs in "private"[94] and "permissioned"[94] Blockchains.

- **Public Blockchains** are typically open for anyone to participate. Thus their main advantage lies in the amount of transparency they provide. This is the reason why there are public websites allowing a detailed examination of currently included Blocks. One of these websites is Etherscan.[30] It provides data about every Ethereum transaction ever occurred and can therefore be used as a valuable tool in data analysis. Because of this each of the transactions and Blocks can be inspected and checked for their validity. The previous chapters already listed two prominent examples of this archetype: Bitcoin and Ethereum are both being hosted publicly. As both these networks benefit by providing every member with publicly verifiable information about the history of transactions maximizing transparency is a reasonable choice. Intuitively guaranteeing a high degree of transparency counteracts a need for privacy. Their interdependence is a crucial part in the research area "Information Security". This is being explained in greater detail in later parts of the thesis. Shortly summarizing its principles: if every data is available to the public there are no secrets. Thus choosing a private system instead can provide benefits depending on one's use case.

- **Private Blockchains** as its name already indicated private Blockchains limit who is able to partcipate and who is not. Such a scenario might involve a single or multiple companies restricting their Blockchain and its participants to their employees or customers. Hosting one's own infrastructure oneself leads to responsibility in

choosing nodes. This fact often leads to a lot of confusion. One might argue that if a company selects its nodes by itself it doesn't necessarily require any consensus at all as it places inherent trust into its nodes. However there might be multiple departments within a company which don't necessarily trust each other.[94] Another scenario can be constructed by thinking of multiple non-trusting organizations trying to establish trans organizational trust without using a third party. Whenever multiple companies agree on using a Blockchain together they form a "consortium".

Both public and private Blockchains can be permissioned.[94] The act of setting permissions could include restricting the readability of transactions of different nodes within the system.[94] In addition nodes can be managed by being added or excluded from the validation process.[70]

In order to achieve a differentiation between participating and excluded nodes a reasonable requirement is for each of them to be uniquely identifiable. In Ethereum nodes can be identified by a combination of their IP address, their port and their public key.[19] After an initial establishment of a connection to nodes who are hard coded within Ethereum's source code each node proceeds to inform others about the existence about peers it is connected to.[19] Optionally a node can provide a file called "static-nodes.json"[19] which can be used to define a set of additional nodes to initially connect to. As it is JSON formatted it is structured as an array of nodes. An example of its structure can be found below.

```
1 [
2   "enode://publicKeyA@ipA:portA",
3   "enode://publicKeyB@ipB:portB"
4 ]
```

Listing 2.3: Structure static-nodes.json

This makes it possible for every computer wanting to participate in the network to join. A simple way of achieving this is to install the publicly available software "geth"[36]. It offers a command line interface which can be queried to gain insight about currently connected nodes or to send transactions to the network. The choice about whether to restrict users from joining a network can be regarded as a way of exercising access control by system design making it an important part of the modelling process. One Blockchain allowing a detailed specification of the different rights a node has is "Quorum"[42]. To achieve this Quorum extended geth's functionality by methods to dynamically add and remove nodes or to promote and demote them according to the principles described in the thesis' section about Raft consensus.[70] This fact already indicates that Quorum is originally based on Ethereum. Therefore both of these chains are explained in the following.

2.3.4 Ethereum & Quroum

While Bitcoin's primary goal lies in the financial world Ethereum extends this concept by the ability to store executable programs on its Blockchain. To make them addressable Ethereum introduces the concept of "accounts". It differentiates between "Externally Owned Accounts"[75] (EOA) and "Contract"[75] accounts. The former is being held and managed by users and their private key while the latter is not. In other words both types of accounts are assigned a public address while only EOAs hold a corresponding private key.

Smart contracts are stack-based programs which are stored on Ethereum.[92] Due to their turing completeness are very versatile.

However they are limited by their distributed nature. Since the Ethereum network requires a common state it is necessary for each node to execute the Smart Contract until the network reaches consensus about its execution. Every contract's byte code is executed in the nodes' "EVMs"[23] (Ethereum Virtual Machines). As a consequence the byte code encodes basic EVM operations to instruct the machine. In the public Ethereum chain every node receives a copy of its byte code as it is being stored on the Blockchain. Because of this both the contracts and the network's transactions are being known to all the participants. This way every node can confirm a contract's correct execution and results.

Each contract can be interfaced via transactions directed to its public address containing information such as the transaction's signature and the parameters. Similar to multiple servers communicating through their APIs contracts can interact with other contracts via messages. However the starting point for each alteration of the Blockchain's state is a transaction originating from an EOA.

The calling format of all a contract's functions are further specified by a contracts ABI ("Application Binary Interface")[34] which compilers such as Solidity automatically generate while building. As the execution of operations is costly Ethereum transactions include a fee as additional incentive for a miner to include it within a Block. It is called "gas"[55] and is being expressed by filling the transactions fields responsible for stating a gas price and start gas. The actual fee in Ether is then being derived from a multiplication of both these values plus an additional cost for every executed operation.[92] In case the account didn't has sufficient funds the transaction is fully reverted. Because the sender of a transaction can define the amount of gas he is willing to pay it can influence the likelihood of being included in a miner's block by providing more gas as a miner is assumed to be maximizing its profit and therefore prioritizing this Block over others with transactions who provide a lower amount of gas. This allows a sender to to influence the propagation speed of a transaction he sends.

Another reason for introducing gas is to link the execution of operations to a cost as it requires computational effort of the network's participants.[55]

Because a node can listen to all the transactions entering the system it can apply diverse reverse engineering techniques to dissect the network's flow of data.[1] Depending on different factors such as knowledge about the contract's address or its ABI these

magnitude of such an attack can vary. While this open and transparent interaction between EOAs, other EOAs and contracts ensures "public verifiability"[94] on the one hand it is being the core problem of ensuring "data privacy"[86] on the Blockchain. This is amplified by the fact that contracts often hold currency making them an interesting target for attacks. Thus it is important to include considerations about the privacy of a contract's data during the early phases of modelling and before deploying it on the chain.

While Ethereum can also be hosted privately in order protect sensitive data by reducing its exposure to the public a Blockchain known as "Quorum"[42] introduces additional privacy and node management features.

It can be run using the Raft consensus explained in the previous chapter. In addition to Quorum allowing the execution of any Ethereum Smart Contract it can process transactions privately.[42] While previously transactions were assumed to be publicly readable including all their details Quorum introduces a differentiation between private and public transactions. As it is being the subject of a later evaluation it is described in more detail in the thesis' chapter about Access Control and Previous Implementations.

3 Related Work

As a result of the initial literature research multiple related publications were found. All of them provided insights into the current state of access control inside or outside the Blockchain domain. As a consequence this chapter gives an overview about prexisting work in order to present the current state of the art.

"Using Blockchain to build decentralized access control in a peer-to-peer e-learning platform"[53]:
is a related Master's thesis implementing an Access Control system for handling resources in the context of an E-Learning platform for the aboriginal community. As it is one of the few related findings which include details on its implementation including an off-chain MySQL database as persistence layer, a REST-based API and GUIs for teachers and students it is mentioned in this chapter. Its approach is based on managing permissions by storing various flags similar to an Access Control List (ACL) in a JSON-File.

"Blockchain Based Access Control"[76]:
explains the differences between encoding access control as policies within transactions and storing them in off-chain repositories. It mentions the impact of storage limitations of Blockchains in the light of storing access control policies. While its policies are based on an XML language (XACML) its approach is to reduce storage costs minimizing the amount of information stored on the Blockchain. Instead of relying on Smart Contracts it uses transactions as a representation of access control policies. Each of them can be altered during the process of granting new or different rights to subjects while the Blockchain maintains a history of all occurred changes. Further the article explains the possibility of storing policies outside of the chain (e.g. databases). Instead of encoding the policies directly in a transaction's payload it then only carries a resource identifier and possibly its hash-value to ensure its integrity. Interestingly it even hints using torrents as means of distributing policies.

"RBAC-SC: Role-Based Access Control Using Smart Contract"[21]:
RBAC-SC is a Solidity based Access Control framework for Smart Contracts. It is published on GitHub and implements a basic role management. Its management is handled by an administrator which is assigned to the address of the user who initially deployed the contract. By using a combination of software patterns, events and mappings RBAC-SC can be understood as a database keeping track about all its registered users and their roles. By itself it doesn't implement functionality to determine whether a user is having a role or not. While such an extension is easy to implement its core

idea is not based around protecting a Smart Contracts functions similar to an REST-API but instead relies on the implementation of challenge-response-response based protocol which is being executed off-chain. This includes an linking a user's off-chain identity to the role has holds within the Smart Contract. In a resource exchange process a user is therefore required to proof his ownership of the public key linked to a role in order to be granted access to said resource.

"Secure Attribute-Based Signature Scheme With Multiple Authorities for Blockchain in Electronic Health Records Systems"[40]:
describes an encryption based approach on how to handle Electronical Health Records (EHR). Its goal is to allow fine-grained access control mechanisms for EHRs. Patients, hospitals, research institutions and other possible stakeholders are involved in a system of the assignment and possession of attributes which allows a participant (e.g. doctor or patient) to request and store data via a set of different signatures each of which serve as a proof of attributes similar to attribute based encryption schemes:

```
((cardiopath) AND (disease period more than 10 years))
OR (((Harvard professor) OR (Yale professor))
AND (Expert on cardiopathy)))
```

In addition the article outlines the possibility of each of the parties participating in a consortium Blockchain. Each Block in the proposed system represents a patient's treatment and includes additional information such as the state of his insurance. This system tries to expand on the traditional health-care system in which a patient's EHRs are kept by only a few health providers and doctors which are the only entities capable of managing the database holding all the health records.

MedRec: Using Blockchain for Medical Data Access and Permission Management[4]:
presents a different approach for solving the issue of EHRs. It makes use of private Blockchains synchronisation between off-chain provider databases and Smart Contracts. Its concept is based on deploying one Smart Contract for each provider keeping a patient's records. This "Patient Provider Relationship" (PPR) contract keeps information about how to connect to the corresponding database (e.g. port + hostname) and encodes permissions by storing allowed SQL queries. The EHRs are therefore kept on the providers' databases. Its advantages are an easier adaptation of existing systems over Blockchain-only solutions. The PPR additionally encodes viewership permissions as SQL queries, leading to a lot of flexibility. The provider both maintains and executes the available SQL queries. Whenever a patient wants to grant a third party access to his health records he is able to access a website offered by MedRec where he answers different questions about what information should be accessible. MedRec then generates a query string based on the given criteria and stores it in on the Blockchain linked with the third party's public key. The system provides an off-chain access interface for each providers' database which requires a signed request to link a person's off-chain identity with his permissions in a smart contract.

"Security and Privacy on Blockchain"[97]:
is a summary of privacy preserving techniques linked to Blockchain technology. Due to the transparency of Blockchain data there are multiple approaches to encrypt and or handle data. It proposes the usage of advanced technologies such as homomorphic encryption, Zero-Knowledge Proofs, attribute-based encryption, secure multiparty computation and ring signatures. As encryption can enforce confidentiality it can be used as another way of enforcing access rights by design choice.

"FairAccess: a new Blockchain-based access control framework for the Internet of Things"[64]:
is a widely cited framework designed around its own access control protocol. Resources, their owners and requesters are described as potentially residing in different organizations. In order to achieve a high degree of flexibility and preserve the organizations' autonomy over their access control decisions, FairAccess defines its own access control protocol. Each entity (i.e. resources, requesters and resource owners) are identified by their address. A resource owner generates and maintains potentially multiple trees of derived addresses to its resources using a common private key as generator. By doing so identifiers can potentially be reconstructed in cases of data loss. The actual granting and delegation of access rights is handled by including an access token. Whenever a resource owner intends to grant another user access to one of his resources he encodes a token into a transaction and publishes it to the network. The token itself is encrypted with the public key of the requester in order to ensure confidentiality between the two parties. Alongside the token the transaction includes the public addresses of the requester, the resource and instructions about requirements a requester needs to fulfill to spend said token. During the actual request phase a user then includes a proof of his rights alongside a "GetAccess" transaction. In addition, its future work section FairAccess gives an outlook on concrete storage proposals.
"On or Off the Blockchain? Insights on Off-Chaining Computation and Data"[29]
includes insights of previous research about how to preserve important Blockchain properties such as integrity protection and transparency while evaluating different degrees of off-chaining. It includes a list of different design patterns for Smart Contracts and scenarios of when to apply them. Patterns ranging from "Content-Addressable Storage" to "Challenge-Response" find many different use cases and are therefore found in many parts of this thesis. The gathered observations play an important role in the decision process of how to implement various strategies regarding data privacy.

4 Access Control and Previous Implementations

While the previous chapter provided an overview about different approaches of the past the thesis expands on them. While previous research often stated the relevancy of permissioned and private chains there are full-featured implementations are rare. The thesis on the other hand tries to accommodate them right in the beginning of the prototype's designing phase. As a consequence it includes implemented variants of its Smart Contracts which are designed explicitly in order to be run on Quorum. At the same time the thesis system tries to expand on previous projects while still maintaining its capability of being deployed to a public Blockchain. Because of these goals this chapter first defines the steps involved in the evolution of different access control systems and then proceeds by explaining related terminology and concepts. Understanding them is necessary in order to be able to follow the chapter's evaluation of existing approaches. Before each individual analysis related topics are explained. Because two of the evaluations require some basic knowledge of the programming language Solidity the chapter also includes a short description of its functionality. In addition it describes how the private Blockchain Quorum can be used in order to ensure privacy on Blockchains. The chapter then concludes by explaining an approach based on the "XACML"[54] language and architecture and gives examples of possible shortcomings of the existing solutions in order to formulate system requirements for an own implementation.

Authorization revolves around the basic question "who" is able to do "what". As simply as this might seem it dictates a careful assessment and determination of involved actors and their roles including who is able to read and alter data. Thus it is strongly linked to the three fundamental goals of Information Security: confidentiality, integrity and availability.[66] In literature these are also being referred to as "CIA Triad"[66]. Because these data properties are interdependent maximizing one factor can lead to a decrease of another one.[66] As an example: one can argue that by ensuring a high amount of confidentiality its availability is reduced. Differently stated: If only a single person within a whole organisation knows a secret its confidentiality is high while its availability very low. In terms of Blockchain technology the factors of integrity and availability are maximized due to its distribution of transparent transaction data. In addition it is often being argued that its confidentiality property due to its transactions being publicly readable.

Not fulfilling requirements such as integrity or confidentiality by not adequately modelling a system's flow of information may lead to security breaches or data loss

which can strongly impact the public image of a company leading to economical damage and potential lawsuits. For this reason several economic branches evolved. There are certification companies which a a company can consult to prove its trustfulness and insurances which cover the damages caused by data breaches. Even the previously mentioned Certificate Authorities can be regarded as being part of this business model. This showcases the amount of money their customers are willing to pay in order to prove their integrity. Oftentimes fulfilling the necessary requirements for achieving a certification enforce the definition and establishment of strict rules. In order to prove their trustworthiness companies are trying to fulfill their requirements despite the additional workload their implementation poses and the implications they might have on its hierarchical structure.

4.1 Types and Terminology

Access control by itself is no new discipline. Instead it evolved over several decades and is being found in a multitude of different projects. It is in no way limited to the domain of web technology. Instead it is being used by electrical devices just as much. One common example is the "fingerprint sensor"[78] which is often integrated into modern smart phones. During the evolution of the discipline itself the development of its terminology followed. This makes it necessary to give a basic overview of the existing types of access control. Because Access Control is subdivided into three major types it makes sense to start by giving an explanation of each of them first.

- **Discretionary Access Control (DAC)**[44] In DAC each resource has an owner who is responsible for managing other subjects' rights to it. One example of widespread usage of DAC are Linux systems. In Unix each user who creates a file has the option to grant others access to it. Its permissions are defined in a so called "Access Control List"[44]. Such a list provides information about which operations can be carried out on a resource. In Linux each file's permissions are expressed by three values of the size of a single byte called a "flag". A can then be defined by a binary addition of the following numbers:

  ```
  100 Read    (r) 010 Write   (w) 001 Execute (x)
  ```

  The access control list of a given file in Linux can be viewed by a calling "getfacl" followed by the file's path. This is being shown in the following example.

```
1 $ getfacl tum.pdf
2 # file: tum.pdf
3 # owner: thomas
4 # group: users
5 user::rw-
6 group::r--
```

```
7 other::r--
```

Listing 4.1: Linux Access Control List

The code listing shows that the file's owner is called "thomas" and that he currently holds read and write rights (rw) identified by the binary number "110". Each file's permissions can be altered via the "chmod" command followed by three decimal numbers representing the binary number converted to its decimal representation. These three numbers indicate the permissions the file's owner, the group called "users" and all others have. The example would therefore indicate a permission flag of 644. In Linux an owner can even revoke his own permissions to alter the given file. To prevent inaccessible files every administrator in Linux (i.e. root user) can change the permission of a file nevertheless. Such emergency mechanisms can be used in order to prevent user-side errors.

- **Mandatory Access Control (MAC)** is based on the concept that each subject and resource are assigned security levels.[44] Depending on the type of MAC a user with a lower security level than a resource is not able to write said resource but is still able to read it. Similar to the permissions an Access Control List grants, the security levels are responsible for the determination of whether a subject is able to read and / or write a resource. The decision on whether to allow or deny such a request is based on the concepts such as "No-read-up"[12] and "No-write-down"[12]. No-read-up says that a resource can only be read by a subject if the subject's security level is higher than the one of the resource.[12] No-write-down states that the security level of a subject needs to be less than the resource's in order to alter it.[12] Due to the strictness of permission assignment and revocation, literature defines MAC's main areas of applications in sectors requiring high amounts of confidentiality (e.g. military sector). These extreme security considerations can lead to problems with the originally intended transparency of Blockchain's making MAC a rather uncommon option in Ethereum.

