UNIVERSITY OF TWENTE

MASTER THESIS

Towards Updatable Smart Contracts

Author:

F.W.C. BLOO
f.w.c.bloo@alumnus.utwente.nl

Supervisors:

Dr.ir. J.M. MOONEN
Dr. N. SIKKEL
(University of Twente)

F.H. TAN MSc MISM
(Northwave B.V.)

Business Information Technology - Track: Business Analytics
Faculty of Electrical Engineering, Mathematics and Computer Science

October 19, 2018

Abstract

In recent years, research institutes and other organisations have shown an increased interest in blockchain technology. A blockchain is a distributed append-only database that ensures a high degree of availability, integrity, and transparency. Multiple blockchain implementations, for example Ethereum, support the storage and execution of executable code, called smart contracts. Smart contracts on the Ethereum blockchain are public and can become the target of an attack. The Decentralised Autonomous Organisation and the Parity Multisig Wallets both became the victim of an attack and lost respectively $50 million and $150 million. Due to the immutability of smart contracts, security updates and new functionalities can currently not be implemented. This poses a major risk for organisations that want to adopt secure smart contracts. From this, the main goal of the study is derived: to investigate whether a smart contract can be updated in a decentralised manner.

We adopt a Design Science approach to create an updatable smart contract. A solution design involves a technical aspect to bypass the immutability of a smart contract and a decision-making process to reach a consensus on an update amongst the (anonymous) participants. This thesis presents a design that bypasses the immutability by adopting a proxy smart contract which redirects incoming calls to the most recent version of the smart contract. The decision-making process that we present, is extracted from four illustrative case studies. The aim for each of these case studies is to a develop decision-making process that provides each participant a fair stake. In this study, a fair stake is implemented as the value at risk: the stake in the process is relative to the value stored on the smart contract. The motivation for this approach is that participants who store assets with a high value on the smart contract can lose more value in case of a breach, and are thus more concerned. As the results show, the applicability and feasibility of this concept are limited for smart contracts. First, it is not applicable when the value at risk cannot be determined, which is the case when it stores unique objects (i.e. non-fungible tokens). Second, boundary cases can occur because centralisation of the voting power cannot be prevented in all situations. Third, participants might lack knowledge of the programming language in order to make a well-informed decision. Last, the decision-making process is time consuming. Additional limitations as a result of the technical implementation are: (I) The big bang adoption of an update might result in invalid transactions. (II) The update requires one block downtime. (III) The source code of an update must be publicly visible. (IV) A hard fork might result in conflicting instances. The results are validated by means of interviews with four industry experts.

An updatable smart contract is one approach of improving smart contract security. This study investigates whether a solution can be designed that allows to update smart contracts in a decentralised manner. The design that we present in this study is not viable and should not be implemented by industry. A promising concept that could be studied for an improved design is delegated voting, this allows participants without knowledge of the programming language to delegate their vote to another participant. A second research topic is to investigate a viable solution to implement updates that patch critical vulnerabilities.

Acknowledgements

This thesis marks the end of my period as a student at the University of Twente. The last months have not only been exciting, but also very challenging. I am thankful for all the support that I received during the research. First of all, I am grateful that Hans Moonen and Klaas Sikkel both agreed to supervise my graduation project. Despite busy schedules, they were able to review my thesis many times in order to provide feedback. Their guidance, experience and insights were valuable to me and helped me to improve my research. Furthermore, I would like to thank Northwave and in particular Fook Hwa Tan for offering me a graduation project. With many great colleagues, Northwave has been a good and fun place to write my thesis. Last, I would like to thank my family and friends for their support during the entire project.

Contents

List of Figures

List of Tables

List of Abbreviations

ABI	**A**pplication **B**inary **I**nterface
BIP	**B**itcoin **I**mprovement **P**roposal
DAO	**D**ecentralised **A**utonomous **O**rganisation
dApp	**d**ecentralised **App**lication
ERC	**E**thereum **R**equest for **C**omments
EVM	**E**thereum **V**irtual **M**achine
ICO	**I**nitial **C**oin **O**ffering
IoT	**I**nternet **o**f **T**hings
PoS	**P**roof **o**f **S**take
PoW	**P**roof **o**f **W**ork

Chapter 1

Introduction

1.1 Introduction

In recent years, blockchain has emerged as an innovative technology with the potential to change the way society, politics and businesses interact. The technology is evolving at a high pace and is far beyond its first application in the crypto currency Bitcoin. Blockchains can be thought of as append-only transactional databases, distributed over a decentralised peer-to-peer network, thereby facilitating trust-less transactions without the need for a trusted third party. Participants are not required to trust each other to interact: the transactions are verified by a set of algorithms, without the interference of a human or a central authority. A blockchain, sometimes referred to as a distributed ledger, ensures a high degree of availability and integrity of the stored data; tampering with data is nearly impossible.

A feature of interest that a number of blockchain platforms started adopting is the support for executable code. The storage and execution of executable code on top of a blockchain allows to develop a wide range of decentralised applications (dApps). The executable code, hereinafter referred to as a smart contract, is a software protocol that is designed to enforce, facilitate and verify traditional contracts in a digital manner. Its design allows for the automatic execution of transactions without any interference from the outside world. The integrity and availability of smart contracts are ensured as they are stored and executed on top of a blockchain, hence transactions executed by smart contracts are traceable and irreversible.

A smart contract runs autonomous on a blockchain and becomes nearly impossible to alter once it is deployed, the code is considered immutable. The immutability provides a key element for the digitalisation of traditional contracts but is considered a double-edged sword. There is no method to implement a solution for a defect in a smart contract, the defect is irreversible and permanent. Considering that a compelling number of crypto currencies are operated by smart contracts, such a feature is appealing.

Two incidents that exploited vulnerabilities in smart contracts resulted in significant financial losses of crypto currencies. During the Decentralised Autonomous Organisation (DAO) attack, a malicious user was able to withdraw around $50 million from a smart contract that was used to store all tokens of a crowdfunding project (Popper, 2016). A few days prior to this incident, one of the core developers announced that a recursive call bug was found in the DAO software and assured that funds were not at risk (Taul, 2016). The assurance turned out to be false; a malicious user was able to approach the smart contract recursively in such a way that he was able to drain money into another contract. In another recent incident, a malicious user gained control over all Parity Multisig wallets which were used to store Ether and other types of digital currencies (Parity Technologies, 2018). The root cause of

this incident was a misconfiguration in a single smart contract that acted as a library to the wallets (Parity Technologies, 2017). The malicious user was able to gain access to specific functions of the smart contract, after which the user called the self-destruct function (Etherscan, 2017a,b). All types of crypto currencies stored on the Parity Multisig wallets got locked forever. In a statement of the organisation behind the Parity Multisig wallets, it was confirmed that a total amount of 513,774.16 Ether, with a total value of $150 million at that time, was stored on these wallets.

1.2 Problem Statement

A common characteristic of the DAO and Parity Multisig wallet attacks, as described in the previous section, is that both attacks exploited vulnerabilities of Ethereum smart contracts. This seems to be the tip of an iceberg. A recent study that analysed deployed smart contracts on the Ethereum blockchain on four different types of vulnerabilities, showed that 8,833 out of the 19,366 analysed smart contracts contain at least one of the vulnerabilities (Luu et al., 2016). Another study that analysed smart contracts for other types of vulnerabilities concluded that 4,905 Ether, worth $2.6 million, is stored on vulnerable smart contracts (Nikolic et al., 2018).

The maturity of smart contracts is not yet at the desired level; the slightest mistake in the code can have disastrous implications. The incidents from the previous section indicate that even experienced developers face difficulties to write secure code. Organisations aiming to adopt smart contracts need to understand the associated risks for a successful and secure implementation.

From a security perspective, the immutability of a smart contract ensures organisations that, given a state, a smart contract will behave in a consistent manner; no one is able to alter the code or to interfere with its execution. At the same time, the immutability results in the fact that the functionality provided by the smart contract cannot be changed and that solutions to patch security vulnerabilities can thus not be implemented. The importance of smart contract security has gained attention from academics in light of the recent attacks. Studies to date tend to focus on describing and detecting security vulnerabilities rather than providing secure smart contract development methods. Furthermore, these studies fail to provide methods to patch vulnerabilities found in deployed smart contracts.

Given the financial value that smart contracts can process, there is a need for smart contracts that can be updated in order to adapt to the current needs and desired level of security. Considering that the entities that use a smart contract do not necessarily trust each other, no single organisation should have full control to implement an update.

1.3 Research Questions

The objective of this is to investigate whether smart contracts can be updated in a decentralised manner. The answer to the main research question should be a validated artefact.

Research Question: *How could smart contracts be updated in a decentralised manner?*

The research question can be decomposed into the following subquestions. Together, the answers to these sub-questions form the answer to the main Research Question of this study.

Subquestion 1: *What is the current state of the art of blockchain technology?*

Subquestion 2: *What is the current state of the art of smart contracts?*

Subquestion 3: *What are existing security vulnerabilities of smart contracts?*

Subquestion 4: *What are design requirements for smart contracts that can be updated in a decentralised manner?*

Subquestion 5: *What is a design for smart contracts that can be updated in a decentralised manner?*

Subquestion 6: *Does the designed artefact work as desired?*

The results of this study contribute to the maturity of smart contracts and it provides organisations with a method to adopt smart contracts in a more secure manner. Additionally, this study presents an overview of existing vulnerabilities in smart contracts in order to equip developers and organisations with knowledge to mitigate these vulnerabilities.

1.4 Research Design

This study adopts the Design Science in Information Systems Research methodology by Hevner et al. (2004) in order to provide an answer to the main research question. While the goal of the study is to design an artefact, following this research methodology will result in a better understanding of the problem domain and its solution. Wieringa (2014) provides a detailed description of applying design science methodology in a research. A design science research can be seen as a cycle which is part of a larger cycle, the engineering cycle. Consequently, the design science cycle consists of the following tasks:

1. Problem investigation
2. Treatment design
3. Treatment validation
4. Treatment implementation
5. Implementation evaluation

The starting point in the cycle is to get a complete understanding of the problem by identifying, describing and evaluating the problem. The goal is to "investigate an improvement problem *before* an artefact is designed and when no requirements for an artefact have been identified yet" (Wieringa, 2014). The answers to the first three sub-questions will provide an understanding of the problem by giving an overview of the current state of the art of blockchain, smart contracts and its security vulnerabilities.

Once the problem investigation is finished, existing treatments and necessary design requirements for the artefact are researched. Following this, the artefact will be designed by means of a decision-making process design and a technical design. The decision-making process design allows to identify the opportunities and limitations that need to be considered for the technical design. The design for the decision-making process will be developed by generalising a model from four illustrative case studies.

After the artefact has been designed, it requires verification and validation to predict its behaviour when it is implemented in the context of the problem. The defined requirements for a solution design will be used as assessment criteria during the verification of the designed artefact. Interviews with experts will be conducted in order to validate the solution design. Due to practical reasons and time constraints, this thesis does not engage with a treatment implementation and implementation evaluation as specified in the design science cycle.

1.4.1 Literature Studies

Multiple literature studies will be conducted in order to answer the sub-questions of this study. The first step is to retrieve academic articles from the academic databases IEEE, ACM, Scopus and Web of Science. The articles found are first filtered based on the title, abstract and keywords, to determine their relevance for this study. The remaining articles after this selection are considered relevant and will be read in full detail. Next to reading the full text, this study adopts backward reference searching to find relevant articles identified by other researchers. These articles go through a similar selection as the articles initially found in the academic databases.

As the developments in blockchain technology and smart contracts advance at a high pace, the assumption that academic sources are not up to date with the newest developments is considered realistic. Therefore, non-academic sources will be consulted next to academic literature. After identifying literature from academic sources, non-academic online sources such as blog posts, Google search engine, and official documentation, are consulted in order to enrich information from academic sources. Although these sources are not validated by academic research, the transparent nature of blockchain allows to verify the information without requiring a comprehensive research.

1.5 Scope

The main objective of this study is to investigate whether a smart contract can be updated in a decentralised manner. The language in which a smart contract is programmed depends on the blockchain that will be used. Albeit the fact that the concept of smart contracts is adopted by a wide range of blockchain platforms, this thesis solely focusses on the Ethereum blockchain. Ethereum is currently a prominent

blockchain that supports smart contracts. A recent study indicated that 1082 start-ups relied on the Ethereum blockchain and smart contracts to raise capital in 2017 (Fenu et al., 2018). At the time of writing, August 2018, 87.6% of the projects listed on Coinmarketcap run on the Ethereum blockchain (CoinMarketCap, 2018). As a result of solely focussing on the Ethereum blockchain, the solution design needs to consider the associated key characteristics of this type of blockchain. These key characteristics include a public environment and anonymous use.

Smart contracts for the Ethereum blockchain can be written in different programming languages (Ethereum Foundation, 2018j). Before being able to deploy smart contracts in one of these languages, they are first compiled to Ethereum Virtual Machine (EVM) code, which is the actual code stored and executed on the Ethereum blockchain. Two low-level programming languages to write a smart contract are LLL and Serpent. The third and one of the most popular languages, due to its high-level approach, is Solidity. As a high-level language that directly compiles to EVM code and the strong similarities with JavaScript, Solidity is a popular language within the community. Numerous examples and an extensive documentation are available. It is therefore that this study will only consider the smart contracts programmed in Solidity. This does not imply that high-level concepts the designed artefact in this study cannot be applied to smart contracts in other languages or blockchain platforms.

The artefact that will be designed during this study will contain a technical element and a governance element. The technical element is required to understand how the immutability of smart contracts can be bypassed and to understand the process of an update. The governance element is focussed on a decision-making process to reach a consensus on whether an update will be implemented or not. Although both elements will be heavily studied, this thesis will not provide a full proof-of-concept implementation. Instead, this thesis provides code examples of the key elements of an implementation.

1.6 Structure of the Report

Figure 1.1 depicts the research deliverables for the intermediate steps of this study and outlines the structure of the report. The rest of this report is as follows, chapter 2 presents an overview of the current state of the art of blockchain technology. In chapter 3 the current state of the art of smart contracts is discussed, followed by its vulnerabilities in chapter 4. Chapter 5 gives an overview of existing solution designs and precedes chapter 6 on the requirements for a solution design. Chapter 7 presents the results of the illustrative case studies and is followed by the solution design in chapter 8. The last chapter, chapter 9, states the conclusion and discussion of this study.

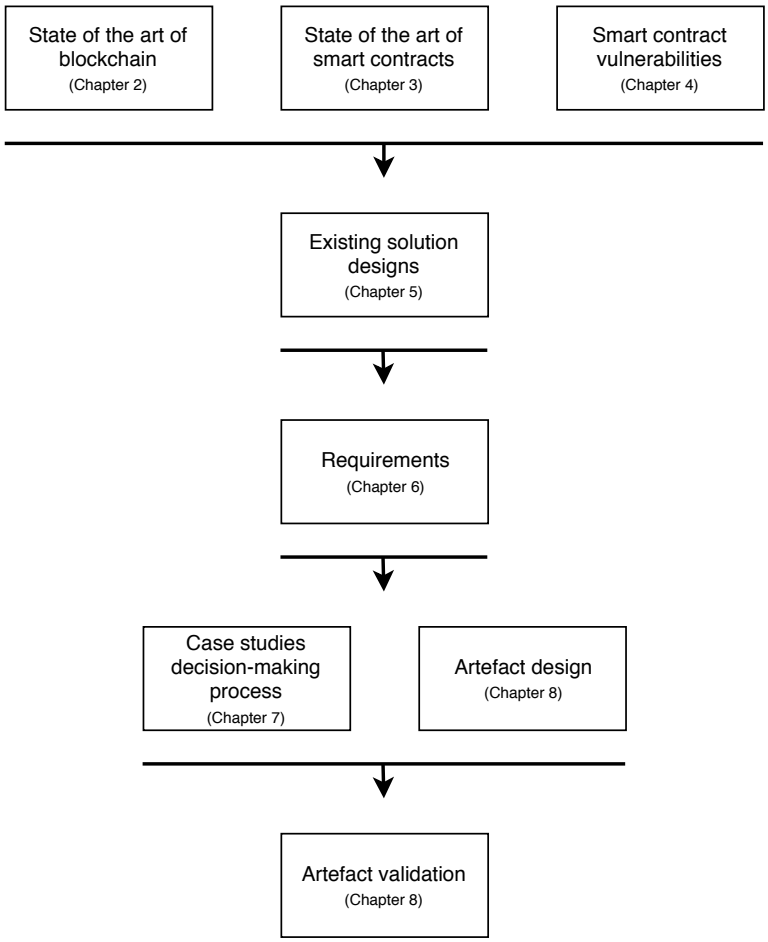

FIGURE 1.1: Research deliverables

Chapter 2

Blockchain Technology

Introduction

This chapter presents an overview of the current state of the art of blockchain technology and provides an answer to the first sub-question of this thesis:

1. *What is the current state of the art of blockchain technology?*

In order to find relevant papers for this study, the search query: "*Blockchain AND technology AND foundational*" was used to retrieve relevant articles. The additional condition "*AND foundational*" was added to narrow the scope of results, for the reason that the search query "*blockchain AND technology*" results in a list of articles which mention blockchain and technology only once, for example as a potential solution.

First, the concepts of blockchain technology and key characteristics are introduced along with the different types of blockchains. This is followed by a detailed description of the processing of transactions and consensus mechanisms. The last section of this chapter is focussed on the security of blockchain based systems.

2.1 Blockchain

2.1.1 Blockchain Concept

In 2008, Satoshi Nakamoto, whose real identity is still unknown, published a white paper about the core functionality and core principles for a peer-to-peer payment network that eliminates financial institutions (Nakamoto, 2008). The idea for a digital currency, as presented by Nakamoto, is not entirely new. Before Bitcoin marked the start of digital currencies, there were multiple projects that aimed at doing the same thing, hence not all technology behind Bitcoin not is completely new. However, Nakamoto was the first one able to solve one of the major issues in a peer-to-peer payment network: double-spending. As digital copies of a digital currency are easy to make, a peer-to-peer payment network needs effective methods to prevent double-spending. According to Tschorsch and Scheuermann (2016), Nakamoto cleverly combined decades of research in a creative and sophisticated manner in order to create a digital currency using a blockchain.

Simply put, a blockchain is a nearly immutable ledger distributed over a peer-to-peer network. In the Bitcoin project, it is used to save how much Bitcoins everyone has by saving all the executed transactions in a ledger. To save this data in a peer-to-peer network, the ledger is divided into blocks, each of them containing a list of executed transactions. Next to all transactions, each block additionally contains at least a hash of the block header, a hash of the block header of the previous block, and

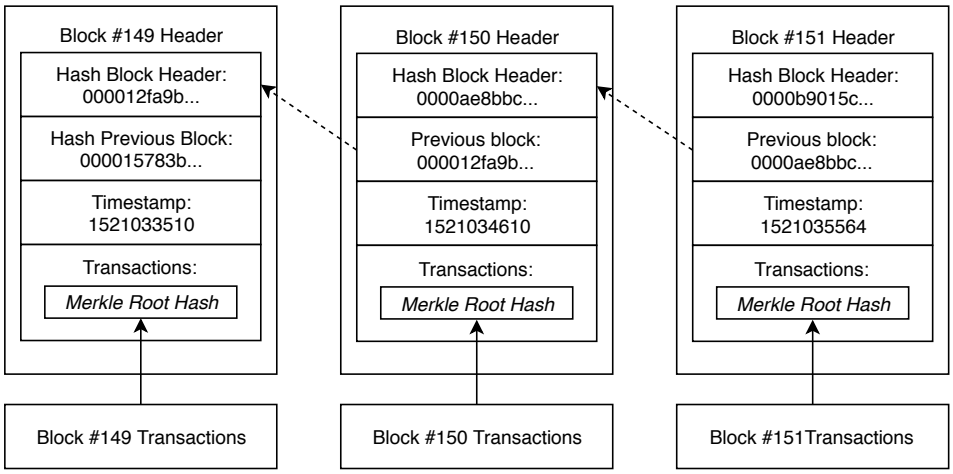

FIGURE 2.1: Simplified chain of blocks

a timestamp. By "linking" each new block to the previous block, a chain of blocks is created, hence this technology is called blockchain.

Figure 2.1 shows a simplified representation of the information in a block and the link to the previous block. As can be seen, each block contains the hash of the block header of the previous block and hash of its own header, which is necessary to ensure the integrity of the data in a block. If the content of a block in the blockchain is altered, the hash of the block header will become invalid, subsequently the hashes of all succeeding blocks also become invalid. For example, if one alters the data of *Block #149* in Figure 2.1, the hash (Proof of Work) becomes invalid. The hash of the block needs to be recalculated to become a valid hash again. However, this means that the hash of all succeeding blocks also needs to be recalculated as they become invalid due to the chaining mechanism.

Bitcoin, Ethereum, and the other blockchains can be viewed as a transaction-based state machine (Ethereum Foundation, 2018j). In the starting state, called the genesis state, transactions are executed to transit to a new state. The state can include different kinds of information such as account balances and reputation. Each state is a valid state, e.g. if a balance of an account is increased, that exact same amount should be deducted from another account. Figure 2.2 shows a simplified view of a state transition.

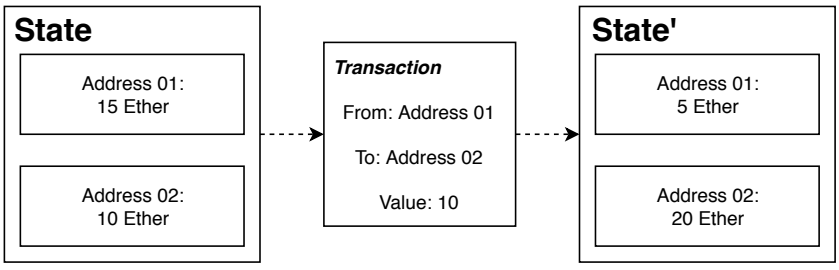

FIGURE 2.2: Simplified state transition Ethereum

2.1.2 Transactions and Accounts

The philosophy behind blockchain technology is that the ledger is distributed over thousands of computers in the network to create a trustful ledger. Even computers

without expensive hardware should be able to work with the ledger. However, saving all transaction data results in an extremely large blockchain in terms of file size. When the number of transactions stored on the blockchain increases, smaller computers will not have sufficient storage to save all data. Consequently, a client with a smaller computer will not be able to validate if his transactions were included in a block. Therefore, blockchains save Merkle trees of transactions (Bitcoin Project, 2018c).

A Merkle tree, sometimes referred to as Hash tree, is a concept in which every leaf at the lowest levels contains the hash of a piece of data, in blockchain often a transaction. Each leaf in a higher level of the tree consists of the hashes of its two children, eventually leading to one hash at the highest node in the tree. The Merkle tree allows for quick verifiability of data stored in the tree. In blockchain, it is used to determine whether a received block is undamaged and unaltered by a dishonest peer.

A client with a less powerful computer does not need to save the complete blockchain, instead, he only needs to save the block headers. In order to verify whether the transaction of the client is included in a block, the client downloads the list of transactions included in the block from a peer with the full blockchain, after which he computes the Merkle tree. This allows the client to validate whether the transaction was included in a block without the need to save the complete blockchain. Figure 2.3 shows a simplified version of how transactions are saved using a Merkle tree. The hash of each transaction is paired with the hash of another transaction in the same block.

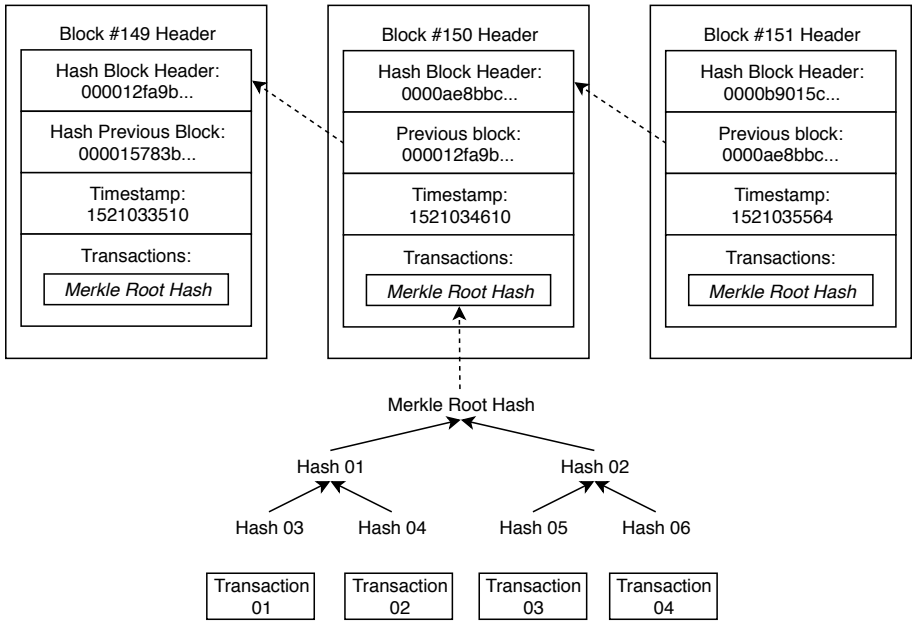

FIGURE 2.3: Simplified chain of blocks with Merkle tree

Blockchain platforms use asymmetric cryptography, or public key cryptography, techniques to ensure that a transaction is legitimate and authentic (Castaldo and Cinque, 2018). Public key cryptography relies on a set of two keys, a public key that is broadcast to the network, and a private key which is being kept secret by the user. A hash of the public key is used as the account address, i.e. the wallet address, of the user and can be used by others to transfer money to the user. To ensure the legitimacy and authenticity of a transaction, the sender of a transaction is required to provide a digital signature of the transaction using his private key. The digital

signature is used by other users and nodes in the network to verify that the transaction is sent by someone that is in possession of the private key that belongs to the public key. This verification uses an algorithmic function that takes the transaction, digital signature and public key of the sender as input and returns a boolean output whether the digital signature is authentic.