- **Role-based Access Control (RBAC)** depends on grouping subjects by their roles.[44] The example of a Linux Access Control List already made use of a very basic types of roles by including groups into their definition. This shows that RBAC by itself can be implemented based on DAC principles. A user might recognize RBAC from its every day computer usage. To give an example a user can log into a computer by either using the administrator account or by using his own account. In contrast to MAC there are preexisting implementations of RBAC Smart Contracts to date. Two examples of them are: *RBAC-SC* and *Openzeppelin's RBAC*. Both of them are evaluated at a later stage of this chapter. To give a more business related example of applied RBAC an employee of a company might hold multiple roles at the same time. Therefore he can both be a general employee and book keeper. Each of these roles is linked to different rights. This allows him to change between the roles depending on the requirements of his current task. The concrete definition and

modelling of purposeful roles is by itself, a demanding task. This is underlined by the fact that there might be conflicting roles and even hierarchical role structures which inherit a parent role's rights. One of its main advantages lies in its ability to provide an easy mapping from real world entities to their respective roles within an RBAC system. However depending on the complexity of the underlying Access Control model this can quickly evolve to a rather complex task.

In addition to the aforementioned terminology there are different reoccurring patterns in the domain of access control. As the popular Solidity framework "OpenZeppelin" includes an implementation a "Whitelist" strategy it is being explained in the following including its counterpart the "Blacklist".

- **Blacklists** Blacklisting refers to the act of allowing anyone to access a resource except for users which are explicitly excluded. Figuratively one could say that these individuals are therefore being part of a so called "Blacklist". Due to its openness it is generally considered to be rather unsafe and not applicable in most business scenarios due to its lack of confidentiality. Using Blacklists based on Ethereum account's address is essentially useless as a subject can easily generate a new key pair and account.

- **Whitelists** are based on the opposite approach of Blacklists and deny access by default. Only users, which are included on a list called "Whitelist" are granted access. In its extreme case a Whitelist can only define a single subject as being allowed to manage a resource. This is essentially done whenever an administrator is the only entity within a system which is allowed to interact with a resource.

All of the previous explanation are being rather concrete in the way they are being implemented. As access control does not necessarily imply limiting oneself to rather primitive means such as Access Control Lists another technique known as "Chinese Wall" provides a more abstract stance on the topic of exercising access control.

"Chinese Wall"[14] tries to combine the advantages of DAC and MAC approaches by grouping different users, resources and companies into "conflict of interest"[14] classes. Chinese Wall therefore operates on a higher level of abstraction. Its original model revolves around companies and their data.[14] For this reason it groups data into "company datasets"[14]. It then explicitly states that a subject is only allowed to access another object if it has already accessed an object from the "same company datasets"[14] or if it belongs to "an entirely different conflict of interest class"[14]. To give a basic example:

An employee works for the content provider network "Netflix". As his everyday job is to provide subtitles to the series the network provides he is directly interacting with the company's datasets. As its competitor "Hulu" can be regarded as being within the same "conflict of interest" class the employee is not allowed to provide its subtitles to Hulu. If he instead shares it with the company responsible for producing the series in order to

ask for corrections he doesn't surpass any the "Chinese Wall" by not leaving his own class of "conflict of interest".

This model provides an intuitive explanation on what companies try to achieve. Many of them try to guard their secrets from their competitors as they can provide valuable insights to them. A company which has a unique way of producing a product could try to take multiple measurements in order to preserve its secret. Therefore one of the main goals of access control is often stated as limiting the "information flow"[46]. In order to understand how to limit this flow of information within a publicly readable Blockchain like Ethereum it is therefore necessary to first analyse existing solutions from the domain of client-server architectures.

4.2 Access Control in Client-Server Architectures

While many of the existing implementations of access control systems in Ethereum are rather basic there are many well established frameworks which are targeting web servers and their applications. Consequentially multiple strategies evolved. By first defining them it is possible to derive conclusions and best practices which can serve as a basis for transferring the gathered knowledge to a Blockchain system. Thus the following paragraphs introduce the two fundamental and universally applicable concepts of "authentication" and "authorization" by explaining the way they are interpreted in traditional Client-Server architectures.

4.2.1 Basic Authentication

According to the definition of "authentication" it is "an act [...] of showing something [...] to be genuine"[3]. In the context of web applications it is used to prove one's identity[43]. TLS already provided a method for proving a server's identity by the signature of a trusted third party.[43] Clients on the other side often don't go through such certification processes.

Single & Multifactor Authentication

Instead many web applications make use of simple password based authentication for identifying their clients. Whenever a user accesses a website he provides his username and his password. The data provider can then look up fitting entries within its user storage to access additional information. This could include personal user data such as his last name or his individual preferences for viewing the website.

To confirm the validity of the transmitted password the database holder compares its hash value to the ones stored in his database. This is common practice as storing hash representations instead of actual passwords adds another layer of security in case of a successful attack.

The initial set up of the password might have been part of a user's registration at the website or subsequent edits of his account's data. In order to ensure that these

credentials are transmitted securely an HTTPS connection can be used. The necessity of ensuring their confidentiality is is especially important in the context of Smart Contracts. If a transaction includes the plain-text version of a password it can be read by every participant in a public chain. Consequentially later sections of this chapter include an explanation on how Quorum makes use of HTTPS in order to solve this issue.

This type of authentication can be categorized as being a method of "Single Factor Authentication"[47] as it only requires the knowledge of a single secret. Its counterpart "Multi-Factor Authentication"[47] requires the knowledge of two or more secrets to improve security. It is often used by banks when they request a TAN alongside a username and a password during online banking sessions.

After a successful authentication the server can respond by opening a so called "session" encoding a unique identifier for the current connection between him and the client. This way the client can provide its identity by sending his session id instead of its password in subsequent requests. To simplify its management its possible to store the session ID within the client's browser.

Challenge Response Authentication

Another way of authentication is known as "Challenge Reponse Authentication"[67]. It requires the client prove his knowledge of a "shared secret"[67] to the server without exposing its content by transmitting it. This secret could be a password only the server and the client know. One possible way such an authentication can be executed by to leveraging hash functions. In case the client knows the secret he uses a one-way hash function to generate its hash representation. Afterwards he can proceed to send it to the server. The server is then able to confirm whether the client has successfully answered his challenge by generating a hash value of the secret as well. If the comparison of both the hash values is proving their equality the challenge is being fulfilled. As a consequence the client is being authenticated by the server.

Single Sign On

All these scenarios can be extended by adding more than one server to the scenario. Because resources are not necessarily stored on a single server but multiple machines instead accessing them would require a user to provide his credentials each time he interacts with a new server.[24] This problem is solved by "Single Sign On"[24] systems who rely on a central authentication server which reigns over multiple machines including their resources.[24] After a successful authentication the server responds with a "token"[24] which can be by a user to authenticate with all the machines. The storage of the credentials is handled by a so called "authentication authority"[24]. The fact that they are also referred to as "Trusted Third Parties" already indicates that the domain's participants who the actual resource data need to be able to trust the entity responsible for the management and validation of user credentials. This is why related literature mentions that a successful attack on this system grants an attacker access to

the complete domain. In other words the system relies on a strong centralization of trust.

In the context of Blockchains authentication strategies can range from simply owning a private key linked to a public address to more complex ones involving proving ones knowledge of a shared secret ("zk-SNARKs"[35]). In order to implement a full access control system it is necessary to consider the transparency involved in the process of authenticating a user. This makes it necessary to implement authentication strategies which don't rely on the transmission of secret data. One secure proposal of an authentication model could be modelled the following way: A Smart Contracts initializes itself during its deployment by storing the address of its deployer. During its initialization phase it requires a second parameter which is being an address of a second user. After both arguments are provided the contract proceeds to store both users as registered members. It then provides the functionality to vote on new members which request registration. In case at least two registered members confirm its registration the new user is being added. While this strategy is rather simple it does not rely on any secret data being shared.

4.2.2 Authorization

After the user is authenticated it is necessary to determine his permissions. This is the subject of authorization. It tries to answer the question "who has the right to do what?". One everyday example of applied authorization is found in the management of WhatsApp groups. Each group has a single or multiple administrators who are able to invite new members. Other non-administrators are not able to do so. While the impact of deciding who administrates such a group might be negligible, protecting sensitive customer data is not.

Over the years multiple forms of Access Control have emerged. Common terms in the context of authorization are: "Subject", "Object" and "Action"[44]. The previous examples already included a possible object when they introduced an API's resources. Restating the previous question in access control terminology leads to the assessment of whether a subject (e.g. a user) has the right to perform an action on a resource. These questions are being the subject of "authorization".

OAuth2

One common framework in the context of authorization is "OAuth"[90]. It is being publicly available and allows a user to authorize a third party API which is referred to as "Client"[90] to access another API which acts as a "Resource Server"[90]. It should be noted that this terminology is based on OAuth's variation called "Authentication Code Flow"[90]. The user is being responsible for deciding whether to grant a client access to his resources or not. Therefore he is also being called "Resource Owner".[90] This is achieved by using a central "Authorization Server"[90]. The resource server can potentially keep any type of information. Therefore the resource server could include

personal information about the user such as the user's first and last name or different photos and videos he stored. The authorization server provides an interface for clients trying to access the a user's resources.

Summarizing the steps described by OAuth's documentation[90] its process can be described in the following steps:

- A client requires user authorization to access one of its resources
- Therefore it redirects the user to a website hosted by the authorization server. The website can be regarded as an interface.
- The user is then being prompted for its confirmation
- After the client's confirmation the server redirects him back to the client's application
- This redirect results in an "Access Token"[90] for the Client
- The client can then use the "Access Token"[90] for subsequent requests to the resource server

In more detail: To be able to communicate which actions a client wants to use the request defines a "scope"[**oauthscope**]. The scope could be set to "update" and "create" indicating that the third party application intends to use these types of operations. However depending on the use case its possible to define own scopes. The authorization server's website can then warn the user about the intentions such an application might have and about the possible consequences such an authorization. As a reminder the create and delete operations have already been explained in the thesis' chapter about HTTP Requests and APIs.

In addition the request contains a URL to which the user is being redirected after finishing his authorization.[31] Because this URL can be set to an endpoint of the client it can be used in order to send it an access token which it can use in future requests.[31]

To give an example of applied OAuth one can imagine a third party Smartphone application which tries to post photos to a user's Instagram timeline. As it can be assumed that the application is not automatically able to interact with Instagram due to missing access rights the application redirects the user to the authorization server hosted by Instagram. In its request it encodes the scope of the operation as "createPhotos" as the application's only intention is to post new photos. Instagram then informs the user that a new application is asking to post new photos to his timeline. In case he accepts an access token is generated. This token is then being transferred back to the third party application which it can use in order to post photos by interacting with Instagram's API functionality.

XACML

Each company is under the influence of laws, competitors and its customer's desires. To smartly navigate along this frame a company formulates policies indicating rules of

conduct for their employees. Thus concepts such as guidelines and rules a are commonly established. These rules are often formulated as "company policies"[87]. They may be as simple as: "a developer does not need to know the income of the company's employees except his own.". Its employees are then being expected to follow them and to keep informed about policy changes. This is ensured by sometimes long lasting compliance seminars. This underlines the amount of importance placed into their application. A policy is technically useless without its implementation. In other words if no employee follows it, it has no effect on its own. This separation between policy formulation and its "enforcement"[26] is also found in the context of access control.

As access control can be expressed as a set of rules it can be understood as a language including syntax and semantics.[26] Approaching the limitation of information flow at this level of abstraction allows a greater flexibility compared to hard coding restriction mechanisms. concrete implementation of restricting code. By combining multiple policies different security concepts can be established. Such constructs can range from simple blacklists to more complex structures. One common problem in formulating policies is to handle conflicts in case they contradict each other. Residing in the realm of compliance, company policies and access control policies there is a process known as certification. Different trusted institutions offer a so called "certification" granting its owner a presentable indication of quality. It can be used to attract customers and positively reflects a company's image. Certifications are therefore often required to establish a company in the software consulting area or other areas, which require a lot of trust (e.g. banking sector). Because the management of these policies is can be complex many companies use special systems to manage their Access Control System. Alongside the ability to add and enforce new policies and rules it includes monitoring and emergency features in order to audit and handle historical or upcoming requests to a company's data. Because the formulation of policies is a historical problem there are already existing approaches.

The most common one is known as "XACML"[13] (eXtensible Access Control Markup)and presents an both an architecture and a XML-Schema based language for managing and processing access control policies. Due to its flexibility it can model multiple different types of access control.[13] It is being standardized and maintained by the OASIS consortium and provides functionality for the handling and management of requests according to formulated policies.[13]

XACML defines a system of multiple components called "points"[13]. Points define an infrastructure sequentially handling an incoming access request originating from a "subject"[13] with the intention to access an "object"[13]. Each point is responsible for a different step within the process of granting or denying said request by reaching a decision. The conclusion is derived from on a set of policies defined in a Policy Repository.

Policies in XACML are grouped into "PolicySets"[13] which allows an easier management. Each of the policies can include rules. Each of the rules states an effect and targets. Its targets are required to determine whether a given rule needs to be applied. In

addition to an explicit statement of the subject, the user and the action the target refers to XACML provides operators such as "anyOf"[38] or "allOf"[38]. A simple example of an XACML policy can be found below.

```
<Policy>
    <Rule Effect="Permit">
        <Target>
            <Subject "Thomas" />
            <Resource "Movie: Drive" />
            <Action "READ" />
        </Target>
    </Rule>
</Policy>
```

Listing 4.2: Example of a XACML Policy

The code sample states that a user called "Thomas" is allowed to read the resource "Movie: Drive". Including this policy inside an XACML repository results in a successful request whenever the user tries to "read" the movie file. In other words: The user can watch the movie. While the given example is rather simple XACML supports a wide range of operations. In order to achieve this XACML provides a complete architecture for processing incoming requests. To give an intuitive example of how an incoming request is handled the following scenario can be constructed: A user (subject) is trying to watch (request) a movie (object) on Netflix. In order to receive file his request is processed in the following way (based on the information included in XACML's specification[38]:

- A user authenticates himself - this is done outside of XACML
- The authenticated user sends a request to the platform's Policy Enforcement Point containing the URI of the movie.
- The Decision Point currently only holds the information contained in the request itself (e.g. User: "Thomas", Movie: "Drive")
- The Decision Point queries a Policy Repository to gather related policies.
- The Decision Point queries linked Information Points to gather contextual information (such as the user's subscription status)
- The Decision point aggregates all the information he is holding
- As the user is successfully authenticated and has an active subscription status the evaluation of the policies yield a positive result.
- Consequently the Decision Point informs the Enforcement Point about its positive decision
- The Information Point then retrieves the movie from its attached file storage and transmits it to the user

This way XACML can enrich the evaluation of policies by contextual information and provides an abstraction layer on top of subjects, objects and resources. It provides way more flexibility than a simple system like the previously explained Smart Contracts based on RBAC. During literature research a single project was found which tried to fully implement a Smart Contract based XACML system.

4.3 Evaluation of Access Control Systems in Blockchains

After outlining the applied strategies of authentication and authorization within Client-Server architectures it is necessary to evaluate the current state of Blockchain based systems. Thus this chapter includes a description of different approaches related to the concept of access control. It starts by explaining how Quorum implements leverages HTTPS in order to send private data.

4.3.1 Data Privacy & Quorum

In Quorum each node contains a system called "Tessera"[42]. It extends it by a component known as "Transaction Manager"[42]. It is able to communicate with the other nodes' managers via HTTPS.[42] This way it is possible for them to exchange confidential messages without the necessity of exposing this confidential data on the Blockchain. Quorum's documentation contains a list of multiple extensions it provides to the way Ethereum's Blockchain model works. One of its main addition however lies in its capability of sending private transactions and Smart Contracts.[42] In order to understand the way this was implemented its process can be summarized as shown in the list of steps below. The following explanations are therefore being a simplification of the original list provided by Quorum[42].

- As a Quorum node receives a transaction including a field "privateFor". Therefore the node knows that this transaction has to be processed privately.
- The field's content corresponds to the public keys of all the intended recipients' transaction managers.
- Each time a node wants to send private data it sends the original transaction to its Transaction Manager which redirects it to a component called "Enclave"[42]. Its responsibility lies in the encryption and decryption of the transaction's payload.
- The Enclave answers with an encrypted version of the transaction's payload containing different keys which ensure that only the intended recipients are able to decrypt it.
- The Transaction Manager then proceeds to distribute the Enclave's encrypted response to all the Transaction Managers listed in the "privateFor" field. This is being carried out via HTTPS.

- After it is being received each Transaction Manager stores it in a storage which uses the data's hash value as an index.
- The sender's node then waits for the acknowledgement of a successful transfer.
- It then proceeds by replacing the transaction's payload by its encrypted representation and setting the transaction's "V"-field value to "37 or 38" to mark it as a private transaction.
- Then it publishes the transaction to the Blockchain itself. Consequentially every node within the network knows about its existence without being automatically be able to read its content.
- Each node in the network will then query its Transaction Manager for the transaction's hash index to find out whether it has access to its contained data.
- In case it does not it receives a "NotARecipient" message as an answer. Otherwise its Enclave is able to use the Transaction Manager's private key to decrypt the keys necessary for decrypting the payload itself.
- The final step after successfully decrypting the payload is to send its unencrypted data to its EVM to update its state accordingly.

Evaluation The described approach provides a solution to the issue of sending private data through a Blockchain. While it additionally introduces the overhead of requiring the installment of a transaction manager Quorum also offers the capability of deploying private Smart Contracts between participants. Each of these contracts is being protected by not sharing its state with excluded participants. Regarding this subject the documentation explicitly states that the "execution will update the state in the Quorum Node's Private StateDB only."[42] Therefore other contracts don't know about the state of the node's contract. While on the one hand this makes it impossible to reach network consensus over its data it can be leveraged to ensure its confidentiality. In other words this approach limits the public verifiability of its execution in order to privately communicate.

However one could argue that many scenarios such as sharing Medical Records strongly favor privacy related features over being part of a consensus process.