2.1.3 Blockchain Types

A number of publications describe different types of blockchains, however, the categorisation of the types is often conflicting. Peters and Panayi (2016) describe the differences between the types by looking at two aspects. The first aspect makes a distinction between whether authorisation is required to join the network as a node or not (permissioned versus permissionless). The second aspect makes a distinction between whether the data on the blockchain is publicly accessible or not (public versus private). In another study, only two different types of blockchain are described: permissionless and permissioned (Androulaki et al., 2018). In their publication Androulaki et al. also use the term public for permissionless blockchain, however, this is not the same type as described in Peters and Panayi (2016).

The differences between the types of blockchain are more nuanced, since there are more factors that differentiate blockchains from each other. On a high level, blockchains can be used in a public, private or consortium context (Lin and Liao, 2017):

Public: Everyone is able to view, verify and create transactions without revealing their true personal identity. Additionally, everyone is allowed to participate in the consensus making process. Examples of public blockchains are Bitcoin and Ethereum.

Private: In contrast to public blockchains, private blockchains do require authentication. There is an authority in place that controls who has the rights to view, verify and create transactions, and who can contribute to the consensus making process. As a consequence, private blockchains require participants to reveal their identity to a certain extent.

Consortium: A consortium blockchain combines features of public and private blockchains and can be used in business to business projects. Data on the consortium blockchain can either be public or private. The consortium blockchain also enables the entities that participate to decide whether everyone is allowed to run a node or that only selected group of entities is privileged.

Zheng et al. (2017) compared public, private and consortium blockchain by looking at five different properties. The results of this comparison are shown in Table 2.1. As can be seen, the authors included the distinctions that are described by Androulaki et al. and Peters and Panayi. Additionally, the authors considered the properties: immutability, efficiency, and centralisation, which will be discussed in section 2.1.5.

2.1.4 (De)centralisation

From a general perspective, blockchain implementations are focussed on decentralisation. The concepts of centralised, decentralised and distributed networks have

TABLE 2.1: Comparison of blockchain types (Zheng et al., 2017)

Property	**Public Blockchain**	**Consortium Blockchain**	**Private Blockchain**
Consensus determination	All miners	Selected set of nodes	One organisation
Read permission	Public	Could be public or restricted	Could be public or restricted
Immutability	Nearly impossible to tamper	Could be tampered	Could be tampered
Efficiency	Low	High	High
Centralisation	No	Partial	Yes
Consensus process	Permissionless	Permissioned	Permissioned

been a topic of academic research for a long time. Figure 2.4 shows an overview of these networks which was originally published by Tranter et al. in 1964. The definition of (de)centralisation can be explained across three axes when considering the term in blockchains (Buterin, 2017). The term (de)centralisation should not be considered as binary, it rather should be considered on a continuous axis.

Architectural (de)centralisation defines (de)centralisation as the number of physical computers of which the system is made up. A decentralised system tolerates a higher number of failing computers in comparison with a centralised system.

Political (de)centralisation defines (de)centralisation as the number of individuals or organisations that control the system. Decentralised systems are controlled by more individuals or organisations in comparison with centralised systems.

Logical (de)centralisation defines (de)centralisation as the level to which the interface and data structures are maintained as a monolithic object. Decentralised systems operate as independent systems when the system is cut in half.

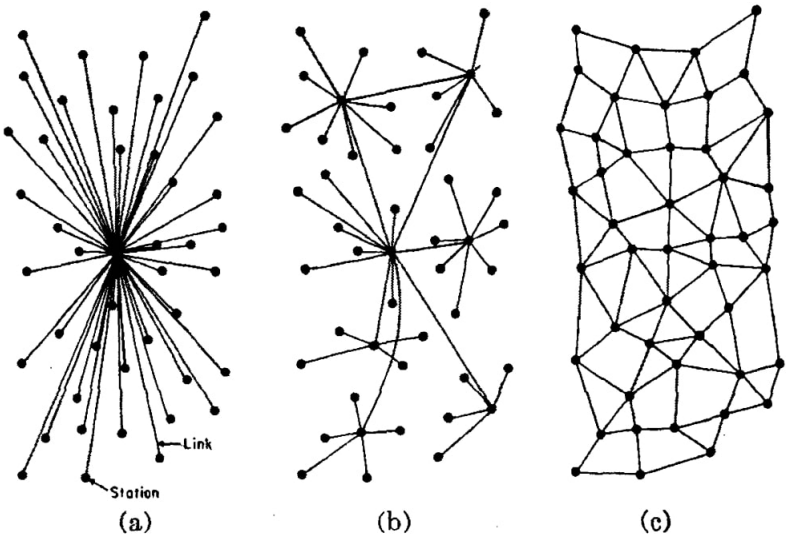

FIGURE 2.4: (a) Centralised (b) Decentralised (c) Distributed (Tranter et al., 2007)

2.1.5 Key Characteristics Blockchain

An extensive study on blockchain technology, architecture, consensus and trends was performed by Zheng et al. (2017). In their paper they describe the following key characteristics of blockchain:

Decentralisation: In traditional transaction-based systems, transactions are processed and validated by a central authority such as a bank. Such a third party is not needed in blockchain as all transactions are processed by all the nodes in the network. Together, these nodes maintain the integrity and availability of the data. As a blockchain network consists of many nodes, it can be assumed that it has a high degree of availability. Distributed Denial of Service and Denial of Service attacks will consequently need much more resources to get all the individual nodes in the network down.

Persistence: Once a transaction is stored on a blockchain, it becomes nearly impossible to alter or delete it. All nodes in the network observe each other to ensure the integrity of the stored records. As a result of the chain of hashes, as explained in section 2.1.1, an attacker would need to have a large amount of computational resources at his disposal to be able to alter records. He would not only need to calculate the valid hash of the block that he altered, but also of all the subsequent blocks.

Anonymity: The addresses that are used by users to send and receive transactions are generated by a cryptographic function. Consequently, the addresses do not contain any personal information about the user, allowing them to interact anonymously on the blockchain.

Auditability: The transactions in a blockchain are visible by anyone with access to the network. It allows the participants to retrieve a full audit trail of the transactions.

The confidentiality of a transaction is desirable in a financial system. It should be noted that although anonymity is a characteristic of blockchain technology, confidentiality is not ensured. The anonymity is based on the key pairs that are used authorise a transaction, however, the results of multiple studies indicate that in some blockchains the addresses can be traced back to an individual (Biryukov et al., 2014; Barcelo, 2014).

Considering the key characteristics of blockchain technology, the technology can be applied in numerous fields to make the exchange of value fairer and more transparent.

2.2 Distributed Consensus

Saving a transaction in a peer-to-peer network is challenging. As more peers join the network, the propagation time of a transaction in the network increases. With a digital currency, this allows a dishonest user to spend his money multiple times. Imagine that a dishonest user first spends money at *shop A*. Before he received a confirmation of the payment, the dishonest user quickly spends the same money at *shop B*. As there is a high chance that this transaction will be processed by another peer in the network, it might lead to a situation that this peer was not yet updated

about the transaction of the dishonest user at *shop A* due to the propagation time. The peers then need to reach consensus over which transaction was first.

There are multiple algorithms available to reach a distributed consensus in a network, each one of these has its own advantages and disadvantages. The consensus algorithms determine which node in the network may forge the next new block. Below four different algorithms will be discussed, these are Proof of Work (PoW), Proof of Stake (PoS), and Delegated Proof of Stake.

2.2.1 Proof of Work

The Bitcoin project makes use of a PoW algorithm to reach a consensus (Bitcoin Project, 2018d). The PoW algorithm lets nodes, which are called miners, calculate hashes of block headers. For every calculation of a hash, the miners use a different nonce in order to obtain a different hash. A nonce is an arbitrary number that is added to the input of the hash function. The node that is able to calculate a hash that starts with a predefined number of zeros, is selected by the PoW algorithm to mine the block. This predefined number of zeros is a method for the algorithm to adjust the difficulty. The time between finding the correct hash and corresponding nonce becomes lower as more miners with computational power join the network. To compensate for this, the PoW algorithm increases the difficulty by requiring a higher number of leading zeros for the correct hash. Once a miner is able to calculate a hash of the block that complies with the requirement, the miner broadcasts the solution and corresponding hash to all other miners in the network. The other miners validate whether the block is valid and the combination of the hash and the nonce indeed complies with the requirement. After that, the miners add the block to their own local blockchain, and the process starts all over again. As calculating hashes requires a significant amount of computational resources, this algorithm is not energy efficient.

2.2.2 Proof of Stake

The PoS algorithm is used by a number of different blockchains, and in the future also by Ethereum (Buterin, 2018). The PoS algorithm selects the node that is allowed to add the next block to the blockchain by a combination of randomisation and stake, or by coin age. The nodes participate in a lottery where the chance of winning is proportional with the stake of a node. This means that nodes are required to stake coins to get a chance to add a new block to the blockchain. As this does not require as much as computational resources as the PoW algorithm, it is more energy efficient. A downside of the PoS algorithm is that it is inexpensive for nodes to vote for different versions of the blockchain because the nodes have nothing to lose. Hence, this might result in situations in which no consensus can be achieved.

2.2.3 Delegated Proof of Stake

This type is another flavour of the PoS algorithm described above. While the PoS is completely democratic, Delegated Proof of Stake lets stakeholders select a representative to generate and validate blocks. The advantage of this method is that it reduces the number of required nodes to validate the blocks. A lower number of required nodes reduces the propagation time of the network, hence reducing the transaction times.

2.2.4 Other Consensus Algorithms

Apart from the aforementioned algorithms, there are many more different algorithms to be found, examples include:

- Tendermint
- Practical Byzantine Fault Tolerance
- Ripple
- Proof of Space
- Proof of Time
- Directed Acyclic Graph
- Proof of Authority

Each consensus algorithm has its own advantages and disadvantages. Per blockchain project, one should choose the most suitable and appropriate type based on the specific situation.

2.3 Security

As the price of a Bitcoin has increased significantly since its creation, it has become a potential target for attackers. Since Bitcoin makes use of an uncontrolled and decentralised environment, it is hard for attackers to steal or to commit fraud with the transactions. Any change or action of fraud can be traced and is visible for all other people in the network. In recent years, a number of studies have analysed the security of specific aspects of the Bitcoin system, however, more research is needed to make blockchain technology more mature (Matsuo, 2017). Kiayias and Panagiotakos provided a formal analysis of the Bitcoin backbone protocol by investigating the trade-off between the process speed of transactions and provable security (Kiayias and Panagiotakos, 2015). Conti et al. (2017) published a comprehensive survey on the security and privacy issues of Bitcoin. They discuss the current state of the art of attacks on transactions and user security in the blockchain, as well as the efficiency of solutions to mitigate these attacks. Although the survey is mainly focussed on the security and privacy aspects of the Bitcoin system, the results will for a large part also apply on the many alternative crypto currencies that use Bitcoin and its proof of work protocol as a basis. Below each vulnerability as found by Conti et al. on Bitcoin and the consensus protocol will be discussed shortly.

2.3.1 Double-Spending (Race Attack)

In the traditional banking system, double-spending by executing multiple concurrent transactions is prevented by the bank as a central entity. In a decentralised environment without a central bank, it is possible to do two concurrent transactions to different addresses. The Bitcoin protocol solves this issue partially by letting the entire network verify the legitimacy and existence of the transaction, hence double-spending of coins will be noticed by others in the network because a transaction will only be valid when the majority of miners agrees. Karame and Androulaki (2012) published a detailed study on double-spending attacks in the Bitcoin network. The

results indicate that the security of transactions can only be ensured when a verification time of tens of minutes can be tolerated. Karame and Androulaki (2012) suggest a modification for the Bitcoin implementation to be able to accurately detect double-spending attacks. Preventing the spreading of inaccurate information in the network is not a Bitcoin-specific problem, Lamport et al. (1982) described a similar problem called Byzantine Generals Problem back in 1982. Although Bitcoin is actually a clever solution to the Byzantine Generals problem, not everyone agrees that it is a complete solution (Garay et al., 2015). Without the Proof of Work concept introduced in Bitcoin, the network would be vulnerable to pseudospoofing (Sybil) attacks. A pseudospoofing attack is an attack in which multiple identities are forged in a system (Schreiber and Alexandre, 1974; Douceur, 2002).