As Quorum allows the deployment of arbitrary Smart Contracts it allows the implementation of an access control system which can both be used for private and public scenarios. The thesis therefore decided to provide an access control system which additionally includes Smart Contracts which are intended for being used within a Quorum environment. This serves as an extension to previous research by offering a new perspective to the on-going debate of data privacy on Blockchains. In addition to the privacy features themselves Quorum also introduces a complex permissioning system which is entirely based on the implementation of Smart Contracts. Therefore a synthesis between the proposed system and existing functionality is imaginable. Further

each account can be assigned a role which it holds within an organization reminding of the Chinese Wall principle described earlier.

4.3.2 Access Restriction via Smart Contracts

In the past there have been various examples which don't make use of Smart Contracts within their access control system. One example of such an approach can be found in "Blockchain Based Access Control"[25]. As the thesis goal is to specifically determine Access Control mechanisms in the Ethereum environment limiting its approaches to pure transaction-based access control is ignored in favor of Smart-Contracts implementing AC functionality. This decision is based on a crucial implication: Both Quorum and Ethereum include contracts. Therefore the developed mechanisms can be used in applications based on either chain. As Quorum introduces private transactions[42] it can then be used in conjunction with the implemented system to restrict access even further via exercising control through Raft's mechanisms or via Quorum's permissioning.

Basic Programming in Solidity

As this chapter includes detailed analyses of reference code it requires some basic understanding of programming in Solidity. Thus this chapter begins with a short summary of its basic functionality. Other than object-oriented programming languages like Java or C++, Solidity is based around "contracts"[20]. Each source files provides the ability to import other source files via an import statement.[20] This allows the inclusion of libraries.[20] The complete chapter serves as a summary of the information provided by Solidity's official documentation.[20]

Variables Variable assignments require an explicit declaration of their type. Similar to C++, it includes types such as integers, strings and booleans. In addition Solidity provides its own bytes and address types. Thus every declaration of integers and bytes can be followed by a number. In case of integers this indicates its required bits ranging from 8 to 256. If the variable was of type bytes instead the number needs to be multiplied by 8. A bytes32 variable therefore has the same length as a int256 value. In addition integers can be marked as unsigned by adding the letter "u" in front of their declaration as in "uint". Each of these primitive types can be used in conjunction with arrays or mappings. A mapping links a key to a value. Thus they are being expressed by the notation (typeA => typeB). By default mappings initialize all values to the zero representation of typeB. A mapping of type (address => uint) would therefore result in a 0 value for every address unless explicitly set otherwise.

Functions can include multiple operations. A full declaration includes parameters and the data type of the return value. They can be declared as "pure" in case they don't read or modify state variables or "view" if they are only reading from state without writing it.

Contracts consist of multiple functions and and variables. This allows them to represent state. In addition they can include an optional constructor which is similar to the ones known from other programming languages. Aggregating the previously explained concepts in a single contract yields the following example:

```
1  pragma solidity ^0.5.0;
2  contract ExampleContract {
3      uint callCount = 0;
4      // mapping from address to uint
5      mapping(address => uint) public balances;
6
7      // array of size ten of 8 bit unsigned integers
8      uint8[10] unusedArray;
9
10     // Sets value of callcount to 1 during initialization
11     constructor () public { callCount = 1; }
12
13     // msg.sender is a constant value from the transaction's sender
14     function changeBalance(uint balance) public {
15         balances[msg.sender] = balance;
16     }
17
18     // pure function
19     function unsignedMultiply(uint a, uint b) public pure returns (uint) {
20         return a*b;
21     }
22 }
```

Listing 4.3: Implementation of a Contract

Sidenote: In line 15 the array is indexed by the constant "msg.sender". It is being provided by Solidity and corresponds to the transaction's sender identified by its address at runtime.

Inheritance Just like C++ allows inheriting from parent classes contracts can be inherited from. This allows overriding functions and basic polymorphism.

Structs can be used to group the combination of multiple types under one name. Currently it is not possible to declare structs recursively by self-referencing. Still structs are often used in order to build data-structures like linked lists. This is a common workaround found within implementations. The reason for their existence lies in the fact that Solidity offers no concept like C++ pointers.

```
1
2        ListEntry[] entries;
3        struct ListEntry {
4            uint index;
5            int next;
```

```
6       }
```

Listing 4.4: Implementation of a Struct

The code shown above introduces an array keeping all the list's entries. Whenever a new ListEntry is appended its index can be set to the current length of the "entries" array. Setting the "next" value of an element therefore refers to the referenced element's index within the array. To introduce an entry as last element its next value can be set to a negative value such as "-1".

Events can be used in order to notify clients that a state change occurred. Their definition can include multiple parameters of basic types, making it possible to add more information to the notification itself. Based on the previous example of a Smart Contract featuring lists one idea would be to extend its functionality by emitting an event whenever an entry was appended. Its event's parameters could then include information such as the index it was stored to.

Exceptions are thrown when one of the following statements fail: assert, require, revert. All of them include parameters.

Function Modifiers A modifier is used to to annotate functions. Thus their definition requires the inclusion of a "_:" statement. This part of the code is then being is replaced with the annotated function's body. Using them in conjunction with throwing an exception is often being used in order to stop the execution of a function.

```
1 modifier annotatedFunctionIsNeverExecuted(address irrelevantParameter){
2     require(1 == 0);
3     \_; // this is being replaced by the actual functions body
4 }
```

Listing 4.5: Implementation of a Modifier

This technique is often used in the context of access control and part of Solidity's official documentation[20].

As the basics of implementing Smart Contracts have now been described it is now possible to analyse existing implementations of Access Control within Solidity. During the next few sections the library "OpenZeppelin"[62] and another project known as "RBAC-SC"[21] are being evaluated.

OpenZeppelin

OpenZeppellin is an MIT-licensed library hosted on GitHub with more than 200 contributors. It calls itself "a library for secure smart contract development"[62]. It implements different security features including basic RBAC functionality. As it is a library its main focus lies on being applicable in multiple different projects by extracting common pieces of code and presenting them in a reusable fashion.[62] Because it is hosted on

GitHub the open-source community is able to actively participate in its development. An important addition to the common understanding of how traditional libraries work in languages such as C++ is that in a distributed network such as Ethereum a chain only needs to persist the compiled library once making its storage more effective and providing functionality for the whole network.

OpenZeppelin includes the implementation of two contracts regarding access control:

Ownable.sol is a contract included in OpenZeppelin. It includes a state variable called "owner" which is assigned within the contract's constructor. It is being initialized with the constant value "msg.sender" which refers to the address of the transaction's sender. It then introduces a modifier "onlyOwner" which limits the ability to call an annotated function to the owner himself. Therefore it can be regarded as a Whitelist containing only a single member. In addition the contract includes events whenever changes of ownership occur. Initiating a transfer of ownership is being protected by the onlyOwner modifier and takes another address as a parameter. Alongside the ability to transfer ownership the contract provides functionality to unset its owner. This is being achieved by setting the state variable's value to address(0). In Solidity this can be done to remove an as it is being considered extremely unlikely to generate the private key corresponding to the null address. The code's comments explicitly state that this is a permanent decision. As a consequence it states that setting the owner variable to null will effectively disable every function which makes use of the function modifier. The revocation of ownership is restricted by an "onlyOwner" modifier itself as a means of protection. There are only two non-restricted and publicly accessible functions. One of them returns the address of the owner. The other one returns a boolean value determining whether its caller's address (msg.sender) is equating to the address of the owner. Because they don't alter the contract's state they are being implemented as "view" type functions.

Evaluation The contract's implementation by itself is rather basic. It doesn't include any of the flexibility any more advanced features such as roles. Still its inclusion in OpenZeppelin hints that it solves a reoccurring problem in the domain of Smart Contracts. It adds events in order to notify subscribers about transfers of ownership. In addition implementing functionality to permanently revoke one's own ownership can be seen as being dangerous. This is underlined by the fact that a fallback mechanism is not implemented. Potentially such a feature could be implemented by inheriting from the contract and overriding existing functionality. Without it an owner could accidentally revoke his own ownership permanently losing the ability to execute all annotated functions. Another disadvantage is that its entire access control model is based on the usage of modifiers. Therefore adding new functionality to a contract requires manual addition of a corresponsing modifier.

RBAC.sol[72] can be used in order to implement various access control types. The name already implies its usage of RBAC. The implementation's source file "RBAC.sol" imports from one of its own solidity libraries which is being stored in a separate file called "Roles.sol". It includes the definition of the following "Role" struct.

```
1 library Roles {
2     struct Role {
3         mapping (address => bool) bearer;
4     }
5 }
```

Listing 4.6: Role Library in Roles.sol

The listing shows that internally the role membership of a user is being determined by its address being mapped to a bool value. Until now there is no name of the role defined yet. This is being done within "RBAC.sol" instead.

```
1 contract RBAC {
2     using Roles for Roles.Role;
3     mapping (string => Roles.Role) private roles;
4     event RoleAdded(address indexed operator, string role);
5     event RoleRemoved(address indexed operator, string role);
6 }
```

Listing 4.7: Registering Roles in RBAC.sol

The code sample imports from the previously defined Role library and uses its struct in conjunction with a mapping. As the Role struct itself is only including a mapping the code can be deconstructed in the following way: mapping (string => mapping (address => bool)) dissectedMapping; This shows that Role Membership is expressed by a nested mapping. Further it can be derived that the strings serve as identifiers for roles. This is an understandable decision as by using nested mappings a role can only be defined once. If the same string would be used for the definition of another Role it would overwrite the existing contents. The mapping's functionality can be formulated as the following sentence:

a role can contain multiple addresses which all have an own individual bool value indicating their membership status.

Further RBAC.sol provides functionality regarding the removal and addition of role assignments. In both cases it emits an event notifying subscribers about such a change. It also includes a hasRole function which returns a bool value indicating whether an address is being linked to a role. In addition RBAC.sol includes a modifier "onlyRole" parametrized by a string value. It is being applied before its contained function's body throwing an exception in case msg.sender doesn't have role stated by its parameter. All of these operations are being internally conducted on the struct defined in Roles.sol.

Evaluation The intended purpose of RBAC.sol is to provide an inheritable contract. This

can be seen in the contract "RBACWithAdmin.sol" which is being included within the repositories example directory.

```
contract RBACWithAdmin is RBAC {
    string public constant ROLE_ADMIN = "admin";
    ...
    constructor()
      public
    {
      addRole(msg.sender, ROLE_ADMIN);
    }
}
```

Listing 4.8: Initial Assignment of an Administrator in RBACWithAdmin.sol Example

This allows a programmer to inherit the base contract potentially overriding functionality. As the code listing shows this can be leveraged in order to assign an admin role to the message's sender during the contract's construction. This address then refers to the account who deployed the contract. As a consequence of inheritance each of the resulting contracts follow a similar structure. They all implement events to notify its subscribers about changing role assignments and they all provide functions to assess and change the membership status of their users. The main benefits of this approach lie in its flexibility and reusability. Because the programmer can decide how to combine roles to more complex structures like hierarchical RBAC could potentially be implemented. The example contracts already provide functionality for Whitelisting. Logically following it is possible to derive an implementation of Blacklisting similarly. However the downsides of using this library are outweighing its benefits for more complex applications. For an example one could argue that its approach of linking strings to addresses is being error prone and requires constant care of a system's administrator. As it is expressing authorization rather primitively it requires the manual addition of modifiers to every newly implemented function. Forgetting to add them could therefore lead to a breach of protection. Further as it is not providing a full access control system it doesn't automatically provide any considerations on how to handle confidentiality issues. While this is by no means the intention of the OpenZeppelin project it still serves as a first step in assessing the landscape of existing Access Control implementations.

In addition there might be scenarios where the "hasRole" function might not be required at all. This is due to the fact that events are permanently accessible in Ethereum which allows a client to check whether a user has a certain role or not without the need to call a function. This is also being leveraged by another project implementing more complex access control via "Smart Policies"[25]. Its approach is being evaluated in the next chapter about XACML.

4.3.3 RBAC-SC

RBAC-SC is the concrete implementation of an access control contract described, implemented and published by multiple members of the IEEE. Its source code is hosted on

GitHub and provides an alternative approach to OpenZeppelin's RBAC in Ethereum. Because this article is quoted by multiple different works it offers another perspective in both the current state of access control as well as the scientific community's understanding of the AC discipline.

It bases its concept around a "role-issuing organization"[21] which manages roles and a "service-providing organization"[21] which confirms whether a user holds them or not in order to determine whether the user is able to access its service.[21] In order to assign roles the role-issuiing organization needs to deploy a Smart Contract which it maintains by adding new users including their roles. In case a user wants to access a service he sends a request to the service-providing organization stating that he is authorized by a role he holds within the role-issuing organization's contract.[21] To prove the validity of this claim both parties (the user and the service provider) execute a challenge response authentication. During its execution the service provider therefore requires the user to sign a random message by its private key associated with his Ethereum account's public key.[21] In case this public key is assigned a fitting role within the Smart Contract of the role-issuing organization it serves as a prove that the user in fact holds the role.[21]

Its code is contained in a single source file and makes use of modifiers, events and structs. The code doesn't explicitly define a constructor. Instead a function called "SCRBAC"[72] essentially resets all the contract's variables and initializes the contract. During this process an owner variable is set to the value of msg.sender. Other than the owner himself the system includes structs for the definition of two different entities. They are being referred to as "Users"[21] and "Endorsees"[21]. Both of them are being stored in separate arrays of their respective struct type. This is shown in the code listing below. It is being part of the original source code which is being provided by the authors via GitHub.[72]

```
1  uint public numberOfUsers;
2  uint public numberOfEndorsees;
3  mapping (address => uint) public userId;
4  mapping (address => uint) public endorsedUserId;
5  User[] public users;
6  Endorse[] public endorsedUsers;
7
8  struct User {
9      address user;
10     string role;
11     string notes;
12     uint userSince;
13 }
14
15 struct Endorse {
16     address endorser;
17     address endorsee;
18     string notes;
19     uint endorseeSince;
20 }
```

Listing 4.9: Structs in RBAC-SC

Both the mappings "userId"[72] and "endorsedUserId"[72] map to indices in their respective arrays. The sample shows that the code contains two variables of type uint stating the current amount of both Endorsees and Users. Both of these values are modified whenever new members of these groups are added or removed. The addition of new Users is only permitted to the owner himself. Still Users can add and remove Endorsees by appending or removing from the "endorsedUsers"[72] array. An Endorsee is then linked to the User who initially added him by reference via its field "endorser"[72]. The functions responsible for the addition and removal of Endorses are being protected by an onlyUsers modifier. This modifier performs a lookup operation within the mapping userId. A code snippet from the GitHub's repository[72] can be found below:

```
1 modifier onlyUsers {
2     require(userId[msg.sender] != 0);
3     _;
4 }
```

Listing 4.10: OnlyUsers Modifier in RBAC-SC

It can be seen that the modifier performs a lookup operation based on the address of the transaction's sender (msg.sender) in order to determine whether he is being a valid User or not. To achieve this it checks whether the corresponding value equates to zero. As the mapping's values are being defined as being of type uint any provided address would yield the value 0 if not explicitly assigned otherwise. Therefore RBAC-SCs construction includes an append operation to its arrays adding both an "empty" User and an empty Endorse to the arrays. Therefore RBAC-SC assumes a valid index to be greater than zero. This makes it possible to easily check whether an address is being registered as either User or Endorse. In order to do so it only requires performing a lookup within its respective mapping. If this operation returns a non-zero value it is being included in the array as it has a valid index. In order to link the appending of new Users (or Endorses) to the state of the mapping the functions for their addition and removal are not simply adding values to the arrays. Instead they simultaneously increases or decreases the total count of the corresponding entity and sets the mapping accordingly. In order to prevent gaps within the arrays the implementation therefore uses sorting within its "remove" and "add" operations.

Another means of exercising control within the system is being implemented by a state variable named "status". It allows the owner to set the contract's status to either true or false depending on whether it is considered active or inactive. If the contract is currently inactive no users can be added.

Evaluation In contrast to OpenZeppelin's Roles.sol no hasRole function is implemented. This is intended by RBAC-SC developers as they rely on a challenge-response based protocol including off-chain entities responsible for the actual enforcement of access control. This is being carried out by requiring the user to sign random data with its private key corresponding to his Ethereum account. This way it is possible to link his

identity to the role he holds within the contract.

Other than OpenZeppelin, RBAC-SC specifically tailors its solution to a given problem statement. Therefore its contracts are not as reusable as the contracts provided by the library. Instead it introduces endorsees with the intention of making transitive rights management possible even if is only to a limited degree.[21] As the text implies that the contract is deployed by the role-issuing organization it can be derived that the owner in the source code refers to it. As a logical consequence of this model the contract heavily centralizes its control. If the corresponding private key is lost no alterations of role assignment can be made. As the assignment of an owner only occurs within the function SCRBAC all role assignments are cleared in case of ownership changes. However the most important finding lies within the fact that RBAC-SC other than OpenZeppelin is based around the enforcement of a request.[21] It explicitly describes a scenario in which the actual enforcement of the request is being executed off-chain.[21] This raises the question whether this is desirable as it can't be guaranteed that a service provider is really carrying out a successful request of a user.

Summary of RBAC-Evaluations The main issue of the previously implemented solutions is that while they are rather lightweight none of them includes a any abstraction for actions or objects. Instead a developer is required to implement these features by himself. If this is not done however it results in the necessity of applying modifiers to each operation they aim to protect. This is not optimal as programming errors like forgetting to add them can easily break protection. Both these contracts are therefore better suited for smaller scale projects. If complex access control is to be used both RBAC contracts require the introduction of multiple roles. This makes it questionable whether these approaches are suitable for more complex scenarios. Another more flexible solution is based on a language used for modelling access control. It is called "XACML"[25].