2.3.2 Finney Attack

A Finney attack can be seen as a different flavour of a double-spending attack. Finney, the person who received the first Bitcoin transaction from Nakamoto, describes an attack in which the attacker mines a block with a transaction from address A to address B, both owned by the attacker (Finney, 2011). Before broadcasting the mined block to other nodes in the network, the attacker performs a transaction from address A with a merchant who accepts 0 confirmation transactions. The merchant will wait a few seconds before accepting the transaction to prevent double-spending. Once the merchant agrees with the transaction, the attacker broadcasts his mined block to other nodes. This makes the transaction with the merchant invalid, hence the merchant will not receive any coins at all. A widely used method to prevent this attack from the merchant's perspective is to always wait for multiple confirmations, i.e. multiple mined blocks after the block with the transaction.

2.3.3 Vector 76 (One-Confirmation Attack)

The Vector 76 attack, also referred to as the one-confirmation attack, is a hypothetical attack that combines the Finney attack with a double-spending attack (Bitcointalk Forum, 2011). In this attack, the attacker mines a block including his own transaction, which he did not send to the transaction pool. Once the attacker mined the block, he does not broadcast it to other nodes, instead, he waits till another miner mines the next block. When the next block is announced, the attacker quickly sends his own pre-mined block nodes close to an exchange. At the same time, the attacker requests a withdrawal from the output address that he included in his own pre-mined block. As there is a confirmation for the coins on that output address, the exchange will allow the withdrawal. At this moment in time, there will be two chains, one with a block of the attacker and one with the regular block. If the chain with the block of the attacker does not survive the validation of other nodes, the block will become valid. However, since the attacker already did a withdrawal from the exchange, the exchange will end up with a loss of coins. Again, a method to prevent these kinds of attacks is to wait for more confirmations.

2.3.4 $>$50% Hash Power

Mining is generally organised in so-called mining pools, these are groups of miners that solve the PoW puzzle together and share the rewards. When a mining pool reaches $>$50% of the total hash power, the pool will be able to control the whole network. If the pool decides to adopt new rules or a different strategy the other

miners are obliged to follow. The other miners are required to follow otherwise their blocks will be marked as invalid, withholding them from receiving any rewards at all (Kroll et al., 2013). Other consensus algorithms such as proof of stake and proof of activity try to overcome these hash power attacks for chains using the proof of work algorithm (Bentov et al., 2014; King and Nadal, 2012).

2.3.5 Selfish Mining (Block Discarding)

To perform a selfish mining attack, a dishonest miner keeps a mined block private for some time before broadcasting it to the public (Eyal and Sirer, 2014). This creates a temporarily private chain for the dishonest miner meanwhile the honest miners are still working on the main chain. After a few minutes, just before the next block is announced by honest miners, the dishonest miner broadcasts his found block. The honest miners will validate this block, add it to the main chain, and they will start to mine on the new main chain. In the meantime, the dishonest miner was already able to mine on the new main chain for a couple of minutes because he already had the last block. Vitalik Buterin, the person who proposed the idea of Ethereum late 2013, suggested in November 2013 that selfish mining is not worth to worry about as the higher-level economics makes this kind attacks unwanted for the attackers themselves (Buterin, 2013).

2.3.6 Block Withholding

Miners are often organised in pools to solve the proof of work puzzle together and to share the rewards. A block withholding attack takes place in these mining pools and takes advantage of the others in the pool (Bag et al., 2017). The attacker, in this case a participant in the mining pool, constantly sends the results from the proof of work puzzle to the pool administrator. However, the results of the proof of work puzzle are not complete, they are only partial. In this way, the attacker tries to let the administrator think that he is utilising his computing resources for the pool without actually doing that. Hence, the attacker receives an unfair reward for utilising his resources.

2.3.7 Fork After Withholding Attack

Similar to the block withholding attack, the attacker submits a full proof of work to the pool administrator only when another new block is found and broadcast to the network (Kwon et al., 2017). If the pool administrator accepts the proof of work, there will be two different chains (forks). The other participants in the Bitcoin network then have to choose one of the chains. If the chain of the attacker is selected, the pool, including the attacker, receives a reward. This type of block withholding attacks can also be executed on multiple pools, which can increase the rewards for the attacker with 56%. The participants could keep on mining on different chains, this is called hard forking.

2.3.8 Misbehaviour Attacks

Apart from the attacks on the Bitcoin network and protocol, Conti et al. also identified misbehaviour attacks (Conti et al., 2017).

These misbehaviour attacks are:

1. Bribery attacks
2. Refund attacks
3. Punitive and Feather forking
4. Transaction malleability
5. Wallet theft
6. Time jacking
7. DDoS
8. Eclipse (netsplit)
9. Tampering
10. Routing attacks
11. Deanonymisation

A description, the primary targets, adverse effects and possible countermeasures for these kinds of attacks are provided by Conti et al. (2017).

2.3.9 Usage Risks

In order to create a transaction in a blockchain network, the sender needs to sign the transaction with his private key. By doing this, all the miners in the network are able to verify that it is a legitimate transaction, i.e. the creator of the transaction has access to the account. This also means that the private key should be kept private, otherwise other people will be able to sign a transaction. If the owner of an address lost the private key, he cannot access his account any more.

Recent reports suggest that there is dedicated malware that replaces Bitcoin addresses that are copied to Windows clipboard (Abrams, 2018). This malware automatically changes a Bitcoin address that is copied to the clipboard of Windows to the address of the attackers.

Chapter 3

Smart Contracts

Introduction

This chapter presents the results of the study on the current state of the art of smart contracts, and provides an answer to the second sub-question of this study:

2. *What is the current state of the art of smart contracts?*

In order to provide an answer to the sub-question, this thesis adopts a desk research methodology. Preliminary research towards smart contracts indicated that smart contracts are not yet the topic of research in academic literature, although, vulnerabilities and vulnerabilities scanners for smart contracts are studied to a certain extent. The desk research relies on the yellow paper of Ethereum to outline the structure of the current state of smart contracts. Additional sources such as the Ethereum documentation, Solidity documentation and other websites are consulted to retrieve detailed information and examples.

This chapter starts with a general description of the concept of smart contracts, after which it describes the Ethereum smart contracts in detail. To provide more context on the added value of smart contracts, the chapter finishes with a non-exhaustive description of use cases.

3.1 Smart Contract Concept

Smart contracts are software programs that are executed on a blockchain and facilitate agreements between mutually distrusting actors. The term smart contract was coined by Nick Szabo in 1996 (Szabo, 1996). As Szabo describes, a contract is a conventional method to formalise promises and is one of the building blocks of the free market economy. Companies rely heavily on their contracts with other businesses to provide their services and to keep up with their promises. The principles of contracts can also be applied to the digital world. This led Szabo to present the idea of smart contracts:

"The basic idea of smart contracts is that many kinds of contractual clauses (such as liens, bonding, delineation of property rights, etc.) can be embedded in the hardware and software we deal with, in such a way as to make breach of contract expensive (if desired, sometimes prohibitively so) for the breacher." (Szabo, 1996)

When the attention towards blockchain technology increased, smart contracts resurfaced in consideration that blockchain would enable smart contracts to be used in everyday life. Blockchain is considered as a suitable technology to store and execute the smart contracts as it safeguards the availability and integrity, as described in section 2.1.5. By deploying smart contracts on a blockchain, the contracts are automatically enforced without the need for a trusted third party. Similar to conventional

contracts, the program code of a smart contract becomes immutable once it is stored on the blockchain. The correctness of the execution is ensured by the blockchain; tampering with the execution of the program and its results is nearly impossible.

The basic concept of smart contracts is supported by multiple blockchains, for example, by the Bitcoin blockchain (Bitcoin Project, 2018a). In the white paper of Ethereum, it is argued that Bitcoin has several limitations for contracts, such as the lack of Turing-completeness (Ethereum Foundation, 2018j). Turing-completeness refers to the set of computational operations that is supported by a programming language and, in this context, a blockchain. For instance, Bitcoin does not provide support to run a loop in a smart contract, in order to avoid infinite loops during the execution of a smart contract. After the limitations of scripts for Bitcoin were identified, the founders of Ethereum implemented a programming language that avoids the limitations of Bitcoin. To date, Ethereum is the most prominent and well-known blockchain that supports smart contracts.

Gavin Wood, the Chief Technology Officer of the Ethereum Project, presented the initial technical description of Ethereum blockchain (Wood, 2014). In his paper, he describes the three key elements of Ethereum: blocks, states and transactions. Moreover, he describes transaction execution, transaction payment, message call and the execution model. The Ethereum project is constantly evolving, the improvements are reflected in a yellow paper that is maintained on a Git repository (Ethereum Foundation, 2018c).

3.2 The Ethereum Smart Contract

3.2.1 Solidity

Smart contracts for the Ethereum blockchain can be written in different programming languages. For the sake of this study, only the programming language Solidity is considered, as previously stated in the scope of this thesis (section 1.5). Solidity is a high-level programming language specifically targeted at writing code that compiles Ethereum Virtual Machine (EVM) code (Ethereum Foundation, 2018a). EVM code is the code that is stored and executed by nodes in the Ethereum network (Ethereum Foundation, 2018j). The syntax of Solidity overlaps with the JavaScript syntax, which makes it for many developers trivial to understand. The documentation of Solidity describes the language as:

"Solidity is a contract-oriented, high-level language for implementing smart contracts. It was influenced by C++, Python and JavaScript and is designed to target the Ethereum Virtual Machine (EVM). Solidity is statically typed, supports inheritance, libraries and complex user-defined types among other features." (Ethereum Foundation, 2018a)

```
1  pragma solidity ^0.4.23;
2
3  contract SimpleStorage {
4      uint storedData;
5
6      function set(uint x) public {
7          storedData = x;
8      }
9
10     function get() public constant returns (uint) {
11         return storeData;
12     }
13 }
```

LISTING 3.1: Contract written in Solidity

An example of a smart contract written in Solidity is given in Listing 3.1. The example shows a single contract class called `SimpleStorage` that introduces two methods: `set()` and `get()`. This contract, as provided by the Solidity documentation, allows to store and retrieve an integer on the blockchain. Although this example is an actual smart contract, it actually functions as a simple and straightforward program and is not really useful in practice. That being said, Solidity is capable of more advanced computations and structures, for instance, inheritance of contract classes and payable methods. A payable method is a function that contains the modifier `payable`, it allows the function to receive Ether, the digital currency that is used on the Ethereum blockchain. The contract class above could be inherited by another contract class, which would give the contract class access to the `set()` and `get()` methods. Detailed examples of a decentralised auction and safe purchasing are presented in the Solidity documentation (Ethereum Foundation, 2018k).

3.2.2 Ethereum Virtual Machine

To deploy a smart contract written Solidity, first, the source code is compiled to EVM code (Ethereum Foundation, 2018k). The EVM code contains all the necessary computations to execute the contracts; inherited contract classes are thus included in the EVM code. The source code of the `SimpleStorage` smart contract from Listing 3.1 is compiled to:

```
608060405234801561001057600080fd5b5060df8061001f6000396000f300608060405
260043610604957600357c010000000000000000000000000000000000000000000000000
0000000000900463ffffffff16806360fe47b114604e5780636d4ce63c146078575b600
080fd5b348015605957600080fd5b506076600480360381019080803590602001909291
9050505060a0565b005b348015608357600080fd5b50608a60aa565b604051808281526
020019150506040518091039f35b80600081905550565b60008054905090560a165
627a7a72305820e484bdb4cb178a6c0c5b95a0c2eee2775f79fb007d438689aa13cb73d
99fbef50029
```

The string is a series of bytes in which each byte represents a computational operation. The EVM code of a smart contract is executed within an EVM, which is a lightweight operating system that each node runs. After the Solidity smart contract is compiled to EVM code and deployed on the blockchain by means of a transaction, the EVM code becomes immutable.

Smart contracts can be used to transfer money, consequently, participants need to be sure that the smart contracts do what they supposedly should do. An issue here is that the EVM code is not human readable, making it impossible for participants to validate the contents of a smart contract. Etherscan (2018a) launched a platform that allows developers of smart contracts to upload the source code. The platform compiles the source code to EVM code and verifies whether the source code matches with the smart contract on the blockchain. On one hand, publishing source code can be useful, for example, to gain more trust of participants, on the other hand, it is also a risk due to the fact that it is showing the functions of a smart contract. It provides malicious users with the necessary information, such as function names, to exploit a smart contract.