4.3.4 XACML and Smart Policies

"Smart Policies"[25] are an XACML based approach implemented in Solidity. They were proposed by the work "Blockchain Based Access Control Services"[25]. They use a combination of on- and off chain programs to both compile XACML policies into executable Smart Contracts and deploy them.[25] It calls these contracts "Smart Policies". Each of them exposes functionality to evaluate its value. This evaluation is being equal to the response a Decision Point would yield.[25] During their evaluation they query so called "Attribute Managers" [25](AMs) which are used to fill the policy's contextual information and are therefore effectively replacing Information points. AMs are being deployed as Smart Contracts and their address being hard coded into the Smart Policies themselves during their compilation phase.[25] In order to remove Smart Policies each of them contains self destructing functionality which can be called by their owner. In order to keep track of the deployed policies' addresses another off-chain component is used.

It is called "Policy Administration Point"[25] and keeps a "Smart Policy Table"[25] (SPT) of all the deployed Smart Policies including their address. Incoming requests are handled by an off-chain PEP exposing an API to users.[25] In case of an incoming request at the PEP's API it establishes a connection to the last off-chain component known as the system's off-chain "Context Handler"[25] indicated by its abbreviation (CHo). The Context Handler then interacts with the PAP by passing the request's information about the resource ID it requests.[25] The PAP is then able to respond with a matching contract address.[25] The Context Handler then encodes the request's parameters to a contract processable format and forwards it to the Smart Policy.[25] Its execution is then yielding either a "Permit"[25] or "Deny"[25] statement. The PEP is then responsible for executing the request. Policy creation is occuring whenever a "Resource Owner" enters a new policy into the PAP. [25]The PAP then passes the information about the new policy to the CHo which compiles and deploys it via the resource owners account.[25] The evaluation of Smart Policies allows the system to reach an on-chain decision which can be publicly verified.[25] Whenever decision about whether the request is denied or granted it is communicated to the network via events.[25] The events keep both the client and possibly other interested parties informed about the decision-making process. By listening to these events the resource owner knows about the request being permitted or denied.[25]

Evaluation The proposed system introduces a concept which wasn't introduced in the thesis until now. Namely the combination between on- and off-chain software. This however comes with a problem. While Smart Policies might increase the level of abstraction in their access control model to ensure a higher flexibility than OpenZeppelin's RBAC contract its off-chaining comes with a disadvantage. The system itself can't be used in a "Smart Contract" only approach. While its off-chain Context Handler can still modify the Blockchain's state itself it can't be forced to do so. Let's construct the following scenario:

A user wants to request the alteration of a fictional test contract. This contract only stores a variable X. In order to set it the contract offers a publicly accessible set function. If the user queries the PEP's API with this request the system runs into a problem. While the decision that the request was permitted might be successfully be published by the evaluation of the Smart Policy the actual enforcement and therefore execution of the set function needs the off-chain Context Handler to call it. This breaks the direct flow of information within the Blockchain by leaving its domain. Because of this the actual enforcement can't be guaranteed as the Context Handler could be disconnected from the network. This is even reflected in the work's discussion section where it points out that a resource owner could prevent a permitted request by manipulating the PEP.

In addition its flexibility regarding the dynamic addition of information can be questioned as the AMs address is being directly compiled into the Smart Policies. While this might be achieved by a redeployment of new policies or other workarounds the process seems rather complex. Therefore it is considered as less reusable than a

framework like OpenZeppelin. In addition it leaves the responsibility for modelling users and resources in the hands of a programmer. Therefore it doesn't provide any unified user or resource model. As a consequence it doesn't include authentication mechanisms.

While this is not part of its scope a full access control requires the inclusion of a basic authentication just as much. Because many approaches rely on simply enforcing their decisions simply based on the value of the constant "msg.sender" they lack abstraction and are less flexible than a system based on URI's identifying the users.

While this by itself is a universal approach and therefore not to be considered rather special it imposes important features to user management as this allows them to be subjects of access control just as much as any other resources.

"Blockchain Based Access Control Services"[25] itself correctly describes "auditability"[25] as one of the benefits of the public evaluation of Smart Contracts this feature can be pushed even further by placing additional components such as the actual storages on the Blockchain. To a certain degree this is already being done in both RBAC-SC and RBAC.sol as they also use basic data structures like arrays and mappings to store their users. However the thesis expands on this by including a more extendable storage model into its considerations. Because the evaluation of existing approaches is now concluded the next chapter can begin to outline the requirements of its proposed model. To achieve this it first starts off by defining its requirements. This critique therefore serves as a guideline in the process of the thesis' presented model and its following implementation.

5 Modelling and Implementation of Access Control Mechanisms in Ethereum Smart Contracts

This chapter describes an approach on how to model access control for the Ethereum Blockchain. As pointed out in previous chapters there are different downsides of using Blockchain technology. Depending on one's individual requirements the system needs to be adaptable for either providing a high degree of confidentiality or a high degree of transparency as these goals are being directly opposed to each other.

While different presented approaches were only based on modifier usage they lack the abstraction a policy language like XACML provides. On the other hand the presented XACML based solution[25] uses off-chain compilation during their access control process effectively negating many of the benefits of running the complete process transparently on a Blockchain. As a consequence the thesis therefore provides a synthesis of the evaluated access control solutions by both basing its core architecture on XACML and providing a reusable framework at the same time. It aims to allow a developer to actively decide how to adapt the system's evaluation of decions according to his own liking. Therefore its goal is to find a balance between OpenZeppelin's reusability and XACML's flexibility.

5.1 System Requirements

One common strategy in the domain of software engineering is to base a system's implementation on the "V-Model"[6]. It describes a system's implementation an iterative process ultimately resulting in the execution of various test cases. This makes it a suitable approach as it provides a structure which can be followed throughout the complete process of formulating, implementing and testing the system. Laying down the base requirements of the system later aids in the evaluation process as it can be determined whether it provides the initially desired functionality. Therefore the first part of this chapter is dedicated to the assessment of the system's most important requirements. Traditionally they are being categorized as being either "functional"[52] or "non-functional"[52].

While functional requirements revolve around a system's "actions"[37] non-functional requirements define desirable "attributes"[37] of the program instead. As the basic principles of requirements engineering have now been laid out the modelling process can start by declaring the different properties it tries to fulfill. Because the implemented

system is heavily based on traditional XACML the following specification focuses on the modifications of the newly proposed system. Still the base XACML system aims to preserve all of the existing XACML features except for its policy language.

Functional Requirements
User
UR 1) ...shall be able to send a request
UR 2) ...shall be able to verify the state of his request
Authentication
AU 1) ...shall be able to register a User within a User Storage
Storage Contracts
XS 1) ...shall provide interfaces for CRUD via URIs
XS 2) ...shall notify Subscribers when CRUD data
XACML
Enforcement Point
XE 1) ...shall be able to enforce requests on-chain
XE 2) ...shall be able to notify off-chain Enforcement
XE 3) ...shall notify Subscribers about a Grant
XE 4) ...shall notify Subscribers about a Deny
Decision Point
XD 1) ...shall be able to read from a User Storage
XD 2) ...shall be able to include retrieved User information during decision
XD 3) ...shall notify Subscribers about a Deny
XD 4) ...shall notify Subscribers about a Grant
XD 5) ...shall notify Subscribers when its connections to Information Points change
XD 6) ...shall notify Subscribers when its connection to Policy Repository changes
Information Points
XI 1) ...shall be able to respond to the Decision Point
Policy Repository
XP 1) ...shall be able to respond to the Decision Point
Non-Functional Requirements
NF 1) Extendability
NF 2) Security
NF 3) Availability

Table 5.1: Requirements Specification

Users, Authentication and Storage Contracts The specification is subdivided into different categories. The first one is describing the "User" component. It states that every user should be able to send requests and to verify the state of his request (UR1 UR2). This allows him to directly communicate with the XACML system. Because XACML's main responsibility lies in handling the authorization instead of the authentication the

specification explicitly includes an Authentication Component. Its main feature is the registration of users (AU1). By including this component into the system the model provides another level of abstraction on top of the User's EOA address.

In addition the specification lists Storage Contracts. They are being components which are implementing basic functionality for reading and writing. The specification states that these operations should be callable on the storage's data via an URI (XS1). A primitive example of such an URI could be a consecutive number like in a database. However the specification does not enforce any specific type of URI by itself.

Information Points and Policy Repository Both the Information Points and the Policy Repository are required to communicate with the Decision Point (XI1, XP1). While this is part of the standard XACML model it is being included within this specification to highlight that both of them provide the Decision Point with data as this is being used by the thesis own implementation of a minimal policy language. Note that the specification defines Multiple Information Points instead of a single one.

Notifications The document expands traditional XACML functionality by requiring the emission of notifications whenever an Enforcement Point or a Decision Point decides or responds to a request.(XE3, XE4, XD3, XD4). Additionally XD5 and XD6 state that a Decision Point needs to inform its subscribers in case its connection to the Information Points or the Policy Repository changes. As these two points are an XACML system's source of data these notifications provide another layer of security within the system. Similarily Storage Contracts send events whenever their data is being either altered or read via its interfaces (XS2). This can be a valuable information to resource owners which could be responsible for the storage contract's administration.

Enforcement & Decision Point The requirements XE1 and XE2 are providing functionality both regarding on- and an off-chain enforcement. The former requires the newly implemented Enforcement Point to be able to completely enforce a request on a resource held by one of the system's "Storage Contracts". Therefore the system requires the implementation of feature XS1. By leveraging and combining the operations provided by this feature the enforcement point is able to execute varying operations on a storage. XE2 only provides a notification for off-chain services. This allows the framework to be used in different environments. In case a use case favors transparency a request can be handled by the Enforcement Point oneself. Otherwise it needs to be delegated off the chain leaving essentially no proof of the request's actual enforcement. This way the framework allows the implementation of flexible off-chain solutions. If the system operates in this mode it is essentially being a variation of the approach outlined by the work of about Smart Policies as they are also being monitored by an off-chain system which waits for their evaluation. However instead of relying on the constant compilation of Smart Contracts the thesis introduces its own basic policies based on matching pairs of conditions and attributes. These constructs and the reasoning behind

linking the Decision Point to a user storage (XD 1) is part of the Modified XACML Architecture the system provides. It is described after a short explanation of the system's non-functional requirements.

Non-Functional Requirements In order to include important attributes the software needs to fulfill the following Non-Functional Requirements have been declared:

- **NF 1) Extendability** can be considered as one of the most crucial properties of the proposed systems. As it aims to advance the state of existing access control solutions one important feature it has to offer is to be extendable and therefore adaptable to individual problems. This property is directly derived from the evaluation of OpenZeppelin's contracts. While it is a rather unrealistic goal to provide a feature complete extendable system right from the start its implementation and design designs still lay the groundwork.

- **NF 2) Security** is a primary concern in the design process of this solution as both "public" and "external" functions potentially expose a contract's state to the outside world. Therefore the interactions between the different components of the system need to be secured. Instead of relying on arbitrary user inputs they need to implement a strict flow of information relying on trusted components of the system.

- **NF 3) Availability** is implicitly being included by system design. As long as the system runs in full "on-chain mode" it is able to provide high availability due to the fact that it is being executed by every participant of the network. However its availability may vary depending on the amount of participants within a Blockchain. This is a crucial consideration to make as the system can be run in private Quorum instances just as in an Ethereum Blockchain.

5.2 Modified XACML Architecture

One crucial design decision linked to its reusability has to be made right from the start. As the system's goal is to to be use able in both full on-chain and partially off-chain systems it is required to provide a simplified policy language. This decision is made based on the verbosity of both XML and XACML itself. An important downside of not making use of the original policy language is that the proposed system can not possibly achieve a similar level of flexibility. However one can argue that at the current state of research this is still a rather unrealistic approach as the gas costs linked to the evaluation of such policies can potentially be very high. This assumption is being made based on the fact that in order for policies to be evaluated they are being parsed first. This can be seen by looking at public source codes by other policy based frameworks such as "node-Casbin".[58] While a full explanation of the underlying reasons is being omitted here it can be understood intuitively by imagining that a policy has to be filled by inputs

such as the user ID in order to be evaluated. This however is extremely ineffective regarding gas costs and also directly opposes the ideas of the "Low Contract Footprint Pattern" which states that a Smart Contract has to strictly optimize its usage of write operations. This is based on the fact that read statements such as pure functions don't cost any gas when being called from EOAs. If a system relies on multiple evaluations of policies this can therefore be regarded as rather inefficient. In addition it should be noted that executing string operations in Solidity is still rather complex. By default strings don't provide any functionality such as splitting or substrings.

Similar to Smart Policies[25] the system is based around attribute based policies. The policy model is based on the assumption that each policy can be expressed as a tuple of (userID, resourceID, actionID) = requiredAttribute. This attribute is then either being supplied by an Information Point or not. As a consequence both the Information Point and the Policy Repository receives the request tuple and return either an attribute (Information Point) or a condition (PolicyRepository). If necessary they can be linked to an internal storage and perform lookup operations within it to determine which of its attributes are related to the request. The Decision Point then queries both its Policy Repository and all of its Information Points for the requested conditions and the provided attributes. It then proceeds to combine them in order to come to a decision on whether the request should be granted or not. Additionally the Decision Point is being linked to a User Storage. This can be regarded as a modification to base XACML. The reasoning behind this is to allow the Decision Point to resolve the address of the user sending the request to a UserID. This is necessary as the policies should not depend on "msg.sender" but instead the user's index as an URI.

As the intention of the system is to maximize public auditability all its storage systems need to be indexed in order to provide public lookup functionality. This decision is based on the thesis assumption' that a programmer who wants to to publish private data takes precautions such as using encryption or either permissioned or private Blockchains. The public accessibility of the storage allows each User to confirm that his request was actually enforced instead of just being simulated by the Enforcement Point.

In addition the introduction of an indexed storage system aids in the development of a Authentication Point as it provides a storage system for registered users. Because a user management solely based on identifiying users via "msg.sender" is rather unflexible regarding lost keys etc. abstracting them by storing a reference to their public address allows a replacement of public keys by storage administrators or authentication contracts. Therefore the thesis aims to provide a inheritable Base Contract for implementing one's own variation of authentication.

Other than in base XACML and the Smart Policy system the thesis' Enforcement Point is being subdivided into two different implementations by inheritance. Its base contract only includes a simple request function which is publicly accessible and basically mirrors the decision of the Decision Point by returning it. As a consequence of the proposed request tuple this function takes an actionID and a userID as its only parameters while returning a bool indicating its decision. This decision is based on the fact that a user

can't be trusted to send his correct ID. If he was able to provide it by himself it would open the system to possible manipulation attempts. After the It returns either true or false depending on whether the request was granted or not. It is important to note that this base contract intentionally does not provide any enforcement yet. Therefore the simplest implementation of an Enforcement Point is provided by its off-chain variant which effectively only serves as an emitter of events to off-chain systems. As they need to be able to interact with a storage contract they can be declared as its owner similar to the modifier based approach both OpenZeppelin and Ethereum's documentation provide. As all of the storages offer public auditability by simply being readable by system participants the parties can agree to use it as an auditable interface. Therefore the storage needs to allow an arbitrary resource owner to manage its data. In case he requires full privacy he could potentially deploy it as a private Smart Contract on Quorum in order to share it with only few exclusive participants of the network. However as the thesis didn't place much emphasis on the implementation of an efficient storage system this is not advised.

Even if the Enforcement is delegated to an off-chain system the previous decision still needs to be communicated. This is being already described by "Blockchain Based Access Control Services"[25] which describes "auditability"[25] as the main reason for doing so. This opinion is valid and therefore needs to be reflected by each access control system on the Blockchain as its one of its main advantages over non-Blockchain solutions.

In order to provide a even higher degree of auditability than Smart Policies are able to provide the thesis' system intentionally offers the possibility to implement a full on-chain enforcement. To achieve this even actions need to be stored within an accessible storage. As they both need to provide functionality and need to offer references they are being modelled as executable Smart Contracts. The way this is achieved is therefore being part of the next chapter explaining the derived models which served as a reference for the system's implementation.

5.3 Derived Models

The previous chapter already pointed out the key modifications which have been made to the original XACML system. In order to present how the system is able to provide the desired features and modifications this chapter lays out the different models. Each of them has evolved through an iteration of multiple re-implementations. As the system is being based on a set of different acting components each of them is now being explained in greater detail in order to provide information on how to implement a similar system on one's own.

Smart Contracts

Smart Contracts and Transactions are used to implement every functionality of the system. This way it can be used by both Quorum and Ethereum. By listening to a

contract's events any participant can be informed that a user sent a request and remains informed about its handling. Named Contracts Almost every contract in the system inherits from the Contract "Named". This contract's only functionality lies in exposing a publicly accessible name of type bytes. This allows providing an ASCII name for the child contracts. Due to its simplicity a UML diagram is omitted at this part of the thesis. However it can be inspected by referring to complete UML diagram. Indexed Storage Any contract offers the capability of allocating persistent storage which is directly stored on the Blockchain. By applying a layered architecture a smart contract can therefore be used as a primitive persistence layer. Consequentially such an approach introduces the common Blockchain benefits of protecting the storage's integrity ensuring a high availability due to its distribution. On the other hand data replication is costly and data privacy might be more important than transparency. As the design decision depends on a company's model the distribution of priorities can be gradual.

Depending on the system, different entities require a certain amount of storage. Just like applying a modifier based access restriction strategy requires storing a user's address, policies and other types of entities all need to include storage.