3.2.3 Ethereum Transaction

Ethereum can be viewed as a transaction-based state machine in which the world state is a mapping between addresses and account states; transactions are executed to alter the current state (Wood, 2014). These transactions are used to transfer money, to deploy a smart contract, and to invoke methods of a smart contract.

Table 3.1 presents an overview of the fields that an Ethereum transaction contains. A transaction can be used to deploy a new smart contract by providing EVM code to the *data* and/or *init* field of a transaction. EVM code provided via the *init* field is only executed during contract deployment and is immediately discarded afterwards. The code provided via the *data* field is deployed on the blockchain and gets its own 256-bit address that can be used to interact with the smart contract. The address can either be used to send money to the smart contract or to invoke a method of the smart contract.

As Table 3.1 shows, the person or smart contract that initiates a transaction needs to provide a *gasLimit*, this number can be seen as fuel for the transaction. By charging a fee for the execution of a transaction, abuse of the network by flooding it with transactions becomes expensive (Atzei et al., 2017). The miner of a block receives all the gas that is paid for the individual transactions in that specific block as a reward. All unconsumed gas after a transaction is processed is sent back to the transaction initiator, hence it is called *gasLimit*. In case that processing the transaction exceeds the *gasLimit*, the transaction will be halted, and the state will be reverted to its initial state. The total costs of a transaction are composed of the following four components:

1. Base transaction fee (21,000 gas)
2. Cost for every zero byte of data or code for a transaction
3. Cost for every non-zero byte of data or code for a transaction
4. Execution costs

Thus, the total amount of gas that is required for a transaction is based on the data and computations that are needed to process the transaction: a simple transfer of balances is consumes less gas than executing a complex smart contract. The price for which gas will be bought is provided by the sender via the *gasPrice* field in a transaction. The value in this field states the price of one gas in Wei: 10^{18} Wei is equal to 1 Ether. While considering that the miner of a block receives the gas of all the transaction in the block mined by him, it is expected that transactions with a high *gasPrice* have lower processing times. The price of gas is determined by the

market; it increases when the demand to process transactions grows. Figure A.1 in Appendix A shows the *gasPrice* of mined transactions between August 16^{th} 2017 and August 15^{th} 2018.

TABLE 3.1: Fields in an Ethereum transaction (Wood and Ethereum Foundation, 2018)

Field	**Description**
nonce	A scalar value equal to the number of transactions sent by the sender; formally T_{n}.
gasPrice	A scalar value equal to the number of Wei to be paid per unit of *gas* for all computation costs incurred as a result of the execution of this transaction; formally T_{p}.
gasLimit	A scalar value equal to the maximum amount of gas that should be used in executing this transaction. This is paid up-front before any computation is done and may not be increased later; formally T_{g}.
to	The 160-bit address of the message call's recipient or, for a contract creation transaction, $\varnothing$, used here to denote the only member of $\mathbb{B}_0$; formally T_{t}.
value	Values corresponding to the signature of the transaction and used to determine the sender of the transaction; formally T_{w}, T_{r} and T_{s}.
v, r, s	Three scalar values corresponding to the signature of the transaction and used to determine the sender of the transaction.
data *(optional)*	An unlimited size byte array specifying the input data of the message call, formally $T_{\mathbf{d}}$.
init *(optional)*	An unlimited size byte array specifying the EVM-code for the account initialisation procedure, formally $T_{\mathbf{i}}$.

3.2.4 Interacting with a Smart Contract

To invoke a method from a smart contract, one includes the method identifier and arguments in the *data* field of a transaction. The method identifier can be computed by taking the first four bytes of the SHA3 hash of the signature of the function (Ethereum Foundation, 2018b). Smart contracts can contain a fallback function, which is implemented as an unnamed function, that will be triggered when a non-existing function is invoked. The fallback function does not take any arguments and has by default no return value.

The EVM code that is visible on the blockchain does not provide the necessary information to call a function within a smart contract because it does not reveal the method identifier and parameters, it only reveals the EVM code. Consequently, to interact with a smart contract, one should have access to the source code or to the application binary interface (ABI) (Ethereum Foundation, 2018e). The ABI describes the method identifiers, arguments, and return values. Furthermore, it provides a developer of a smart contract with the required information to interact with a smart contract without the need to have access to the Solidity source code. Listing 3.2 provides the ABI of the smart contract in Listing 3.1.

```
[
    {
        "constant": false,
        "inputs": [
            {
                "name": "x",
                "type": "uint256"
            }
        ],
        "name": "set",
        "outputs": [],
        "payable": false,
        "stateMutability": "nonpayable",
        "type": "function"
    },
    {
        "constant": true,
        "inputs": [],
        "name": "get",
        "outputs": [
            {
                "name": "",
                "type": "uint256"
            }
        ],
        "payable": false,
        "stateMutability": "view",
        "type": "function"
    }
]
```

LISTING 3.2: The ABI of the Smart Contract from Listing 3.1

3.2.5 Operational Code

Only the EVM code of a smart contract is stored on the Ethereum blockchain. EVM code is hard to understand for human beings, making it nearly impossible to determine the execution path that is followed by a smart contract. An alternative for this is to translate the EVM code to operational code, which describes all low-level computations that are executed in a smart contract in a more human-readable representation. Ethereum Foundation (2017); Wood (2014) provide references for the hexadecimal codes for these low-level operations, and the amount of *gas* used for each operation. A snippet of the operational code of the EVM code in section 3.2.2 is:

```
PUSH1 0x80 PUSH1 0x40 MSTORE CALLVALUE DUP1 ISZERO PUSH2 0x10 JUMPI
PUSH1 0x0 DUP1 REVERT JUMPDEST POP PUSH1 0xDF DUP1 PUSH2 0x1F PUSH1 0x0
CODECOPY PUSH1 0x0 RETURN [...]
```

The operational code allows to compute all possible states in which a smart contract can be. With the use of dedicated tools e.g. Mythril, a visualisation of all the possible states can be generated (ConsenSys Diligence, 2018a). An in-depth description of Mythril and an alternative is provided in section 4.2.

3.2.6 Oracles

Smart contracts are designed to interact with the outside world on a limited basis, only in case a smart contract is deployed, or a transaction is sent to a smart contract. An oracle is the bridge between blockchain and the outside world and allows to get external data onto the blockchain (Clack et al., 2016). Although it can be argued that a direct call from a smart contract to an outside located interface is supported by default, the traffic that will be generated by all the nodes to the interface might result in high response times or non-responding interfaces. Moreover, each node in the network might receive different results. For instance, an external service that is used to retrieve a random number between 1 and 100 will send different outcomes to each node. An oracle relies on a centralised service to perform the call to the external interface and to store the results on the blockchain (Oraclize Limited, 2018a). While the centralised service only performs a single request, each node will get the same return value from the oracle and the external interface will not be flooded with requests. A disadvantage of an oracle service is its centralisation, the oracle service provider could interfere with the execution. Oraclize Limited solved this challenge by implementing a proof of authenticity, using TLSNotary, to provide evidence that they did not interfere with the request or its result (Oraclize Limited, 2018b).

3.3 Use Cases

A growing body of literature investigates use cases of smart contracts. The results indicate that a smart contract provides value for a wide range of sectors. Idelberger et al. (2016) state that use cases of smart contracts include escrow services, savings and digital tokens in the banking sector, decentralised markets in the trading sector, and administration of royalties in the music industry. A study by Frantz and Nowostawski (2016) suggests that smart contracts can be used for voting, crowdfunding, asset management and workflow management. Two domains that received significant interest from academics are the energy sector and Internet of Things (IoT). The energy sector can improve its efficiency and transparency by adopting smart contracts (Knirsch et al., 2018; Kounelis et al., 2017; Hahn et al., 2017; Lamers, 2018). In IoT, smart contracts can be implemented to enhance the level of security of the devices. (Banafa, 2017; Christidis and Devetsikiotis, 2016; Kshetri, 2017b; Bahga and Madisetti, 2016; Kshetri, 2017a).

This section touches upon use cases for smart contracts in digital tokens, escrow services, energy sector and IoT. It should be noted that the presented list of use cases in this section is non-exhaustive.

3.3.1 Escrow Services

Smart contracts can facilitate the exchange of digital assets by fulfilling the role as escrow agent (Wood and Buchanan, 2015). In traditional systems, escrow mechanisms allow two distrusting parties to exchange value by selecting a trusted third party, such as a bank or notary, that will ensure that both parties will fulfil their obligations. The trusted third party could be replaced by a smart contract, which could

be designed such that the assets are released after certain conditions are met by both parties. Frantz and Nowostawski (2016) and Ethereum Foundation (2018h) provide examples of smart contracts specifically designed to replace an escrow agent.

Current implementations of escrow smart contracts do not support the exchange of physical assets. A major challenge to digitalise the exchange of physical assets is to link the physical assets to a digital object. Although this is challenging, there are platforms that started supporting the exchange of physical assets, such as Propy (Prnewswire, 2018). Propy is a blockchain-based platform that enables users to buy and sell real estate in a digital manner. Start-ups, such as VeChain, aim to solve these challenges by implementing NFC-tags in physical assets (VeChain, 2018).

3.3.2 Digital Tokens

In recent years, it has become a prevailing method for start-ups in the blockchain sector to raise capital by launching an initial coin offering (ICO). A study by Fenu et al. (2018) showed that 1082 start-ups raised a total capital between 30 and 40 billion US by means of an ICO in 2017. During an ICO, a start-up raises money by selling digital tokens using smart contracts. Catalini and Gans (2018) suggest that an ICO provides a better price discovery and that an ICO enables a start-up to attract more investors without giving each of them the rights that are typically associated with equity. Digital tokens can be categorised as either fungible or non-fungible tokens.

Fungible Tokens

Fungible tokens are digital tokens that are interchangeable and thus have the same value. An analogy of this concept is can be made with a one Euro coin: a one Euro coin can be exchanged with any other one Euro coin, it does not change anything for the holder. The ERC-20 standard was introduced to enable the re-use of this concept across different kinds of dApps (Vogelsteller et al., 2018).

Non-fungible Tokens

Non-fungible tokens are digital tokens that represent unique objects and are not interchangeable. An analogy of this concept are educational certificates: although each educational certificate is a certificate, the certificates do not have the same value. To enable the re-use of non-fungible tokens, the ERC-721 standard was introduced (Entriken et al., 2018).

3.3.3 Energy Sector

In the energy sector, smart contracts and blockchain technology can provide financial benefits (Lamers, 2018). One third of the price that a consumer spends on energy can be accounted to the actual cost of the energy, the other part of the costs consists of fees and taxes. From this perspective, it makes sense that academics are interested in this research area. Knirsch et al. (2018) present a privacy-preserving system for determining tariffs on a wide scale in a smart grid, by using fungible tokens. These smart grids deploy incentive-based solutions to improve the efficiency of the energy market, for example by making energy less expensive at night. The authors showed that tariff decisions for the energy market can benefit from smart contracts by improving the transparency, verifiability, reliability and privacy of the system. Another study that takes advantage of smart contracts and blockchain technology is aimed at automating energy exchange and auctions for micro-generation (Kounelis et al.,

2017). The researchers deployed a smart contract that automatically issued new digital tokens for energy producers every time that the energy producer releases energy to the grid. The energy producers, such as households with solar panels, receive digital tokens linearly with the amount of energy that they produce. A similar research was done by Hahn et al. (2017), however, they put more emphasis on the auction side of such smart grid systems. The researchers demonstrated the implementation of an auction that contains the key features and data to manage the energy auction process. This concept is also described in the master thesis by Lamers (2018).

3.3.4 Internet of Things

There is a considerable volume of published studies describing the role of smart contracts in IoT. Christidis and Devetsikiotis (2016) present a use case for smart contracts in asset tracking with IoT. As the authors describe, a supply chain process involves a number of different stakeholders. Each stakeholder usually maintains their own database to keep track of the assets. Whenever a stakeholder sends an update about the asset, the other stakeholders update their own databases. The researchers describe how IoT-enabled devices and smart contracts can improve the efficiency of the asset tracking process. By placing an IoT-enabled device on the asset, the asset itself can start broadcasting updates: "*when the shipping carrier reaches the destination port, they send a signed message to a predetermined and agreed-upon smart contract to allow everyone on the chain to know that the container is now at point C. Since the transaction is signed, it acts as a cryptographically verifiable receipt of the shipping company's claim that the container has reached the destination port. The receiver at the port posts to same smart contract to confirm that*" (Christidis and Devetsikiotis, 2016).