In contrast to other works the thesis implements its own storage contracts. Each of them essentially provides the same functionality. They support basic operations such as reading via retrieval, writing, overwriting and deletion. As a consequence of their

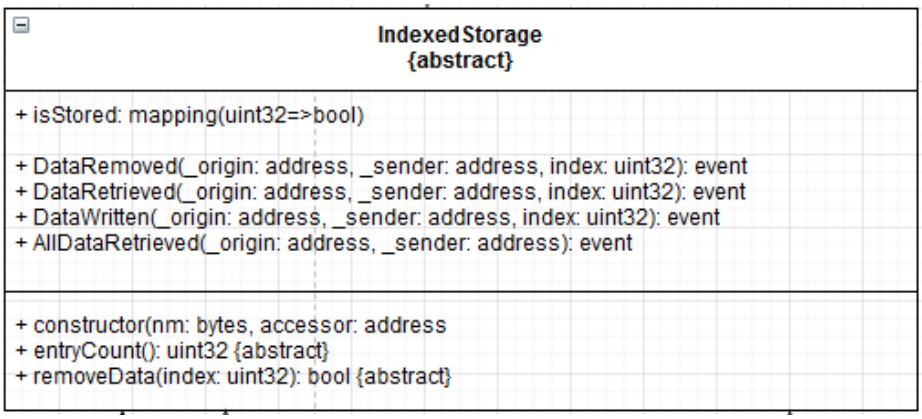


Figure 5.1: Indexed Storage UML

similarity multiple contracts inherit from the "IndexedStorage" contract shown above. As it is being declared as abstract it can't be instantiated directly. The most important functionality this contract provides lies within its mapping. Just like OpenZeppelin's Roles.sol mapped addresses to a bool in order to determine whether a user has a role the IndexedStorage maps indices to a bool in order to determine whether they are included or not. This allows implementing an efficient retrieval strategy by terminating retrieval operations early in case the mapping's state indicates that a specific index is not yet set. As indexing lies directly in the responsibility of the described contract this state variable is therefore being included.

In addition the contract's basic capabilities include the emission of multiple events. They occur in case of writing, overwriting or retrieval. The differentiation between overwriting and writing is that writing requires a currently unwritten index as a parameter. This is to ensure that no accidental overwriting occurs. Internally this check is executed by looking up the index within the mapping and determining whether its value either corresponds to true (index already set) or false (index currently unset). Such a feature is not included in overwriting.

In addition to the events it provides basic interfaces for either querying the amount of currently included items within a storage or removing data via its index. While this might seem as being rather limited the contract's interfaces are intentionally being designed this way. The reasoning behind it is that Solidity does not support any concept such as generics. However as a more detailed explanation of this circumstance is being discussed in more detail within the chapter about "Specific Implementation Details" it is omitted here.

The contract's constructor takes two arguments. The first one of them is of type bytes. This is going to reoccur frequently throughout these models as it refers to the contract's name (inherited by the Named Contract). The second parameter is of type address. This is being used to link the storage to another contract in order to limit access to its functionality. It is therefore used in order to ensure that only administrators and another contract are allowed to write its storage. An possible example includes linking a PolicyRepository to a storage in order to grant it write rights. As the underlying functioanlity is being inherited from the "Protected Contract" specific details on how this is achieved can be found in its corresponding section.

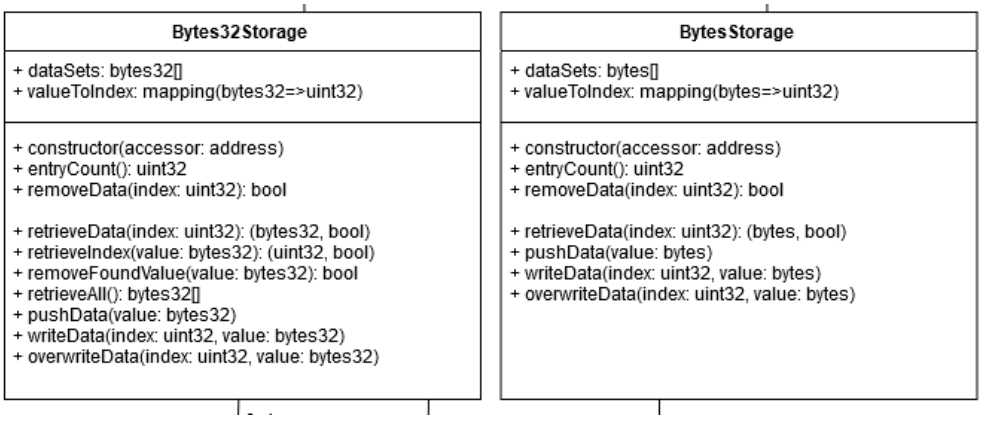


Figure 5.2: Bytes32 & Bytes Storage UML

The previous example shows two children of the Indexed Storage. As can be seen both of the contracts provide functionality for data retrieval, writing and overwriting. The dissimilarities between both of them arise due to limitations regarding the way Solidity handles strings. More information about this circumstance is being provided in a later part of this chapter. However the diagram shows that BytesStorage offers the

most basic capabilities a storage can solve by including an array of bytes. This array carries all the information the storage contains. It is therefore the source and target of the operations the contract provides. As can be seen these include pushing, writing, overwriting and retrieving. Every function except pushing is based on the data's index. This can be understood intuitively since pushing refers to the act of appending to an array.

The other representative of an IndexedStorage is the Bytes32Storage. The system uses it for storing the conditions of policies as well as the attributes provided by Information Points. As it is not based on bytes it offers extended functionality. This includes operations for retrieving a dataSet via its index. When it returns it includes both the index and a bool value. This value includes information of whether the data was actually found or not. The reason for this is that the system relies on unsigned indices. Therefore it is unclear whether an index of 0 indicates a non-existing entry or an actual element. Another possible implementation would be to append "null" elements such as RBAC-SC did when defining its users and endorsees.

Alongside the two presented types of storage the system also includes an AddressStorage containing an array of addresses instead. However it only differs from a Bytes32 by storing addresses instead of bytes32 values. Its main purpose is that it is being used for storing the users and actions of the system. However as it stores addresses it can be leveraged to store arbitrary references to existing Smart Contracts.

The reasoning behind modelling storage in such a way is that just like a traditional database it uses indices for making data accessible. An index can be used as a simple URI. Like in REST this ensures a globally equal system of addressing resources and simplifies the modelling process of access requests. In order to support a combination of multiple storages is imagineable. As the implementation includes an AddressStorage they can be linked recursively.

In addition the system can accomodate generic URIs to external systems. This is being done in a pattern called "Content-Addressable Storage". Such an approach is possible as the BytesStorage could contain arbitrary URIs of external off-chain systems. The software pattern makes use of Content Addressable Storage systems which are indexed by the data's hash values. Further the pattern states that storing these hashes on a Blockchain turns it into a kind of register. In addition such separation of storages can be used in order to limit the data's readability as it is not guaranteed that a node of the Blockchain has access to the external system.

While the inclusion of these types of systems is a byproduct of the aim to provide a general purpose storage there are still valid reasons for relying on an off-chain storage. One of them could be that the Blockchain carries each historical information of all the transactions. Therefore the understanding of the delete and update operations has some important implications as they are essentially undoable.

Enforcement Point As in base XACML the Enforcement Point is the central point interacting with both the resources and the user. The proposed system therefore implements the following Smart Contract.

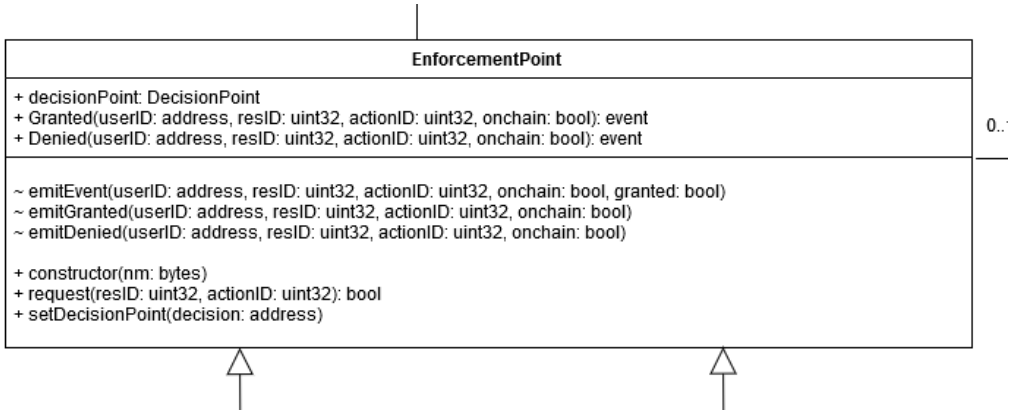


Figure 5.3: Enforcement Point UML

As the diagram shows it has to be linked with a Decision Point. The reason for this is that it either enforces a request or not depending on the decision from its referenced DecisionPoint. Therefore the user itself is not involved in the decision process whatsoever. It only allows a user to send a simple request encoding what operation he intends to perform and its target provided as a resource ID. It can be seen that both of the parameters are being passed as being of type uint32. The reasoning behind this is that the storage uses this data type for its indices. Thus it is apparent that both parameters refer to indices within a storage.

This can be leveraged for both on- and off-chain storage scenarios as has already been pointed out in the previous section about the IndexedStorage.

One notable design decision is that its model does not include any attached Storage by default. Therefore it allows the implementation of an storageless Enforcement Point. This can be used in order to turn the Smart Contract into a proxy for further off-chain Enforcement solely interacting via the emission of events. Consequently these events mirror the state of the of the request by either being of type "Granted" or "Denied".

Along this implicit information they encode the address of its sender's account, the resource ID and the actionID as well as a bool indicating whether the enforcement was carried out either on-chain or off-chain. Thus this presents the purest form off off-chain enforcement.

In its most basic form the corresponding request only encodes identifiers which have no representation on the chain itself. Instead the off-chain system needs to provide its own mapping of operations and users. Therefore a developer can gradually decide which level of transparency he wants to provide by including additional storages. By itself this does not necessarily include any operations on storages' data but instead allows using storages as a public interface for further information. As an example a Storage of type bytes could serve as an register containing an operation's description. This way system's participants could receive additional information about the enforcement which is handled off-chain.

In contrast to the pure off-chain approach on-chain enforcement requires the ability

to execute actions. As it was already hinted in the previous chapter this is being done by referencing executable Smart Contracts. Consequently they can be stored within an address type Storage. This makes it possible to model the following chain of Smart Contract inheritance:

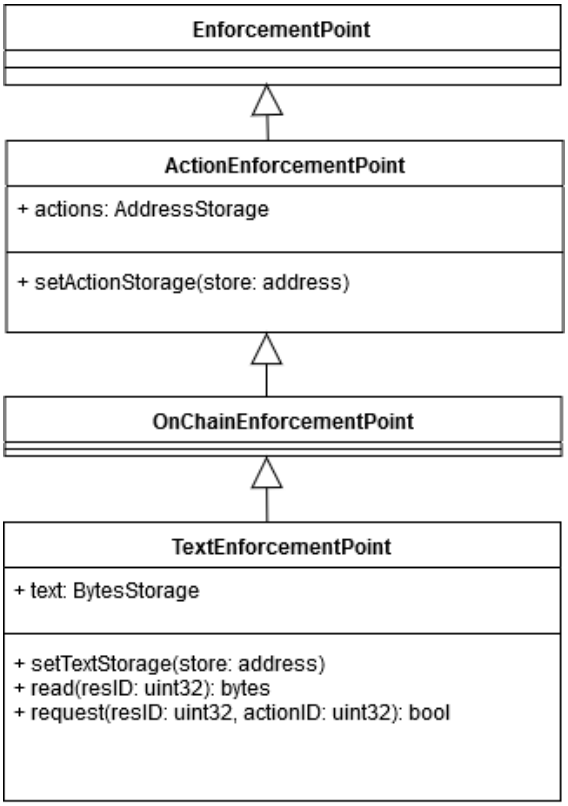


Figure 5.4: Chain of Inheritance Enforcement Point UML

The example shows how a TextEnforcementPoint can make use of the given infrastructure in order to attach both an AddressStorage representing its actions as well as a BytesStorage containing arbitrary sized text. Combining both of these components grants it the ability to execute operations directly on the chain as he is being granted the ability of altering and reading the attached storage's text. This way he can call a related action by delegating the operation itself to the referenced Smart Contract. A more detailed description of how this is achieved can be found in the following paragraphs.

The proposed system uses abstraction in order to provide actions. Because they are only being referenced by their address it is possible to model one's own arbitrary combinations of Enforcement Points and corresponding actions. However the thesis includes a sample implementation based on the simple assumption that most alterations of data can be modelled by a single input parameter being converted to an output of the

same type. This can be modelled the following way:

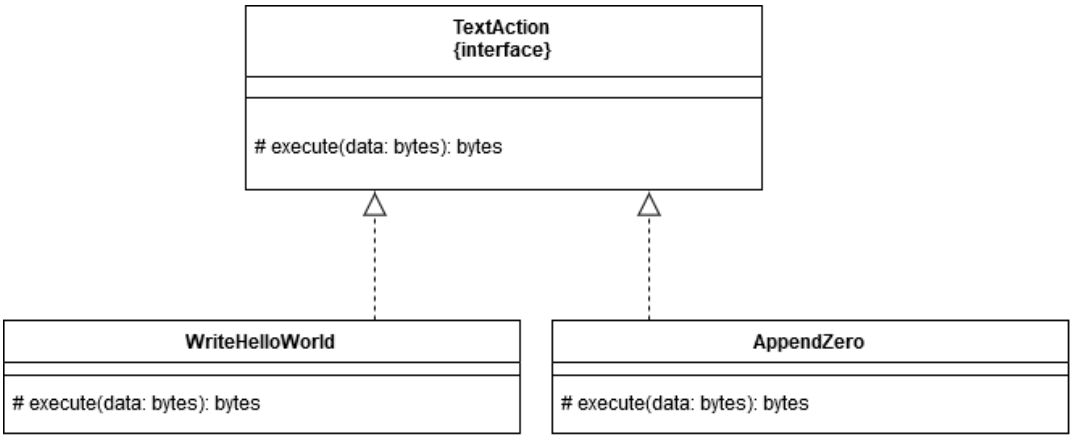


Figure 5.5: Inheritance TextAction UML

The diagram shows the definition of an interface called "TextAction". Because interfaces can not be instantiated directly they need to be inherited from in order to provide matching functionality. Thus two child contracts are shown. Both of them implement the parent's functionality by providing their own interpretation of the execute function. The contract called "AppendZero" works the following way:

- Its execute function is being called including a bytes encoded text parameter, e.g the unfinished statement "1 is not equal to"
- The AppendZero contract receives and alters it by appending the hex representation of the character 0 (0x30)
- Consequently this results in the string "1 is not equal to 0"
- Afterwards it returns the altered text

The example shows that this way multiple different operations can be modelled. Its counterpart WriteZeros is ignoring the input's parameter and blindly responds with the bytes representation of a string containing 32 zeros. An Enforcement Point can then use this result in order to alter the state of its attached storage based on a predefined action. In order to provide more transparency on the internal workings of the action itself it is possible to publish its actual source code for the public. This is especially interesting in the context of a using the website Etherscan as it offers a feature to verify contracts. This procedure uses a contract's address as well as its source code in order to link both of them. This is done by comparing the compilation results of Etherscan itself with the ones published at a given address. This ensures that an action does what ever a source code states.

However the more direct way a user can confirm that a contract executed its request is to compare the results of its attached storage with the ones he expected.

Attributes & Conditions Until now the explanation ignored the internal handling of decisions in favor of explaining the enforcement model. As the policies in the system are expressed as conditions and attributes it is necessary to explain them first. As previously expressed the decision to exclude the XACML language is based on its verbosity. Thus the thesis implements attributes and conditions based on the datatype bytes32.

It is the biggest fixed size data type Solidity provides and therefore does not suffer from the limitations of other data types such as bytes or strings. As an ASCII character requires exactly one byte such a value can therefore encode a string of 32 characters. Consequently an attribute could encode the attribute and its corresponding condition the following way:

```
AsciiToHex("isAdmin") = "69 73 41 64 6d 69 6e".
```

The way both of them are processed is being explained in the following paragraphs as they are involved in the decision process of a Decision Point contract.

Policy Repository & Information Points
Because both the Information Point and the PolicyRepository are communicating with the Decision Point by providing attributes and conditions their processing is being simplified by both of them sharing the same type.

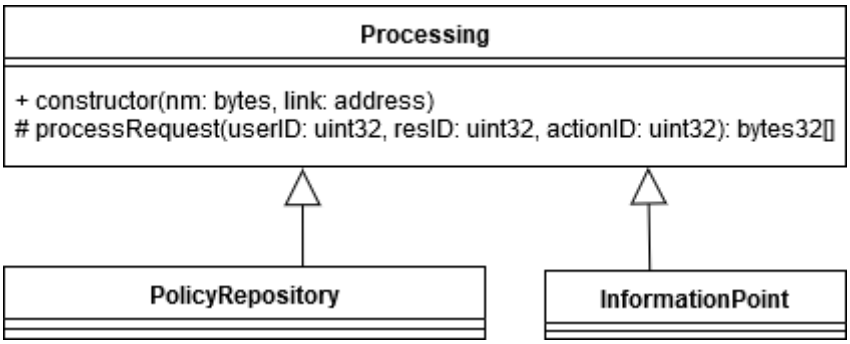


Figure 5.6: Processing Contract UML

This is being achieved by both of them inheriting from a common parent contract. The diagram shows that the system enforces no other constraints on the child contracts except the specification of a processRequest function. The function definition already hints that it assumes that both the Information Point as well as the Policy Repository respond by arrays of type byte32. As the previous section showed this refers to their conditions and attributes. Both the Policy Repository and the Information Point can therefore perform internal operations in order to determine request related information. Consequently it is possible to attach Indexed Storages to them. This makes its possible

for the system to receive requests regarding its own policies by recursively applying the system's model and referencing either the storage holding the conditions or the storage holding the policies. The full model of an Information Point the thesis implemented can be found below.

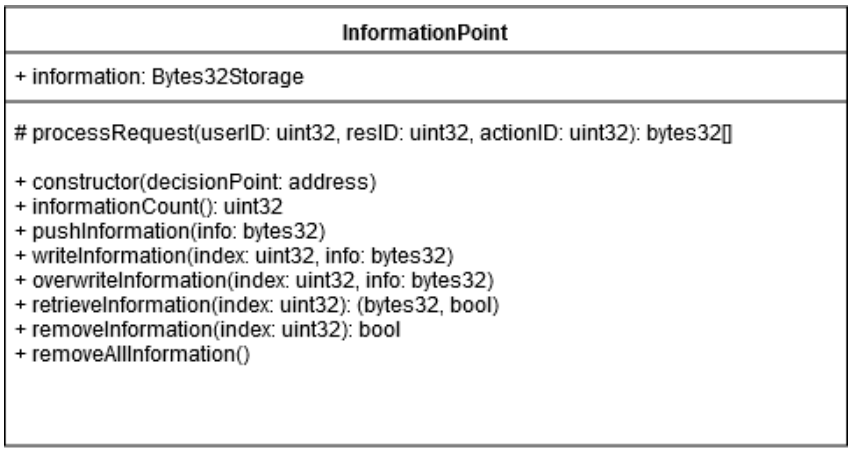


Figure 5.7: Information Point Sample Model UML

The example shows a basic Information Point. It responds to each processRequest call with the complete information contained within its referenced storage via a retrieveAll call. It is implemented in such a that the input parameters are ignored during the process of producing the output. Therefore it is functionally independent. In the context of Smart Policies a similar construct called "static" Information Point is used. However as a programmer is able to provide its own variant of an Information Point it allows adaptation for a wide range of different applications. As both the PolicyRepository and the InformationPoint are inheriting from the same parent class the following example of a PolicyRepository's implementation could therefore serve as an Information Point just as much.

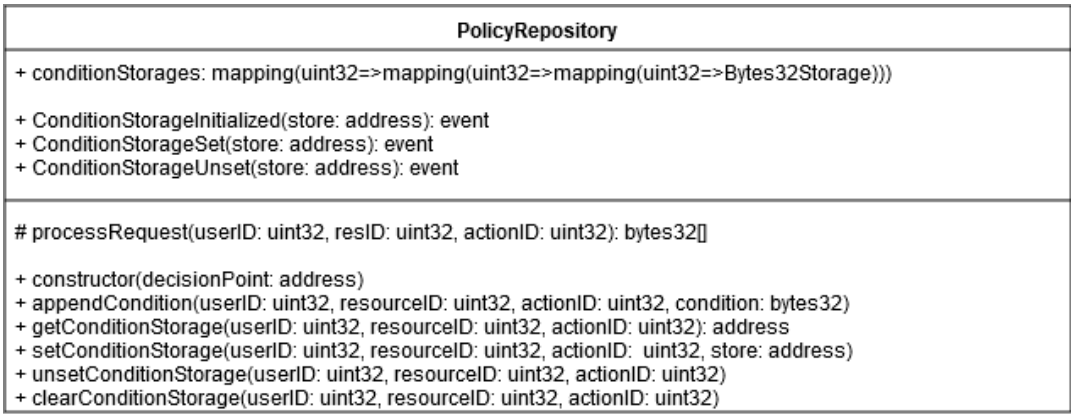


Figure 5.8: Policy Repository Sample Model UML

The model shown was used during the evaluation via software tests. Its main intention is to prove that more complex applications can be modelled via the proposed system. Thus by itself it seems rather complicated. However it is based on a simple process. Each tuple of (user, request and action) is being assigned a corresponding Bytes32Storage. This storage then contains all the conditions for the stated tuple. As the tuple corresponds to a request it allows assigning each individual request a set of conditions. In addition multiple requests can refer to the same Storage effectively grouping conditions together similar to the PolicySets provided by XACML.

DecisionPoint The DecisionPoint is the most central component within this system. As it extends on the principle of XACML it not only interacts with a PolicyRepository and different InformationPoints but also with a UserStorage. This is being shown in the diagram below.

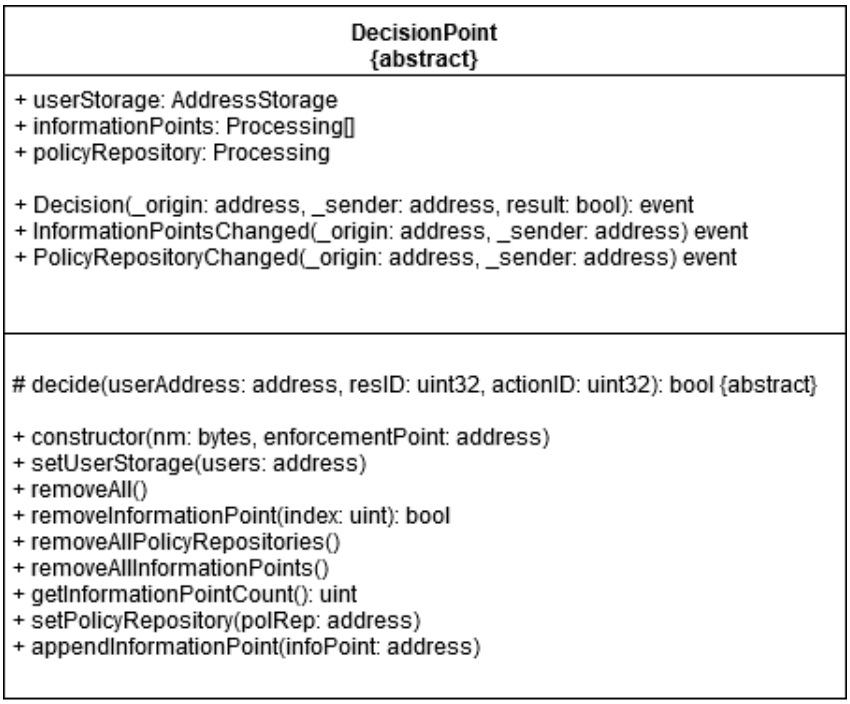


Figure 5.9: Decision Point Model UML

As the DecisionPoint explicitly references instances of the "Processing" contract for both its Information Points as well as its PolicyRepository a programmer is fully capable of providing its own implementation of both these components. As the contract itself is abstract it explicitly requires a developer to inherit from it in order to instantiate it. Along different operations for managing its policyRepository as well as its InformationPoints it allows to implement one's own implementation of the function "decide".

The UML model also shows that the decide function takes an address as its first argument. This corresponds to the "msg.sender" value of the original requester and is

being passed by the EnforcementPoint. This shows that the expected flow of information begins with a user's request to an EnforcementPoint which asks the DecisionPoint for its decision by including the sender's address. The User Storage can then be queried for the provided address to determine whether a user was registered or not and include this information in its decision making process.

Authentication Point & UserStorage The previous section already pointed out that the decision whether a user is registered or not can be rather important in the process of arriving at a decision about whether a request should be granted or not. As this was already explained in previous chapters about authentication and authorization the thesis provides its own extendable model.

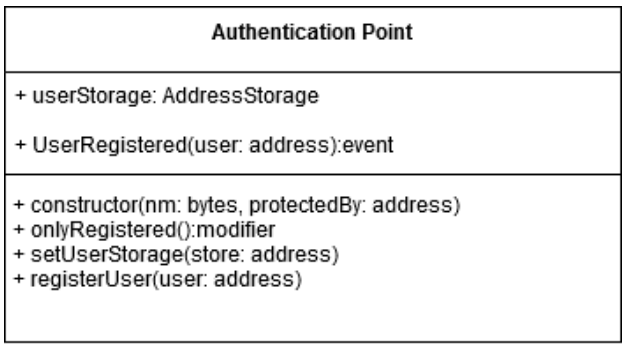


Figure 5.10: Authentication Point Model UML

As the model shows the Authentication Point references a Storage contract of type address. As both users and contracts are identified via its address it can serve as a persistence layer for them. The process of registering a user can be understood as adding it to the storage. Therefore his inclusion indicates that he is authenticated. The simplicity of this idea is being reflected by the diagram. In addition to the base functionality it provides a modifier which can be applied to either execute a function or throw an error message depending on whether "msg.sender" is registered or not. However as the provided system is capable of handling requests to any indexed resource the access to the referenced storage can be handled by another instance of the thesis system. By linking a User Storage to both the DecisionPoint and the Authentication Point it can serve as a central interface. This way it can be managed by the Authentication Point while being read-only queried from the Decision Point. To give a concrete example of an implemented Authentication Point the following model can be used.

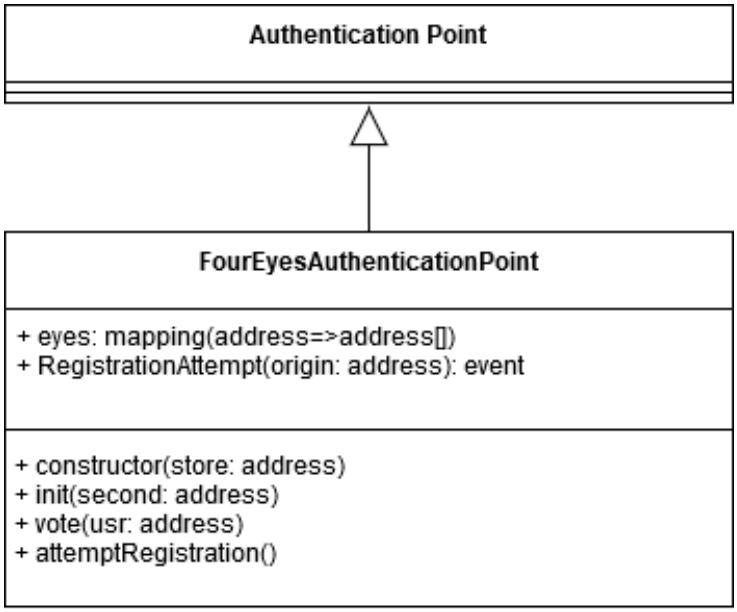


Figure 5.11: Four Eyes Authentication Point Model UML

provides a rather basic authentication system which requires an initiation providing two addresses of initial users. Both of them are instantly added to the linked storage and therefore registered. Then new participants are required to wait for the confirmation of at least two other registered members in order to be included within the Storage themselves. This in turn allows them to participate in subsequent votings about the inclusion of new members.
This concludes the examination of the all of the proposed components with the exception of the "Protected Contract". It is being used to restrict access between the different units of the system. As a consequence it is being involved in most of its processes. Thus it is being included in the next chapter.

5.3.1 Processes & Protection Mechanisms

Because contracts allow everyone to call functions it is necessary to differentiate between trusted and untrusted callers. In other words the system has to generally assume that a user tries to manipulate the it via attacks. This makes it necessary to expose as little functionality as possible to a user.

While it has already been pointed out that the privacy of transaction data itself can currently only be solved by either off-chaining or encryption the system still is able to differentiate between the different addresses indicating the origin of transactions. As OpenZeppelin's Ownership contract already showed this allows to implement basic access restriction via modifiers. Similarily the proposed system internally handles processes by applying modifier based access restriction between its components.

Protected Contracts
Instead of allowing every arbitrary user to call its input each component is restricting access to its vital functionality. The reasoning behind this is to ensure that the system can not be manipulated from external parties. To achieve this almost each of the previously explained contracts inherit from a super class called "ProtectedContract". The only exception to this rule is the definition of Actions as it is unnecessary to protect functionality which by itself only does calculations without altering the system's state. To give an brief overview of the importance of the Protected Contract its UML diagram is being shown below:

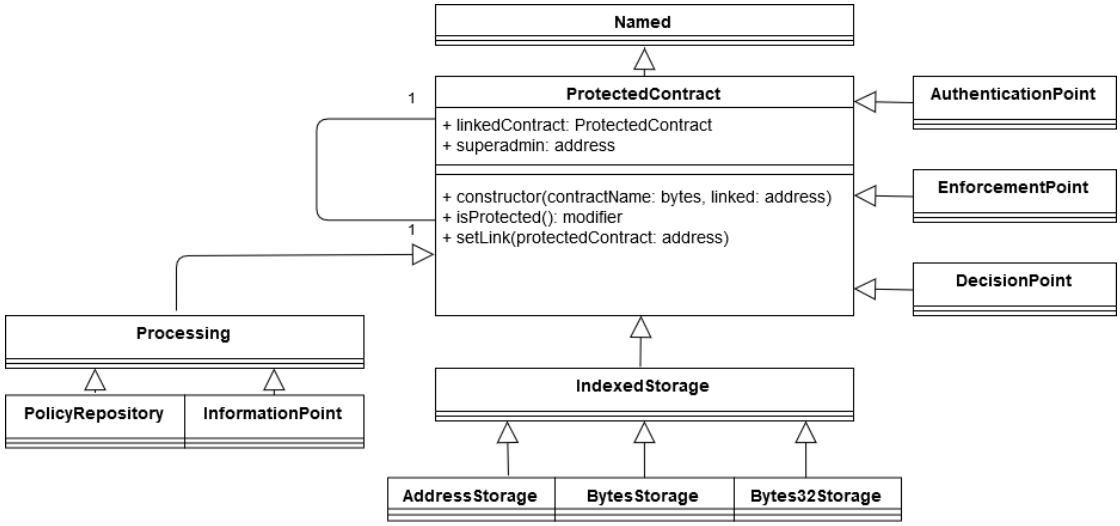


Figure 5.12: Complete Inheritance Graph UML

As can be seen the Protected Contract is being inherited by all the crucial components of the system. It is based on a variation of the Ownership Contract provided by OpenZeppelin. Thus it includes an assignment of an address during the contract's deployment. As its implementation is easy to follow the following listing shows its source code.

```
1  {
2  contract ProtectedContract is Named {
3  ProtectedContract public linkedContract;
4  address public superadmin;
5
6  modifier isProtected{
7      require(msg.sender == superadmin || msg.sender == address(linkedContract));
8      _;
9  }
10
11 function setLink(address protectedContract) isProtected public {
12     require(msg.sender == superadmin);
13     linkedContract = ProtectedContract(protectedContract);
14 }
15
16 constructor(bytes memory contractName, address linked) Named(contractName) public{
17     linkedContract = ProtectedContract(linked);
18     superadmin = msg.sender;
19     }
20 }
```

Listing 5.1: Source Code: Protected Contract

As can be seen a state variable called "superadmin" is initially assigned during the contract's construction. In addition it allows passing an address to a "linkedContract". Both these entities are the only parties who can execute a function modified by isProtected. This can be seen in line 7 which expects the sender of a message to either equal to the super admin or the linked contract. The setLink function is additionally restricted by a require statement which further limits its access to the superadmin itself. Inheriting from it therefore allows each component to make use the isProtected modifier.

Additionally during each of the components' constructions their state remembers the address of its deployer. This introduces the concept of administrators within the system. Each of them is capable of changing the components links to its counterparts. The importance of protecting these links can be understood by imagining a malicious user altering a Decision Point's state by changing one of its Information Points addresses. Doing so would allow him to provide an Information Point which could effectively mirror the attributes of other requesters within the system. As the previous chapters have pointed out he could obtain them by listening to the network's flow of transactions. Doing so would grant him operative access to restricted resources even though he was only able to manipulate a single point within the system.

Thus the ProtectedContract ensures that only a component's admin as well as their linked components are being granted access. Thus the following summary serves as a reference point of the protected functionality. This includes both all the storages' write operations as well as all the setting operations of relationships such as adding or removing a Policy Repository from its Decision Point. The important assumption this design makes is that both the administrator and the linked contract are not attempting to manipulate the system.

The initial setup of the system therefore requires an administrator to either provide a contract with a fitting LinkedContract or to set it to null (address 0). Doing so allows him to be the only entity within the access capable of doing write operations. If a User Storage's would remove its link the Authorization Point could not continue to register new users. However as read operations are not protected it can still read from it. The design decision to allow read access globally is based on the results of the previous chapters. Thus a programmer might opt for deploying the system via private Smart Contracts if he has a high need for confidentiality.

Another consequence of storages blocking write operations from non-trusted parties is that an on-chain Enforcement Point is unable to enforce a request on it by overwriting its data. One direct result of applying the XACML architecture is therefore that the decide function of the Decision Point can be modelled to be publicly accessible as it does not interact with other contracts via write statements.

This can be shown by the following example involving authentication:

- A user registers at an Authentication Point
- Thus the Authentication Point writes to the storage
- Consequentially he has to be the storage's linked contract
- Therefore a Decision Point has to be read-only

Consequentially a DecisionPoint does not require linkage to both its PolicyRepository and its InformationPoints. However a possible consideration is to link an inherited EnforcementPoint to them in order to do direct Enforcement on conditions and attributes. This allows to apply the system recursively effectively providing a flexible approach on access control.