Bahga and Madisetti (2016) elaborate on a blockchain platform for industrial manufacturing IoT device that deploys smart contracts. The Blockchain Platform for Industrial Internet of Things lets devices connect with the blockchain that uses smart contracts to facilitate the agreement between service consumers and manufacturing resources. According to the authors, this platform can be used for:

(i) On-demand manufacturing

(ii) Smart diagnostics & machine maintenance

(iii) Traceability

(iv) Supply chain tracking

(v) Product certification

(vi) Consumer-to-machine & machine-to-machine transactions

(vii) Tracking supplier identity & reputation

(viii) Registry of assets & inventory

Kshetri (2017b) states that automated execution of actions using a smart contract can reduce the costs and capacity constraints in IoT, however, the author fails to provide details regarding the use of smart contracts in this context.

Chapter 4

Smart Contract Vulnerabilities

Introduction

This chapter presents the results of the literature study to smart contract vulnerabilities and provides an answer to the third research question:

3. *What are existing security vulnerabilities of smart contracts?*

Although smart contracts are not described extensively in academic literature, their vulnerabilities are to some extent discussed and researched. To retrieve relevant publications from ACM, Scopus and IEEE, the query *"(smart AND contract AND vulnerabilities) OR (smart AND contract AND vulnerability)"* was used. Furthermore, the database of Google Scholar was queried to include grey literature. As the number of results was too high on Google Scholar, the query was only performed on the title of documents. Additionally, non-academic online sources were consulted to gather more detailed information on vulnerabilities. These sources include the official documentation of Ethereum and the documentation provided by ConsenSys Diligence. The documentation of ConsenSys Diligence was consulted given that the safety section of the Ethereum documentation states that the section itself is outdated and it refers to the documentation of ConsenSys Diligence for a recent version (Ethereum Foundation, 2018g; ConsenSys Diligence, 2018b).

Although not explicitly stated in the sub-question, this part of the study also includes smart contract vulnerabilities scanners. A number of articles found during the literature study describe tools that are able to detect specific vulnerabilities in smart contracts. The results of these researches provide insight into the current state of security of existing smart contracts and are therefore considered as valuable information for this research.

The chapter starts with taxonomies of smart contract vulnerabilities that are found in literature and it provides context by describing a number of vulnerabilities in detail. After that, the studies towards smart contract vulnerability scanners are described. The last section of this chapter presents recommendations for smart contract development.

4.1 Smart Contract Vulnerabilities

Albeit being a relatively new concept, smart contract vulnerabilities are already described and analysed in both academic literature and online sources. Some articles are focussed on describing multiple existing vulnerabilities, while others are specifically focussed on a single vulnerability. One would expect that academic literature and online sources would describe similar vulnerabilities. After a thorough analysis,

TABLE 4.1: Smart contract attacks (ConsenSys Diligence, 2018b)

Category	**Attack**
Race conditions	Re-entrancy
	Cross-function race condition
	Pitfalls in race condition solutions
	Transaction-ordering dependence / Front running
None	Timestamp dependence
	Integer overflow and underflow
	Denial-of-service with (unexpected) revert
	Denial-of-service with block gas limit
	Forcibly sending ether to a contract
Deprecated	Call depth attack

it can be concluded that this is not the case at all, there is actually a significant difference. Not only do academic articles describe a higher number of vulnerabilities, they also tend to classify vulnerabilities.

Table 4.1 lists ten vulnerabilities described by ConsenSys Diligence (2018b). In contrast, a recent literature study of Dika towards smart contract security resulted in a taxonomy with 22 vulnerabilities, as presented in Table 4.2 (Dika, 2017). Besides a classification of the vulnerabilities into the types EVM, blockchain or solidity patterns, Dika also studied the level severity for each vulnerability. The classification used by Dika is the same classification as used in a study by Atzei et al. (2017). The significant difference between the taxonomy by Dika and the vulnerabilities described by ConsenSys Diligence is partially caused by a different definition of a vulnerability. As an example, Dika describes "Lack of transactional privacy" as a vulnerability while ConsenSys Diligence describes this as a recommendation under "Remember that on-chain data is public" (Dika, 2017; ConsenSys Diligence, 2018b). Another challenge between comparing vulnerabilities is the sources use a different naming for the same vulnerability.

Smart contracts require a different mindset when it comes to discovering and exploiting vulnerabilities in comparison with traditional applications (Dika, 2017; Destefanis et al., 2018). As smart contracts run on a blockchain, vulnerabilities can be the result of the state machine principle of a blockchain. As example, below three vulnerabilities from the taxonomy in Table 4.2 are described. A description of all vulnerabilities can be found in the research from Dika (Dika, 2017).

4.1.1 Transaction-ordering Dependence

A vulnerability as a result of the state machine principle is transaction-ordering dependence (Luu et al., 2016). When interacting with a smart contract, i.e. sending a transaction, one can assume that the smart contract is in a specific state. However, the transaction is not immediately executed on the blockchain, it is first sent to a "transaction pool": a list of transactions that need to be processed on the blockchain. Before the transaction is processed in a block, another transaction that interacts with the same smart contract might be processed. As a consequence, the smart contract might be in a different state, for example, a state that pauses the complete smart contract. The transaction-ordering dependence thus means that one can never know in what state a smart contract is when a transaction is processed in a block.

TABLE 4.2: Taxonomy of smart contract vulnerabilities and severity (Dika, 2017)

Type	Vulnerability	Severity Level
Blockchain	Unpredictable state (dynamic libraries)	2
	Generating randomness	2-3
	Time constrains / Timestamp dependence	1-3
	Lack of transactional privacy	1-3
	Transaction-ordering dependence	2-3
	Untrustworthy data feeds (oracles)	3
EVM	Immutable bugs/mistakes	3
	Ether lost in transfer	3
Solidity	Gas costly patterns	1-2
	Call to the unknown	3
	Gasless send	3
	Exception disorders / Mishandled exceptions / Unchecked-send bug	3
	Type casts	2
	Re-entrancy	3
	Unchecked math (Integer over- and underflow)	1-2
	Visibility / Exposed functions or secrets/ Failure to use cryptography	2-3
	'tx.origin' usage	3
	'blockhash' usage	2-3
	Denial of Service	3
	'send' instead of 'transfer'	1-2
	Style violation	1
	Redundant fallback function	1

4.1.2 Re-entrancy

The re-entrancy vulnerability is one of the vulnerabilities that has been exploited in the DAO attack (Popper, 2016). In comparison with the previous described vulnerability, this vulnerability is more technical. Consider the code example in Listing 4.1: a simple contract the sends 2 *wei* to address c. Now suppose that an attacker uploads a contract to the blockchain with special a fallback function that invokes `ping(address c)`. As `c.call.value(2)()` invokes the fallback of the contract of the attacker, the attacker can re-enter `ping(address c)` because `sent` is still *false*. The smart contract will once again execute `c.call.value(2)()`. This loop halts when a transaction is out of gas, the funds of the originating contract are 0, or when the call stack limit has been reached.

```
1 contract Bob {
2     bool sent = false;
3
4     function ping(address c) {
5         if (!sent) {
6             c.call.value(2)();
7             sent = true;
8 }}}
```

LISTING 4.1: Example code for re-entrancy (Atzei et al., 2017)

4.1.3 Tx.origin Versus Msg.sender

A transaction can consist of a chain of multiple transactions. Solidity provides two methods to determine from what address a transaction is sent: (i) `tx.origin` (ii) `msg.sender` (Ethereum Foundation, 2018a). The difference between these is that `tx.origin` returns the address of the full origin of the transaction whereas `msg.sender` only returns the address of the sender of the last transaction. As described in the Security Considerations chapter of the Solidity documentation, the usage of `tx.origin` as authorisation method in a smart contract can be used in an attack (Ethereum Foundation, 2018f). Consider the following example: assume that person *A* holds his funds in the smart contract with the code from Listing 4.2. Now person *A* becomes the victim of phishing type of attack and he is tricked into sending some funds to the smart contract with the code from Listing 4.2. What happens is that the fallback function of the attacker smart contract invokes `transferTo(address dest, uint amount)` function of the smart contract that person *A* owns. While person *A* is the full origin of this transaction, the smart contract will pass the check on `require(tx.origin == owner)` and all the funds on the smart contract are transferred to the wallet of the attacker.

```
1  pragma solidity ^0.4.11;
2
3  // THIS CONTRACT CONTAINS A BUG - DO NOT USE
4  contract TxUserWallet {
5      address owner;
6
7      function TxUserWallet() public {
8          owner = msg.sender;
9      }
10
11     function transferTo(address dest, uint amount) public {
12         require(tx.origin == owner);
13         dest.transfer(amount);
14     }
15 }
```

LISTING 4.2: Vulnerable Smart Contract using Tx.origin (Ethereum Foundation, 2018f)

4.2 Smart Contract Vulnerability Scanners

There exist a number of scanners that analyse smart contracts for vulnerabilities. These tools are developed by academic researchers as well as commercial organisations. Dika presents a list of nine different scanners in his thesis, none of which is capable of detecting all vulnerabilities from Table 4.2 (Dika, 2017). Smart contract analysers have different scopes, some are focussed on Solidity source code while others are only focussed on EVM code. Scanners such as Oyente and Securify are capable of analysing both Solidity source code and EVM code. The effectiveness, accuracy, consistency of each scanner is different, each scanner has its own strong and weak points.

Two studies that describe a security analysis of smart contracts on the Ethereum blockchain are described in more detail below.

TABLE 4.3: Results analysis Oyente (Luu et al., 2016)

	Number of contracts containing issues	**Number of contracts with distinct EVM code**
Mishandled exceptions	5.411	1.385
Transaction-ordering dependence	3.056	135
Timestamp dependence	83	-
Re-entrancy Handling	2	1

TABLE 4.4: Correctness analysis Oyente (Luu et al., 2016)

	Source code available	**False positives**
Mishandled exceptions	116	0
Transaction-ordering dependence	32	9
Timestamp dependence	7	-
Re-entrancy Handling	2	1

4.2.1 Oyente

An extensive analysis of vulnerabilities in existing smart contracts was performed by Luu et al. (2016). A dedicated tool called Oyente was developed to be able to analyse existing smart contracts on the Ethereum blockchain. Oyente uses symbolic execution to enable statistical reasoning about paths followed by the program. At the time of publishing, the implementation of Oyente supported the detection of the following vulnerabilities:

1. Transaction-ordering dependence
2. Timestamp dependence
3. Mishandled exceptions
4. Re-entrancy vulnerability (Race conditions)

During their analysis, Luu et al. collected all deployed smart contracts on the Ethereum blockchain till the 5^{th} of May 2016. This resulted in a total collection of 19,366 smart contracts with a total value of 3,068,654 Ether, at that time worth around 30 million US dollars. After 3,000 hours on four Amazon m4.10xlarge instances on Amazon, the tool found in total 8,833 contracts with at least one security issue or vulnerability. An analysis of the EVM code showed that 1,682 of these smart contracts were distinct. Note that these are distinct in EVM code and not necessarily in source code: this means that two smart contracts with the same logic but with a different variable or function naming will also be distinct. Of the 1,682 distinct smart contracts, the source code of only 175 smart contracts was published on EtherScan (Etherscan, 2018a). With the source code of these smart contracts, Luu et al. were able to verify the correctness of the tool. In 10 out of the 175 cases, the security issue found by Oyente was a false positive. Table 4.3 shows how often each vulnerability is found in a smart contract. The correctness, i.e. the number of false positives of smart contracts with source code available, can be found in Table 4.4. In total 45.6% of all analysed smart contracts contain one or more security issues. This indicates that security of smart contracts can be improved significantly.

TABLE 4.5: Results analysis Maian (Nikolic et al., 2018)

Category	**#Candidates flagged *(distinct)***	**Candidates without source**	**#Validated**	**% of true positives**
Prodigal	1504 *(438)*	1487	1253	97
Suicidal	1495 *(403)*	1487	1423	99
Greedy	31,201 *(1524)*	31,045	1083	69
Total	34,200 *(2,365)*	34,019	3,759	89

4.2.2 Maian

Nikolic et al. (2018) developed Maian, which is a tool that is capable of detecting greedy, prodigal, suicidal and posthumous contracts at scale by using symbolic execution. Symbolic execution only takes the EVM code, and not the source code, of a smart contract as an input. This allowed the researchers to analyse all smart contracts that are deployed on the Ethereum blockchain. Maian analysed 970,898 smart contracts, which are all smart contracts that were deployed till block 4,799,998 (mined on December 26th, 2017).

Maian focusses on four different vulnerabilities:

Greedy Greedy contracts are smart contracts that become in a state in which the funds are locked forever. No one, including the owner, can access the funds on a smart contract.

Prodigal Prodigal contracts are smart contracts that may leak funds to an arbitrary address, allowing attackers to steal funds from these smart contracts.

Suicidal Smart contracts may contain a security feature that gives the owner the option to kill the contract. When these features are not properly implemented, arbitrary accounts might be able to call this kill function. Nikolić et al. classify these contracts as suicidal contracts.