Processes

As the implementation of the Decision Point's decision function lies in the hands of the programmer he can impose different limitations on the validity of requests. As an example of possible constraints being imposed the following list gives an overview of system conditions which default to a deny decision of the prototype's own implementation of a DecisionPoint called "RequiredAttributesDecisionPoint":

- Decide false in case no Information Points are connected
- Decide false in case no Policy Repository is
- Decide false in case no Policies exist
- Decide false if user is not registered

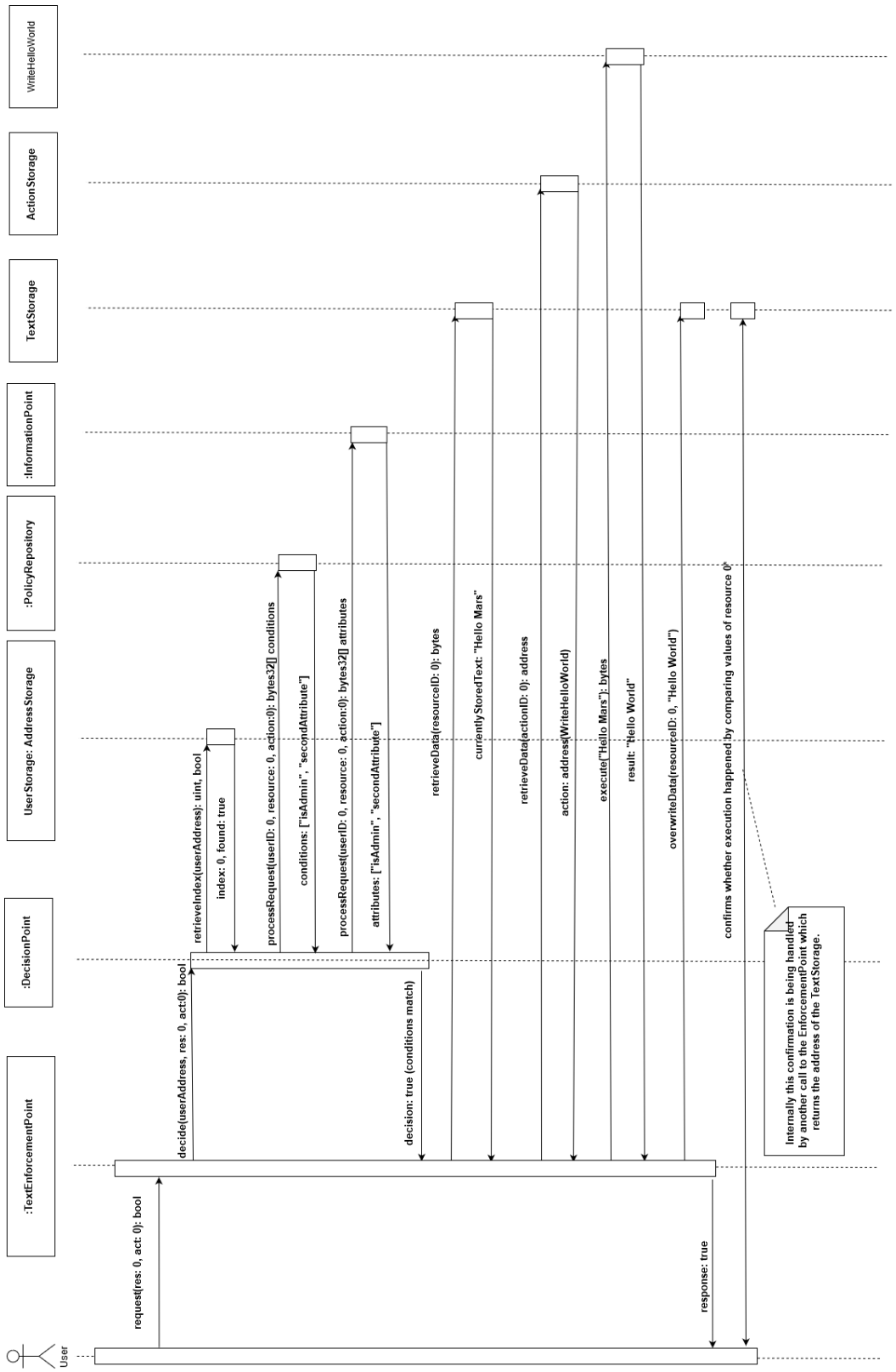
.

Similarily the implementation of an Enforcement Point could deny a request based on its connection status with an Action Storage or a Resource Storage.

In order to understand the way a complete request works a sequence diagram of a successful request is shown on the next two pages.

In the following the request presented in the sequence diagram is being textually described as a list of consecutive steps. It gives an example of the different steps a decision process could involve.

1. A user sends a request for the execution of action 0 on resource 0
2. The Decision Point queries the UserRepository to determine the user's registration status
3. The Storage returns that he found the user and includes his index
4. The Decision Point knows that the user is being registered and knows its internal ID
5. The Decision Point queries its Policy Repository for related constraints providing UserID, ResourceID and ActionID.
6. The repository answers with the conditions "isAdmin" and a randomly required "secondAttribute"
7. The Decision Point queries its (single) Information Point for attributes related to the request
8. The Information Point returns both attributes "isAdmin" and "secondAttribute"
9. As the Decision Point can confirm that the user holds both attributes it decides to grant the request
10. Thus the Enforcement Point retrieves the data identified by its index 0 "Hello Mars"
11. It then queries its ActionStorage for the action 0
12. The ActionStorage responds with the address of the contract named "Hello World"
13. The Enforcement Point calls the execute function of the "Hello World" Contract by passing the initial data ("Hello Mars")
14. The Action responds with "Hello Mars"
15. The Enforcement Point overwrites "Hello World" with "Hello Mars"
16. The user can confirm that his request was successfully carried out by reading the storage and confirming its results answers with


WriteHelloWorld
ActionStorage
TextStorage
:InformationPoint
:PolicyRepository
UserStorage: AddressStorage
:DecisionPoint
:TextEnforcementPoint
User
request(res: 0, act: 0): bool
decide(userAddress, res: 0, act:0): bool
retrieveIndex(userAddress): uint, bool
index: 0, found: true
processRequest(userID: 0, resource: 0, action:0): bytes32[] conditions
conditions: ["isAdmin", "secondAttribute"]
processRequest(userID: 0, resource: 0, action:0): bytes32[] attributes
attributes: ["isAdmin", "secondAttribute"]
decision: true (conditions match)
retrieveData(resourceID: 0): bytes
currentlyStoredText: "Hello Mars"
retrieveData(actionID: 0): address
action: address(WriteHelloWorld)
execute("Hello Mars"): bytes
result: "Hello World"
overwriteData(resourceID: 0, "Hello World")
confirms whether execution happened by comparing values of resource 0
response: true
Internally this confirmation is being handled by another call to the EnforcementPoint which returns the address of the TextStorage.

5.3.2 Considerations on the Implementation

In order to give an outlook of possible future improvements on the proposed model this final section provides a short overview about techniques which could be applied. If and how the related functionality is going to be included within Solidity is unclear.

Generics / Templates
Introducing templates known from other programming would allow to extend the processRequest function by a generic type T. Consequentially this would result in arbitrarily typed conditions and attributes. Instead the system requires re implementation in case a programmer intends to use attributes of type short. This is an imaginable scenario as his intentions might be based on the consumption of gas the operations require. In addition the Storage model could be simplified by making use of an array of generically typed values. Thus the implementation of multiple different storage contracts for the individual types would not be required. The topic of generics was mentioned its official GitHub page via a feature request.[82] However the general stance of the community is that this feature has a rather low probability of being included in near releases.

Structs
Solidity explicitly states within it FAQ[33] that it currently does not support passing structs through external functions. If that was possible a request could be sent as a struct instead of three single values. This could improve abstraction and allow a better readability of code.

```
1 struct Request {
2 uint userID;
3 uint actionID;
4 uint resourceID;
```

Listing 5.2: Request Struct

6 Evaluation

In order to provide a final evaluation of the implemented prototype different techniques were applied. As it is being implemented in Solidity the thesis provides a set of different Test Cases. The Test Setup including a description of the tested features is included in the first chapter of the evaluation process.second chapter includes an overview of the functionality the prototype provides and compares them with Smart Policies.

6.1 Software Tests

6.1.1 Testing Smart Contracts

The Truffle Suite provides a complete environment for the steps involved in creating, testing and deploying a Smart Contract.[93] Consequentially its features include the setup of a local Test-Blockchain. As a means of interaction it both includes one graphical and one non-graphical interface.[84] Both of them are able to deploy the Smart Contracts on the local Blockchain and to test them based on a provided JavaScript based testing language[84]. This framework allows various means of testing code. All of the thesis' tests were conducted with truffle version v5.0.27. The package provides all the necessary components for running the tests. In addition it should be noted that deploying the contracts to truffle allows making use of its "Clean Room Environment"[83] feature. It allows the evm to revert to a previous state in order to allow test cases to be executed separately from each other and without sharing the same state.

6.1.2 Test Setup and Execution

In total 41 different test cases were conducted based on an own implementation of the abstract contracts and interfaces. This was done in order to prove a complete workflow within the system. Thus the testing included the definition of the previously mentioned contracts "TextEnforcementPoint" and "FourEyesAuthentication". In addition a RequiredAttributesDecisionPoint was implemented who bases its decision on whether a request is executed based on a minimum set of fulfilled threshold of conditions. The reasoning behind this testing approach was to both show that the system is both extendable to more complex scenarios and that the described condition attribute matching can be used as a basic way of expressing access control. As the thesis' included sequence diagram is representing a test case of the system it can be used as a reference. In addition the storage contracts were tested both separately and individually. Its individual tests were conducted on the contract described as "AddressStorage". As it shares the exact

same code structure with the Bytes32Storage both of their correct basic functionality (read, write, overwrite, delete) is covered by unit tests or the full system test. The full system test was conducted by giving a set of two initially unregistered users which were registered to a Four Eyes Authentication Contract during initialization. Then both the authenticated and the unauthenticated users performed basic operations such as performing a request or accessing protected functionality. As the Decision Point included a decide() function which blocked unauthorized users by default all of the unauthenticated users were being denied at this stage. The authenticated user was able to perform a full request including on-chain Enforcement via the thesis provided TextEnforcementPoint. This request altered the state of an attached storage system and therefore proved that the proposed approach works. The DecisionPoint was tested on whether it was able to perform an accurate matching of conditions and attributes and to arrive at correct decision based their combination.

Interactions between Decision Point, PolicyRepository and InformationPoints

```
let initialValues = [
asciiHex("ConditionA"),
asciiHex("ConditionB"),
asciiHex("ConditionC"),
asciiHex("ConditionD"),
asciiHex("ConditionE"),
];
```

Listing 6.1: Initial Test Values

To prove this both an Information Point as well as a the PolicyRepository were initialized with the values mentioned above. Then an authorized user was being used to conduct multiple requests via a loop. As this test was focused on the DecisionPoint itself it directly called its decide function. As expected the system reached a state when the provided amount of attributes did not match all the system's conditions anymore. Consequently the system replied by emitting a deny event. In addition the testing involved two different scenarios both involving the random generation of a variable amount of Information Points initialized with an individual random amount of attributes. The system set the amount of randomly generated Information Points to a range between 1 and 10. Each of the Information Points were provided between 1 and 20 different attributes. In addition the Decision Point's Policy Repository was added a single condition. The test then branched off in two different directions. One test case hid a single fitting attribute within all the randomly generated attributes of the Information Points and another one did not. Both test cases were executed in multiple iterations each single one of them asserting that a Decision Point either finds the attribute and grants or does not find it and denies. These two scenarios were run 50 times in order to ensure that the Decision Point responds as expected. times.

Interactions between DecisionPoint, Enforcement Point, Actions and Storages

As the previous tests confirmed that the correct functioning of the DecisionPoint it

was necessary to test whether a TextEnforcementPoint is able to alter the state of an attached storage by executing an Action referenced within a storage. To to this its attached Storage of type bytes was initialized with the text "NOTHELLOWORLD" at position 0. Then two full system tests were conducted by combining the execution of the previously stated tests for the DecisionPoint with the Enforcement itself. The Enforcement Point then executes a request in order to alter the stored value from "NOTHELLOWORLD" to "Hello World". As this can be reflected by a few lines of code the test's main part follows.

```
1 debugMessage("Starting enforcement...", vbo.CRITICAL);
2 await textEnforcementPoint.request(0, 0); // Request is sent
3 let result = await textStorage.retrieveData.call(0); // Retrieval in order to compare
      with expected value
4 let data = result[0];
5 debugMessage("Stored afterwards: " + data, vbo.CRITICAL);
6 assert.equal(data, asciiHexNoPad("Hello World"), "Hello World was not written!"); //
      Assertion that text was altered to "Hello World"
```

Listing 6.2: Full System Test Grant & Enforcement

The example above shows the assertions in case of a Decision Point's grant. It executes the request, then retrieves the storage's data and finally compares it to "Hello World". Its negative formulation correctly yielded a "deny" response from the Enforcement Point. Then the storage's state was asserted to be equal to its state before in order to prove that it does not enforce in case it doesn't receive a grant.

In addition the tests included unit tests proving that the storage's are able to execute their implemented methods of retrieval and deletion. In addition they are being covered by the Full System Tests. In addition the functioning of the Protected Contract was examined. One test case includes a FourEyesAuthenticationPoint who tries to register a user to a storage but fails as it is not being linked. Then the link is established and the registration succeeds.

The test results directly confirm the system's extendability. During the different full system tests the expected notifications were emitted by each of the individual components. Thus the system was successfully implemented according to its requirements specification.

6.2 Comparison With Smart Policies

As Blockchain-based Access Control served as the primary point of reference during the implementation of the thesis' prototype a comparison between both projects is necessary. Instead of providing two competing solutions each of them provides a different approach to a shared problem. As the thesis is approaching the subject from the perspective of providing a reusable framework for future implementations it to focus on specific aspects. Therefore the following table only serves as a rough comparison between both these projects.

	Smart Policies	Implementation
XACML		
X1) Includes Full On-Chain Enforcement	-	**+**
X2) Supports On-Chain Policy Decisions	+	+
X3) Supports Dynamic Addition of Information Points	$\sim$	**+**
X4) Supports Complex XACML Policies	+	-
Utility		
U1) Includes Basic Authentication Contract	-	+
U2) Supports Resource Abstraction	-	+
U3) Uses Events to notify Subscribers	+	+
U4) Allows public Auditability	+	+
Privacy		
P1) Can be deployed on Quorum	**-**	+
P2) Includes Off-Chain Enforcement Point	**+**	-
Extendability		
E1) Promotes Reusability by Design	**-**	**+**
E2) Separation between Private and Public Enforcement	**-**	**+**

Table 6.1: Comparison between Smart Policies and Implementation

As the table shows there are many similarities between the functionality both systems provide. However the proposed system is based on another principle and philosophy. Its intention is to provide a reusable code which can be published and improved iteratively with the help of the OpenSource community. Thus its main benefits lie in the factors E1) and E2). The first of them specifically refers to the fact that Smart Policies make use of off-chain compilation and a Java-based client for the Enforcement. As this forces a future developer to both be knowledgeable in Java and Solidity development the proposed system potentially provides an easier access. In addition the introduction of compilation requires a programmer to understand the internal workings of the compiler system itself. The thesis proposal circumvents this by basing its system solely on Smart Contracts. However Smart Policies are able to compile multiple different executable Smart Contracts. This can prove to be valuable as both systems are not mutually exclusive. Another benefit of the proposed system is that it allows full Blockchain Enforcement maximizing the system's auditability (X1). However other than the Smart Policy-based system it does not provide an actual client for off-chain enforcement. Instead it only serves as an adapter and can fully be deployed as private Smart Contracts within Quorum. This is not possible for the other system as its off-chain enforcement is handled by a Java application. Still the Smart Policy system is capable of private data transfer as it is entirely focused on its off-chain Enforcement. X3) Specifically refers to the fact that Smart Policies use hard coded addresses of their Information Points which are being encoded into the Smart Policies during their compilation. This makes it questionable

whether they are capable of achieving a dynamic addition of Information Points or not.

7 Conclusion and Future Work

7.1 Conclusion

Answering the initially stated questions the system showed that the current challenges regarding access control are strongly linked to the issues of data privacy. As the degree of data privacy decreases with an increasing amount of auditability and verifiability. Consequently each decision regarding access control has to be evaluated more carefully within a Blockchain context. However the proposed system also showed that the Blockchain's inherent properties can be leveraged in order to present a fully auditable system instead. Further it provided a proof-of-concept implementation showing a possible approach on how to both model and implement a functioning access control system. This way it expands on the previous mostly modifier-based projects and variants. As the thesis combines advantages from Smart Policies with the flexibility of OpenZeppelin's contracts it can be applied to many different scenarios. While the thesis only provided theoretical tests future implementations need to determine its final impact on the community. In addition the thesis provided privacy considerations and a basic authentication functionality. As the different design decisions have been laid out it is being part of very few published projects in this area of research. This is being underlined by the fact that it includes its full modelling process including multiple documented parts of source code. With the hope of providing both a common point of discussion and a valuable contribution to the OpenSource community the thesis therefore decides to publish its source code via GitHub. Thus its future is being decided by factors such as whether it can reach wide-spread adaptation and whether is being recognized by the OpenSource community. If it achieves this it provides a tested base implementation which can be improved iteratively. The programming language Solidity itself was a really small project.

7.2 Future Work

The presented system's main advantage lies in its extendability and the fact that its model can be applied recursively. Because the As as a whole introduces a new perspective to solving access control problems by its inclusion of multiple storage contracts and their corresponding extendable actions. There are two possible future additions to Solidity which could simplify the model drastically.

On the one hand the introduction of Template Metaprogramming or Generics (such as in C++) can lead to a more all around implementation of storage. On the other hand

the current version of Solidity doesn't support passing Structs between contracts. As Structs could both improve code readability and maintainability greatly this feature is currently being part of a suite of experimental features. Further the system is not optimized for gas usage yet. While the reference implementation doesn't make use of string comparisons there are possibly different optimization strategies for implementing a more efficient decision process or even the storage contracts. However as most of the implemented contracts are inheritable their functionality can be overridden in order to provide more efficient algorithms.

Its flexibility allows for a wide variety of potential use cases. More specifically it is imaginable that the project MedRec which stores SQL statements inside Smart Contracts could benefit from the proposed systems. The SQL queries could be stored in an Indexed Storage. Afterwards actions could be defined which append or remove single SQL operations to the queries whenever they are being executed. The thesis system could then provide a completely auditable overview about the evolution of SQL queries by the execution of consecutive actions. This information could be aggregated and displayed in a specifically developed front end in order to directly monitor access requests and their handling. More generally the implementation of a front end can be rather useful by itself. It could include features such as off-chain policy conflict resolution of formulated policies as this might be too expensive in terms of gas cost.

Further there is currently a lot of research regarding Zero-Knowledge proofs in the context of authentication. Because the thesis only provided a very conservative contract for authentication future advances could lead to a whole new perspective on the current understanding of the subject. While this remains an open question it still proves that the thesis findings serve as a scientific contribution in a rapidly evolving research area.

List of Figures

List of Tables

List of Listings

Bibliography

[1] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. "A survey of attacks on Ethereum smart contracts (SoK)." In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 10204 LNCS. Springer, 2017, pp. 164–186. ISBN: 9783662544549. DOI: 10.1007/978-3-662-54455-6_8.

[2] Yu Nandar Aung and Thitinan Tantidham. "Review of Ethereum: Smart home case study." In: *2017 2nd International Conference on Information Technology (INCIT)*. IEEE. 2017, pp. 1–4.

[3] *authentication*. https://www.merriam-webster.com/dictionary/authentication. Accessed: 12/12/2019.

[4] A. Azaria et al. "MedRec: Using Blockchain for Medical Data Access and Permission Management." In: *2016 2nd International Conference on Open and Big Data (OBD)*. Aug. 2016, pp. 25–30. DOI: 10.1109/OBD.2016.11.

[5] L M Bach, Branko Mihaljevic, and Mario Zagar. "Comparative analysis of blockchain consensus algorithms." In: *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2018 - Proceedings*. 2018, pp. 1545–1550. ISBN: 9789532330977. DOI: 10.23919/MIPRO.2018.8400278.

[6] S Balaji and M Sundararajan Murugaiyan. "Waterfall vs. V-Model vs. Agile: A comparative study on SDLC." In: *International Journal of Information Technology and Business Management* 2.1 (2012), pp. 26–30.

[7] Alex Biryukov and Dmitry Khovratovich. "Related-key cryptanalysis of the full AES-192 and AES-256." In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2009, pp. 1–18.

[8] Alex Biryukov, Dmitry Khovratovich, and Ivica Nikolić. "Distinguisher and related-key attack on the full AES-256." In: *Annual International Cryptology Conference*. Springer. 2009, pp. 231–249.

[9] *Block size limit controversy*. https://en.bitcoin.it/wiki/Block_size_limit_controversy. Accessed: 11/12/2019.

[10] *Blockchain Cryptocurrency Regulation 2020*. https://www.globallegalinsights.com/practice-areas/blockchain-laws-and-regulations/china. Accessed: 12/12/2019.

[11] *Blockchain Size*. https://www.blockchain.com/de/charts/blocks-size. Accessed: 11/12/2019.

[12] Chiara Bodei et al. "Static analysis of processes for no read-up and no write-down." In: *International Conference on Foundations of Software Science and Computation Structure*. Springer. 1999, pp. 120–134.

[13] Chiara Bodei et al. "Static analysis of processes for no read-up and no write-down." In: *International Conference on Foundations of Software Science and Computation Structure*. Springer. 1999, pp. 120–134.

[14] David FC Brewer and Micheal J Nash. "The chinese wall security policy." In: *null*. IEEE. 1989, p. 206.

[15] Kin-Ching Chan and S-HG Chan. "Key management approaches to offer data confidentiality for secure multicast." In: *IEEE network* 17.5 (2003), pp. 30–39.

[16] Sourabh Chandra et al. "A comparative survey of symmetric and asymmetric key cryptography." In: *2014 International Conference on Electronics, Communication and Computational Engineering (ICECCE)*. IEEE. 2014, pp. 83–93.

[17] Guanling Chen and Robert S. Gray. "Simulating Non-scanning Worms on Peer-to-peer Networks." In: *Proceedings of the 1st International Conference on Scalable Information Systems*. InfoScale '06. Hong Kong: ACM, 2006. ISBN: 1-59593-428-6. DOI: 10.1145/1146847.1146876.

[18] Konstantinos Christidis and Michael Devetsikiotis. *Blockchains and Smart Contracts for the Internet of Things*. 2016. DOI: 10.1109/ACCESS.2016.2566339.

[19] *Connecting to the network*. https://github.com/ethereum/go-ethereum/wiki/Connecting-to-the-network. Accessed: 12/12/2019.

[20] *Contracts*. https://solidity.readthedocs.io/en/v0.4.24/contracts.html. Accessed: 12/12/2019.

[21] Jason Paul Cruz, Yuichi Kaji, and Naoto Yanai. "RBAC-SC: Role-based access control using smart contract." In: *IEEE Access* 6 (2018), pp. 12240–12251. ISSN: 21693536. DOI: 10.1109/ACCESS.2018.2812844.

[22] Ivan Bjerre Damgård. "Collision free hash functions and public key signature schemes." In: *Workshop on the Theory and Application of of Cryptographic Techniques*. Springer. 1987, pp. 203–216.

[23] Chris Dannen. *Introducing Ethereum and Solidity*. Springer, 2017.

[24] Jan De Clercq. "Single Sign-On Architectures." In: *Infrastructure Security*. Ed. by George Davida, Yair Frankel, and Owen Rees. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 40–58. ISBN: 978-3-540-45831-9.

[25] 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)*. July 2018, pp. 1379–1386. DOI: 10.1109/Cybermatics_2018.2018.00237.

[26] Damiano Di Francesco Maesa, Paolo Mori, and Laura Ricci. "Blockchain Based Access Control Services." In: *Proceedings - IEEE 2018 International Congress on Cybermatics: 2018 IEEE Conferences on Internet of Things, Green Computing and Communications, Cyber, Physical and Social Computing, Smart Data, Blockchain, Computer and Information Technology, iThings/Gree*. 2018, pp. 1379–1386. ISBN: 9781538679753. DOI: 10.1109/Cybermatics_2018.2018.00237.

[27] Thaddeus Dryja. "Utreexo: A dynamic hash-based accumulator optimized for the Bitcoin UTXO set." In: ().

[28] Jacob Eberhardt and Stefan Tai. "On or off the blockchain? Insights on off-chaining computation and data." In: *European Conference on Service-Oriented and Cloud Computing*. Springer. 2017, pp. 3–15.

[29] Jacob Eberhardt and Stefan Tai. "On or off the blockchain? Insights on off-chaining computation and data." In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 10465 LNCS. 2017, pp. 3–15. ISBN: 9783319672618. DOI: 10.1007/978-3-319-67262-5_1.

[30] *Etherscan*. https://etherscan.io/. Accessed: 12/12/2019.

[31] *Example Flow*. https://www.oauth.com/oauth2-servers/server-side-apps/example-flow/. Accessed: 12/12/2019.

[32] Michael Fleder, Michael S Kester, and Sudeep Pillai. "Bitcoin transaction graph analysis." In: *arXiv preprint arXiv:1502.01657* (2015).

[33] *"Frequently Asked Questions"*. https://solidity.readthedocs.io/en/v0.4.24/frequently-asked-questions.html. Accessed: 12/12/2019.

[34] Michael Fröwis, Andreas Fuchs, and Rainer Böhme. "Detecting token systems on ethereum." In: *International Conference on Financial Cryptography and Data Security*. Springer. 2019, pp. 93–112.

[35] David Gabay, Mumin Cebe, and Kemal Akkaya. "On the overhead of using zero-knowledge proofs for electric vehicle authentication: poster." In: *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*. ACM. 2019, pp. 347–348.

[36] *Geth*. https://github.com/ethereum/go-ethereum. Accessed: 12/12/2019.

[37] Martin Glinz. "Rethinking the notion of non-functional requirements." In: *Proc. Third World Congress for Software Quality*. Vol. 2. 2005, pp. 55–64.

[38] Simon Godik and Tim Moses. *eXtensible Access Control Markup Language (XACML) Version 1.1*. 2003.

[39] Lisandro Zambenedetti Granville et al. "Managing computer networks using peer-to-peer technologies." In: *IEEE Communications Magazine* 43.10 (2005), pp. 62–68.

[40] Rui Guo et al. "Secure attribute-based signature scheme with multiple authorities for blockchain in electronic health records systems." In: *IEEE Access* 6 (2018), pp. 11676–11686.

[41] Peter Hegedus. "Towards analyzing the complexity landscape of solidity based ethereum smart contracts." In: *Technologies* 7.1 (2019), p. 6.

[42] *How Tessera works*. `https://docs.goquorum.com/en/latest/Privacy/Tessera/HowTesseraWorks/`. Accessed: 12/12/2019.

[43] Martin Husák et al. "HTTPS traffic analysis and client identification using passive SSL/TLS fingerprinting." In: *EURASIP Journal on Information Security* 2016.1 (2016), p. 6.

[44] Xin Jin, Ram Krishnan, and Ravi Sandhu. "A Unified Attribute-Based Access Control Model Covering DAC, MAC and RBAC." In: *Data and Applications Security and Privacy XXVI*. Ed. by Nora Cuppens-Boulahia, Frédéric Cuppens, and Joaquin Garcia-Alfaro. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 41–55. ISBN: 978-3-642-31540-4.

[45] Jens-Peter Kaps and Berk Sunar. "Energy comparison of AES and SHA-1 for ubiquitous computing." In: *International Conference on Embedded and Ubiquitous Computing*. Springer. 2006, pp. 372–381.

[46] Volker Kessler. "On the Chinese wall model." In: *European Symposium on Research in Computer Security*. Springer. 1992, pp. 41–54.

[47] Jae-Jung Kim and Seng-Phil Hong. "A method of risk assessment for multi-factor authentication." In: *Journal of Information Processing Systems* 7.1 (2011), pp. 187–198.

[48] Ahmed Kosba et al. "Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts." In: *Proceedings - 2016 IEEE Symposium on Security and Privacy, SP 2016*. 2016, pp. 839–858. ISBN: 9781509008247. DOI: 10.1109/SP.2016.55.

[49] David M. Kristol. "HTTP Cookies: Standards, Privacy, and Politics." In: *ACM Trans. Internet Technol.* 1.2 (Nov. 2001), pp. 151–198. ISSN: 1533-5399. DOI: 10.1145/502152.502153.

[50] *Laravel - Controllers*. `http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm`. 12/12/2019.

[51] Chen Liyan. "Application research of using design pattern to improve layered architecture." In: *2009 IITA International Conference on Control, Automation and Systems Engineering (case 2009)*. IEEE. 2009, pp. 303–306.

[52] Pericles Loucopoulos and Vassilios Karakostas. *System requirements engineering*. McGraw-Hill, Inc., 1995.

[53] Sihua Ma et al. "Using blockchain to build decentralized access control in a peer-to-peer e-learning platform." PhD thesis. University of Saskatchewan, 2018.

[54] Rahat Masood, Muhammad Awais Shibli, Muhammad Bilal, et al. "Usage control model specification in XACML policy language." In: *IFIP International Conference on Computer Information Systems and Industrial Management*. Springer. 2012, pp. 68–79.

[55] Anastasia Mavridou and Aron Laszka. "Designing secure ethereum smart contracts: A finite state machine based approach." In: *International Conference on Financial Cryptography and Data Security*. Springer. 2018, pp. 523–540.

[56] Satoshi Nakamoto. "Bitcoin whitepaper." In: *URL: https://bitcoin. org/bitcoin. pdf-(: 17.07. 2019)* (2008).

[57] Christopher Natoli and Vincent Gramoli. "The blockchain anomaly." In: *2016 IEEE 15th International Symposium on Network Computing and Applications (NCA)*. IEEE. 2016, pp. 310–317.

[58] *"Node Casbin"*. `https://github.com/casbin/node-casbin`.

[59] Svein Ølnes, Jolien Ubacht, and Marijn Janssen. "Blockchain in government: Benefits and implications of distributed ledger technology for information sharing." In: *Government Information Quarterly* 34.3 (2017), pp. 355–364. ISSN: 0740624X. DOI: 10.1016/j.giq.2017.09.007.

[60] Haroon Shakirat Oluwatosin. "Client-server model." In: *IOSRJ Comput. Eng* 16.1 (2014), pp. 2278–8727.

[61] Shyue Ping Ong et al. "The Materials Application Programming Interface (API): A simple, flexible and efficient API for materials data based on REpresentational State Transfer (REST) principles." In: *Computational Materials Science* 97 (2015), pp. 209–215.

[62] *OpenZeppelin*. `https://github.com/OpenZeppelin/openzeppelin-contracts/tree/master/contracts`. Accessed: 12/12/2019.

[63] Siddika Berna Ors et al. "Power-Analysis Attack on an ASIC AES implementation." In: *International Conference on Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004.* Vol. 2. IEEE. 2004, pp. 546–552.

[64] Aafaf Ouaddah, Anas Abou Elkalam, and Abdellah 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. ISSN: 19390122. DOI: 10.1002/sec.1748.

[65] Venkata N Padmanabhan and Jeffrey C Mogul. "Improving HTTP latency." In: *Computer Networks and ISDN Systems* 28.1-2 (1995), pp. 25–35.

[66] Donn B Parker. "Toward a new framework for information security?" In: *Computer security handbook* (2012), pp. 3–1.

[67] Sarvar Patel. *Method for updating secret shared data in a wireless communication system*. US Patent 6,243,811. June 2001.

[68] Michael Piatek, Tadayoshi Kohno, and Arvind Krishnamurthy. "Challenges and directions for monitoring P2P file sharing networks, or, why my printer received a DMCA takedown notice." In: *HotSec*. 2008.

[69] *Proof of Stake FAQ*. `https://github.com/ethereum/wiki/wiki/Proof-of-Stake-FAQ`. Accessed: 12/12/2019.

[70] *Raft-based consensus for Ethereum/Quorum*. `https://github.com/jpmorganchase/quorum/blob/master/docs/Consensus/raft.md`. Accessed: 12/12/2019.

[71] Dilli Ravilla and Chandra Shekar Reddy Putta. "Implementation of HMAC-SHA256 algorithm for hybrid routing protocols in MANETs." In: *2015 International Conference on Electronic Design, Computer Networks & Automated Verification (EDCAV)*. IEEE. 2015, pp. 154–159.

[72] *RBAC-SC*. `https://github.com/jpmcruz/RBAC-SC/blob/master/RBAC-SC`. Accessed: 12/12/2019.

[73] Mark Richards. *Software architecture patterns*. O'Reilly Media, Incorporated, 2015.

[74] Carlos Rodrigues, José Afonso, and Paulo Tomé. "Mobile application webservice performance analysis: Restful services with json and xml." In: *International Conference on ENTERprise Information Systems*. Springer. 2011, pp. 162–169.

[75] Sara Rouhani and Ralph Deters. "Performance analysis of ethereum transactions in private blockchain." In: *2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS)*. IEEE. 2017, pp. 70–74.

[76] Hamza Es-Samaali, Aissam Outchakoucht, and Jean Philippe Leroy. "A blockchain-based access control for big data." In: *International Journal of Computer Networks and Communications Security* 5.7 (2017), p. 137.

[77] David Cerezo Sánchez. "Raziel: Private and Verifiable Smart Contracts on Blockchains." In: *arXiv preprint arXiv:1807.09484* (2018). arXiv: `1807.09484`.

[78] John C Schmitt and Dale R Setlak. *Access control system including fingerprint sensor enrollment and associated methods*. US Patent 5,903,225. May 1999.

[79] *"Signal and PGP are broken"*. `https://twitter.com/snowden/status/878683882081722369?lang=de`.

[80] Dan Simon, Bernard Aboba, Ryan Hurst, et al. "The EAP-TLS authentication protocol." In: *RFC5216, IETF, March* (2008), p. 1.

[81] Nicolas Sklavos and Odysseas Koufopavlou. "On the hardware implementations of the SHA-2 (256, 384, 512) hash functions." In: *Proceedings of the 2003 International Symposium on Circuits and Systems, 2003. ISCAS'03.* Vol. 5. IEEE. 2003, pp. V–V.

[82] *"templates / generics"*. `https://github.com/ethereum/solidity/issues/869`.

[83] *"Testing Your Contracts"*. `https://www.trufflesuite.com/docs/truffle/testing/testing-your-contracts`.

[84] *Truffle Quickstart*. https://www.trufflesuite.com/docs/truffle/quickstart. Accessed: 12/12/2019.

[85] Steve Vinoski. "Restful web services development checklist." In: *IEEE Internet Computing* 12.6 (2008), pp. 96–95.

[86] Vitalik Buterin. *Privacy on the Blockchain*. https://blog.ethereum.org/2016/01/15/privacy-on-the-blockchain/. Accessed: 12/12/2019.

[87] Rossouw Von Solms and Basie Von Solms. "From policies to culture." In: *Computers & security* 23.4 (2004), pp. 275–279.

[88] Dejan Vujičić, Dijana Jagodić, and Siniša Ranđić. "Blockchain technology, bitcoin, and Ethereum: A brief overview." In: *2018 17th International Symposium INFOTEH-JAHORINA (INFOTEH)*. IEEE. 2018, pp. 1–6.

[89] Xunhua Wang et al. "Enabling secure on-line DNS dynamic update." In: *Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00)*. IEEE. 2000, pp. 52–58.

[90] *Which OAuth 2.0 Flow Should I Use?* https://auth0.com/docs/api-auth/which-oauth-flow-to-use. Accessed: 12/12/2019.

[91] Erik Wilde. "Putting things to REST." In: (2007).

[92] Gavin Wood et al. "Ethereum: A secure decentralised generalised transaction ledger." In: *Ethereum project yellow paper* 151.2014 (2014), pp. 1–32.

[93] *"Writing Tests in JavaScript"*. https://www.trufflesuite.com/docs/truffle/testing/writing-tests-in-javascript. Accessed: 12/12/2019.

[94] Karl Wüst and Arthur Gervais. "Do you need a Blockchain?" In: *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*. IEEE. 2018, pp. 45–54.

[95] Haidong Xia and José Carlos Brustoloni. "Hardening Web Browsers Against Man-in-the-middle and Eavesdropping Attacks." In: *Proceedings of the 14th International Conference on World Wide Web*. WWW '05. Chiba, Japan: ACM, 2005, pp. 489–498. ISBN: 1-59593-046-9. DOI: 10.1145/1060745.1060817.

[96] Rui Yuan et al. "ShadowEth: Private Smart Contract on Public Blockchain." In: *Journal of Computer Science and Technology* 33.3 (2018), pp. 542–556. ISSN: 18604749. DOI: 10.1007/s11390-018-1839-y.

[97] Rui Zhang, Rui Xue, and Ling Liu. "Security and Privacy on Blockchain." In: *ACM Comput. Surv.* 52.3 (July 2019), 51:1–51:34. ISSN: 0360-0300. DOI: 10.1145/3316481.

[98] Lidong Zhou, Fred B. Schneider, and Robbert Van Renesse. "COCA: A Secure Distributed Online Certification Authority." In: *ACM Trans. Comput. Syst.* 20.4 (Nov. 2002), pp. 329–368. ISSN: 0734-2071. DOI: 10.1145/571637.571638.

[99] Aviv Zohar. "Bitcoin: under the hood." In: *Communications of the ACM* 58.9 (2015), pp. 104–113.