Posthumous When a smart contract is killed, the code and global variables are removed from the blockchain. A smart contract is posthumous when it is still able to receive funds, these are then locked forever.

The results of the analyses for greedy, prodigal and suicidal accounts can be found in Table 4.5. Posthumous contracts are excluded from the table as this type of contracts can be verified without using symbolic execution. This was done by looking at smart contracts that do not contain any executable code. In total 853 smart contracts did not contain any executable code, 294 of these received funds after the contracts were killed. A disturbing fact that the researchers present is that in total 4,905 Ether is stored on suicidal and prodigal contracts. At the time of writing, the total value of this amount of Ether is around 2.3 million Dollar.

4.3 Smart Contract Security Recommendations

Keeping the vulnerabilities and the total number of vulnerable smart contracts in mind, improving the security of smart contracts is important. ConsenSys Diligence

published a list of recommendations on their website, but developers are advised to consult multiple sources. The list as of May 25, 2017 is shown in Table 4.6.

TABLE 4.6: Security recommendations smart contracts (ConsenSys Diligence, 2018c)

Level	**Recommendation**
Protocol	External Calls - Use caution when making external calls
	External Calls - Mark untrusted contracts
	External Calls - Avoid state changes after external calls
	External Calls - Be aware of the trade-offs between `send()`, `transfer()`, and `call.value()()`
	External Calls - Handle errors in external calls
	External Calls - Handle errors in external calls
	Don't assume contracts are created with zero balance
	Remember that on-chain data is public
	In 2-party or N-party contracts, beware of the possibility that some participants may "drop offline" and not return
Solidity	Enforce invariants with assert()
	Use `assert()` and `require()` properly
	Beware rounding with integer division
	Remember that Ether can be forcibly sent to an account
	Be aware of the trade-offs between abstract contracts and interfaces
	Keep fallback functions simple
	Explicitly mark visibility in functions and state variables
	Lock pragmas to specific compiler version
	Lock pragmas to specific compiler version - Exception
	Differentiate functions and events
	Prefer newer Solidity constructs
	Be aware that "Built-ins" can be shadowed
	Avoid using `tx.origin`
	Timestamp Dependence - Gameability
	Timestamp Dependence - 30-second Rule
	Timestamp Dependence - Caution using `block.number` as a timestamp
	Multiple Inheritance Caution
	(Deprecated) - Beware division by zero (Solidity < 0.4)

The use of libraries and reuse of functions can be powerful for starting developers. Consider the code example of Listing 4.3 based on the example described in the Solidity documentation (Ethereum Foundation, 2018d). To become the "richest", one needs to send more Ether to the contract than the previous person did. However, when `richest.transfer(msg.value);` invokes a fallback function that throws an exception, the smart contract becomes inaccessible. In this case the use of `richest.transfer(msg.value);` seems to be trivial but is actually dangerous. Developers are advised to reuse functions programmed by experienced developers or professional organisations to prevent these kinds of situations (Ethereum Foundation, 2018d).

```
1     function becomeRichest() public payable returns (bool) {
2         if (msg.value > mostSent) {
3             // This line can cause problems
4             richest.transfer(msg.value);
5             richest = msg.sender;
6             mostSent = msg.value;
7             return true;
8         } else {
9             return false;
10        }
11    }
```

LISTING 4.3: Vulnerable withdraw function (Ethereum Foundation, 2018d)

To keep the risks as low as possible, smart contract developers are advised to use circuit breakers, speed bumps, rate limiting, updatable smart contracts, contract roll-out plans and bug bounty programs (ConsenSys Diligence, 2018d). The first four are controls that should be implemented in the architecture of the smart contracts, while the latter two are activities that should be performed after the initial design and development.

Circuit Breaker Implementing a circuit breaker allows the developer of a smart contract to "pause" functions in their smart contract in case there is something wrong. In Solidity this can be implemented by creating a modifier that checks whether a specific variable, such as `bool private emergency`, is set to true. This modifier can then be used in functions that should not be able to be called in case of emergency.

Speed Bumps A speed bump is a method to delay actions in a smart contract. For example, it can only allow withdrawing the balance of a smart contract when there was no activity in the last 24 hours.

Rate Limiting Rate limiting is the practice of limiting significant changes in a smart contract. For example, the owner is only allowed to withdraw 5% of the total balance per day. In case a dishonest person becomes the owner of the smart contract, he will not have the ability to withdraw the total balance. This gives the actual owner of the smart contract some time to start thinking about a solution.

Updatable smart contracts A smart contract, at least the EVM code, is immutable: it cannot be changed in any way once the code is deployed on the blockchain. As pointed out in the problem statement of this thesis (section 1.2), the entities that use smart contracts need methods to update their smart contracts. ConsenSys Diligence briefly describes two approaches to allow this. The first approach is to create a smart contract that acts as a register. A user that wants to interact with the smart contract needs to look up the address of the most recent smart contract manually in the register. The second method presented by ConsenSys Diligence is to use a proxy smart contract to refer to the most recent version of the real smart contract.

Other Measures Next to the measures described above, smart contract developers should consider the following measures

- Design patterns (Destefanis et al., 2018; Coplien, 1998)
- Best practices (Destefanis et al., 2018; Dika, 2017)
- Libraries (Ethereum Foundation, 2018d)
- Testing (Destefanis et al., 2018)
- Vulnerability scanners (section 4.2)
- External security audit (Dika, 2017)
- Development methodology (Destefanis et al., 2018)
- Formal verification (Abdellatif and Brousmiche, 2018; Mavridou and Laszka, 2018)

During this research, no specific information on the use of software development life cycles models smart contract development was found. Structured processes such as waterfall or agile oriented methods, might contribute to more secure smart contracts.

Chapter 5

Existing Solution Designs

Introduction

This chapter seeks to present an overview of existing solution designs and the associated limitations. These solution designs are identified from the literature studies conducted towards blockchain, smart contracts and smart contract vulnerabilities. The results of these literature studies suggest that smart contracts with an update ability are not a topic of academic research to this date, therefore, this study is required to consult non-academic online sources. Additionally, this chapter provides insights into the update processes of blockchain protocols in order to gain a broader understanding of the challenges.

First, this chapter describes existing designs for smart contracts that have the ability to be updated, after which it elaborates on the update mechanisms of blockchain protocols.

5.1 Manual Register

In section 4.3, on recommendations for smart contract security, two approaches were briefly touched upon: a manual register and a proxy. The first approach, a manual register, requires a second smart contract that maintains a register with the address of the most recent smart contract with the core functionality ConsenSys Diligence (2018d). To use to the core smart contract, a user first needs to look up the address of the most recent core smart contract in the register smart contract. The user can then start interacting with the core smart contract using the found address. Figure 5.1 shows a visualisation of this approach.

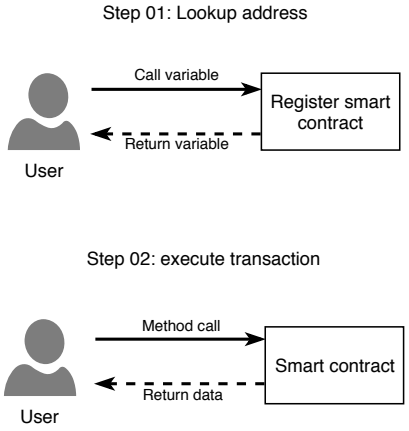

FIGURE 5.1: Manual register smart contract update mechanism

Although the approach is straightforward, it has a number of major disadvantages. First of all, as stated by ConsenSys Diligence, this approach is not efficient as it requires users to perform an extra step. Not only does this lead to a decreasing user-friendliness but it also increases the chance of potential human errors as users need to copy and paste the address manually.

The second disadvantage of this update method is that it is vulnerable to a transaction-ordering dependence attack (section 4.1). Although this type of attack could happen for every transaction, this update method adds another level of uncertainty to it. Between the time that the user looked up the address of the current version of the smart contract and actually created a transaction to that smart contract, the register could be updated with a new version of the smart contract. As such, the users have less certainty about the actual version of the smart contract. The solution also does not provide information on how the update process should be implemented and executed by, presumably, the developers.

Third, the authors do not provide information on how the owners of a smart contract should migrate data and funds to new smart contracts. This could be a major challenge given that the migration should be supported by methods inside the smart contract that needs to be updated. Depending on the situation, the functionality might also need to support the migration when the data structures of the outdated and new smart contract differ.

5.2 Proxy

The second approach touched upon in section 4.3 is an approach that allows a participant to interact with a static smart contract (ConsenSys Diligence, 2018d). The method utilises two different smart contracts: a proxy smart contract and a smart contract with the core functionality. To interact with the core smart contract, the user does not send the transaction to the core smart contract, instead, the user sends it to the proxy smart contract. The proxy smart contract maintains an internal register with the address of the most recent core smart contract. Upon retrieving a transaction, the proxy smart contract looks up this address and "forwards" the transaction to that address using a `delegatecall()`. The `delegatecall()` function supported by Solidity allows the proxy smart contract to invoke a method from the core functionality in the context of the proxy smart contract.

The developer of the smart contract could implement a feature that allows him to change the address to which smart contract the `delegatecall()` is executed. An update is then realised by changing this address to the new updated smart contract. A visualisation of this approach is shown in Figure 5.2 In contrast to the solution

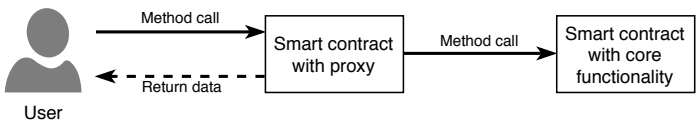

FIGURE 5.2: Proxy smart contract update mechanism

described in section 5.1, this solution enables a participant to use a static smart contract. Each method call is automatically "forwarded" to the correct smart contract. As a result, it is assumed that risks of potential human mistakes are reduced, for instance, interacting with an outdated version.

A risk of using smart contracts that adopt an update mechanism by implementing a proxy is that the participant cannot be sure which implementation is used.

Consider a proxy smart contract that uses a `delegatecall()` to invoke a method from smart contract *A*. A participant creates a transaction that invokes a method from smart contract *A* via the proxy smart contract. This transaction will not directly be executed on the blockchain, it is first placed in a pool of waiting transactions (section 4.1). Assume that prior to this transaction, another transaction is executed that changes the destination of the `delegatecall()` to smart contract *B* Now, the transaction of the participant will not invoke a method of smart contract *A*, instead, it tries to invoke the method in smart contract *B*. Although in blockchain a user is never completely sure about the state of the blockchain during the execution of a transaction, this approach results in additional level uncertainty.

Although this approach is provided by an experienced organisation, the solution is far from perfect. The Solidity documentation states that a `delegatecall()` can only be executed if the size of the returned data is known (Ethereum Foundation, 2018a). Thus, before executing the `delegatecall()`, the proxy smart contract first needs to identify the method that will be invoked and the associated size of the returned data. The Solidity documentation and ConsenSysDiligence do not provide a solution to determine these values, therefore, this solution is not considered feasible.

5.3 Limitations of Existing Designs

The two solution designs described in section 5.1 and section 5.2 have a number of different limitations. First of all, the designs do not provide the necessary information and functionality to migrate data and funds to a new updated smart contract. This functionality is required for the reason that an update of a smart contract is deployed as a new instance on the blockchain and gets a new address. The state of the new instance is not the same as the state of the outdated instance, e.g. funds stored on the old instance should be migrated to the new instance.

A second limitation of the designs is that they fail to satisfy the political decentralisation of blockchain by lacking a decision-making process; they both imply that one participant is in control (section 2.1.4). Without a decentralised approach to update a smart contract, it becomes questionable what the added value is of using a smart contract over conventional centralised software.

Third, the solution to use a `delegatecall()` as described in section 5.2 is technically inefficient. It needs to know return size of the result of the function that it invokes, however, this is unknown and for some functions variable (Ethereum Foundation, 2018a).

5.4 Blockchain Protocol Upgrades

Smart contracts run as a layer on top of a blockchain, the protocols that are used by these blockchains also require regular updates to introduce new features or patch vulnerabilities. From this point of view, it is interesting to see how this problem of performing updates in a decentralised system, is solved on the layer of the blockchain of the protocol itself. The update processes of blockchain protocols are not directly useful as an update process for smart contracts, however, it provides insights on aspects of a potential solution design. The two largest public blockchains in terms of market capitalisation, Bitcoin and Ethereum, rely on a community for their maintenance and updates. When a new feature or patch is needed for the Bitcoin blockchain,

a developer presents a Bitcoin Improvement Proposal (BIP) to this community (Bitcoin Project, 2018b). The community then starts testing this BIP thoroughly and marks the BIP as "final" after all tests are passed. Whether the BIP will actually be enforced on the network depends on the miners, only if the economic majority, i.e. hashing power, implements the BIP, it is enforced. Despite the fact that there is a separation of power between developing improvements and the actual enforcement, a number of observers argue that Bitcoin might not be as decentralised as it seems (Gervais et al., 2014; Gasser et al., 2015). The tendency of elitism and centralisation already visible in the BIP process. Albeit being open-source, only a selected group of core-developers of Bitcoin that have the technical permission to accept a BIP:

> *"in the end, decisions are made — or executed at least — by a team of core developers because only they have the technical permissions to accept submissions. Those core developers form, at least at first sight, Bitcoin's governance group in a narrower sense. Every adjustment to Bitcoin's governance structure must pass through the bottleneck of this small group of people."* (Gasser et al., 2015)

It can be argued that the group of core-developers can be kept out of the loop when developing a BIP, but a challenge that then arises is to get an economic majority to implement the BIP.

In contrast to public blockchains, consortium blockchains do not necessarily need to rely on a community for maintenance and updates. Depending on the individual implementations, these blockchains can also be maintained by the consortium of organisations that run the blockchain. At this moment in time, there is no information available on how consortia exactly operate and govern a blockchain.

Chapter 6

Design Requirements

Introduction

This chapter covers the requirements for a solution design in the form of user stories and it serves as an answer to the sub-question:

4. *What are design requirements for smart contracts that can be updated in a decentralised manner?*

The requirements are based on the limitations of the existing solution designs and use cases from section 3.3. Given the time restrictions for the design phase, the requirements are prioritised by using the MoSCoW method to achieve the maximum result.

This thesis adopts the term participant to describe a natural person or organisation who actively uses a smart contract.

6.1 Functional Requirements

Requirement 1 *The smart contract must be updated after a consensus has been reached amongst the participants.*
Rationale: This user story originates from the ultimate goal of this study: smart contracts that allow to be updated (section 1.3).

Requirement 2 *The smart contract must facilitate in a decision-making process to reach a consensus.*
Rationale: The decision-making process should not be controlled by a single participant, additionally it should be transparent and tamper-proof (section 3.1).

Requirement 3 *Participants of the smart contract must be able to propose an update of the smart contract.*
Rationale: To prevent that a single participant is in control of the smart contract to the extent that it is the only one to propose updates, the design should allow every participant to propose an update (section 5.3). This contributes to the decentralised governance of the smart contract.

Requirement 4 *The smart contract should allow participants to patch critical vulnerabilities near real-time.*
Rationale: The decision-making process of an update of a smart contract might require some time depending on the context. In the case that the update patches a critical vulnerability, it means that the current version that is used by the participants is still vulnerable.

6.2 Non-Functional Requirements

Requirement 5 *Participants must be able to use the smart contract anonymously.*
Rationale: A characteristic of public blockchains is that it enables the anonymity of its users to a certain extent (section 2.1.5). It allows entities to exchange value without the need to reveal their true identity, they are only required to share their public keys. As a consequence, the identities of the participants should not be required in a solution design.

Requirement 6 *Each participant must have a fair stake in the decision-making process on the acceptance of a proposed update.*
Rationale: No single participant should be in control of the decision-making process; every participant should thus have a fair stake in the decision-making process (section 5.3). In this thesis, a fair stake is defined as a stake that is linear with the value at risk. The influence of a participant will increase linearly with the value that a participant has stored on the smart. This principle can be compared with the voting right in a company: the number of votes of a shareholder corresponds to the number of shares he holds. Other options, such as, a stake linear with the number of transactions sent to the smart contract, or, linear to the total gas consumption, are impractical and vulnerable. These could easily be misused, thereby gaining a higher stake in the voting process. The applicability of the concept of value at risk is unknown and should thus be considered as an experiment.

Requirement 7 *The smart contract must have a static address on which it can be reached.*
Rationale: An update of a smart contract could require that a new instance of the smart contract needs to be deployed on the blockchain (section 5.2). Notifying all concerned participants about a new address is prone to error and is not user-friendly. For example, participants might still try to use the outdated version of the smart contract, while others are already using the updated version (section 5.1).

Requirement 8 *Participants must be able to invoke methods of the smart contract in one call.*
Rationale: A limitation of the existing solution design that uses a manual register is that it requires a participant to perform an additional step to interact with the smart contract (section 5.1). It is less user-friendly with respect to designs that do not require the additional step in the process. Moreover, an extra step to interact with the smart contract creates a security risk: as blockchains are state machines, it means that the state between the first step and the second step in the process might differ (section 3.2.3).

Requirement 9 *The solution design must be implemented in a smart contract.*
Rationale: The entire design must be implemented in a smart contract to ensure that it does not depend on any external software. After a consensus has been reached to update the smart contract, the actual update should be implemented without requiring any action from a single participant. This not only prevents that a single participant is in control, it also prevents that a malicious participant implements a malicious update that was not agreed upon.

Chapter 7

Case Studies Decision-Making Process

Introduction

This chapter explores designs of decision-making processes by means of illustrative case studies. The selection of cases is based on implementations of smart contracts use cases as described in section 3.3 for which the source code or a detailed description is available.

The main goal of each individual case study is to determine a fair decision-making process. These kinds of processes are well researched in the perspective of politics and economics, however, not in terms of decentralised applications. Often, these decisions are taken by an elected group of decision-makers, for instance, company boards and committees. This is not a feasible solution for smart contract decision-making processes given the anonymous nature of blockchain and smart contracts. The individual participants cannot choose other participants to which they trust their vote in the process. The aim for each case study is to design a voting system that allows each participant to have a fair stake in the decision-making process without the need to expose their identities or to trust participants.

The first case study regards an escrow service smart contract that can be used for a secure exchange of value (section 3.3.1), typically used by a small group of participants. The second case study is an implementation of an asset registry smart contract using non-fungible tokens, applicable in a number of areas such as IoT, supply chain and gaming (section 3.3.4). The third and fourth case are both implementations based on a fungible token: a fungible token used for ICO (section 3.3.2), and a fungible token used in energy trading (section 3.3.3). The key difference between these cases is found in the distribution of the tokens. In the first case, the ICO launcher retains a significant amount of tokens, whereas in the second case, the tokens are assumed to be more evenly distributed (Catalini and Gans, 2018; Kounelis et al., 2017; Lamers, 2018).

To allow extraction of a generalised model, the cases follow a similar structure. The structure was found after iterating multiple times over the initial case designs. Each iteration was used to find similarities between cases and higher-level concepts in the cases, without degrading the designs.

7.1 Case: Escrow Service

7.1.1 Case Context

Escrow smart contracts are typically used as a secure exchange of value between two parties. An implementation example of this kind of smart contracts is given in the

Solidity documentation (Ethereum Foundation, 2018h). The smart contract allows the seller to withdraw the funds after the consent of the buyer, whereby the buyer is able to control the process in the end. Advanced examples replace the confirmation of the buyer with an oracle in order to make the process fairer (Nazarkin, 2018). For this case study, it is assumed that the participants and their Ethereum accounts are static; no participants can be added, replaced, or removed.

7.1.2 Voting System Design

The Ethereum account addresses of the participants, i.e. the buyer, seller and other parties involved, are administrated in the smart contract (Ethereum Foundation, 2018h). These addresses are used to grant each participant a vote in the process. Reasoning that the smart contract is used as a secure exchange of value, thus equal value at risk for both participants, the votes have an equal weight.

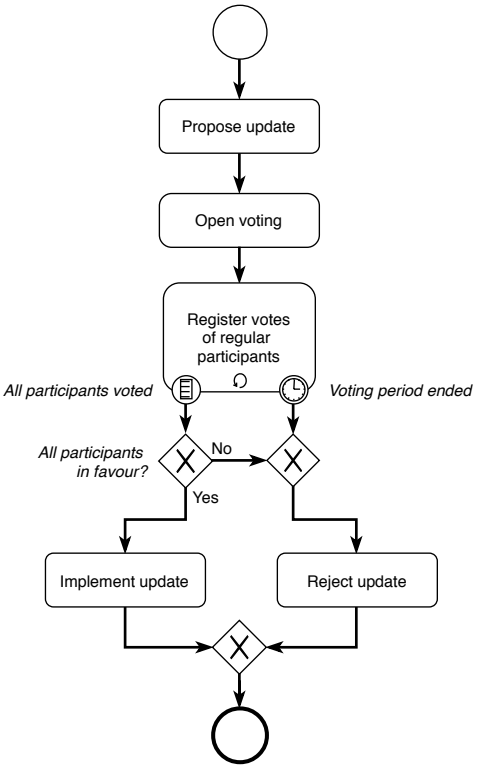

FIGURE 7.1: Process flow of a voting system for an escrow service

The voting process, as depicted in the BPMN model in Figure 7.1, is initiated after a participant has proposed an update for the smart contract. Once an update is proposed, the smart contract starts registering votes for a limited time set by *voting period*. The process of registering votes ends early at the moment each participant has provided a vote. The update is rejected in case not all participants have provided a vote before the end of the voting period. When each participant has voted before the voting period ended, the smart contract determines whether each vote is in favour of the update. Given the small number of participants, the design requires a unanimous decision in favour of the update to implement the update. The update is rejected when there is not a unanimous decision in favour of the update.

The voting period is the only parameter that can be used for this voting system to tailor it to a specific implementation.

7.1.3 Limitations

In this specific context, this voting system knows two major limitations. First, in order for the participants to make a well-based decision, they should have knowledge and experience with the programming language of smart contracts. Especially in smaller organisations there could be a lack of people that understand smart contracts well enough to make an informed decision. A lack of knowledge and experience could potentially lead to situations in which a participant proposes a malicious update that is accepted by all other participants.

The second limitation concerned with this model is that the voting process takes a certain amount of time. In case the update patches a security vulnerability, this time could be used by a malicious person to take advantage of the exploit.

7.2 Case: Non-Fungible Token Asset Registry

7.2.1 Case Context

Asset registry smart contracts are implemented in a number of different areas such as real estate and online gaming (Ubitquity, 2018; CryptoKitties, 2018). This case study examines the applicability of a voting system for the online game CryptoKitties. The game implements cats as virtual assets and allows users to buy, sell, grow and breed them (CryptoKitties, 2018). After its release on the 28th of November 2017, the number of transactions on the Ethereum blockchain increased 600%, causing congestion on the network (BBC News, 2017).

In the smart contract of CryptoKitties (2018), each cat is represented by a non-fungible token and is, therefore, a unique object (section 3.3.2). Non-fungible tokens are not equal in value, in the CryptoKitties game an average cat is sold for $23,06, meanwhile, another one was sold for $117,712.12 (BBC News, 2017). The number of participants in an asset registry smart contract is assumed to be dynamic, it might change over time. Moreover, the power of the participant that deployed the smart contract is limited. In the CryptoKitties case, the developers are the only ones that can "generate" new cats, although participants are able to breed kittens if they have a matron and sire.

7.2.2 Voting System Design

Smart contracts compliant with the ERC-721 non-fungible token standard, such as CryptoKitties, store the administration of which Ethereum account owns which token (Axiom Zen, 2017; Entriken et al., 2018). Ideally, a design grants each Ethereum account that is administrated in the smart contract a single vote with a certain weight component. The weight component would allow to use the principle of value at risk by setting the weight of a vote linear with its stake in the smart contract. An issue arises with non-fungible tokens due to the fact that the value of each token differs and that the relative value cannot be determined within a smart contract. A malicious participant could gain a higher stake in the voting process by registering a high number of low-value assets. An option to apply the same weight to each vote is not considered as desirable given that participants could easily generate a high number of Ethereum accounts.

A sub-optimal solution that is adopted in this case study, is to reduce the level of democracy by excluding a significant number of participants from the voting process. Prior to deploying the asset registry smart contract, the Ethereum accounts

that are allowed to vote are stated in the smart contract, thus excluding any new participant from the voting process. The participants that are allowed to vote could identify themselves in order to gain the trust of people that plan on using the smart contract. Despite their exclusion from the voting process, new participants could be allowed to propose an update.

The business process model for such voting system is shown in Figure 7.2 and shows resemblance with the process flow of the case described in section 7.1 on a high level. The key difference between the two models is that in this case study the decision to implement an update is not based on a unanimous vote. The number of participants allowed to vote could potentially result in a situation in which a unanimous vote is difficult. Furthermore, a unanimous vote would give a malicious participant the control to reject every update.

Once the smart contract is open to receive votes from the participants, there are two possible paths to reach the next step in the process. Either the *voting period* has ended, similar to the model in section 7.1, or the *threshold for early decision* is reached. An early decision refers to a situation in which a certain number of participants voted either in favour or not in favour. It prevents a malicious participant to delay the process on its own and it allows the update to be implemented earlier. The number set as the threshold is at least equal to the *threshold of percentage of votes in favour* of the total number of participants. In some cases, the number could be set higher for the fact that participants could switch side by casting a new vote.

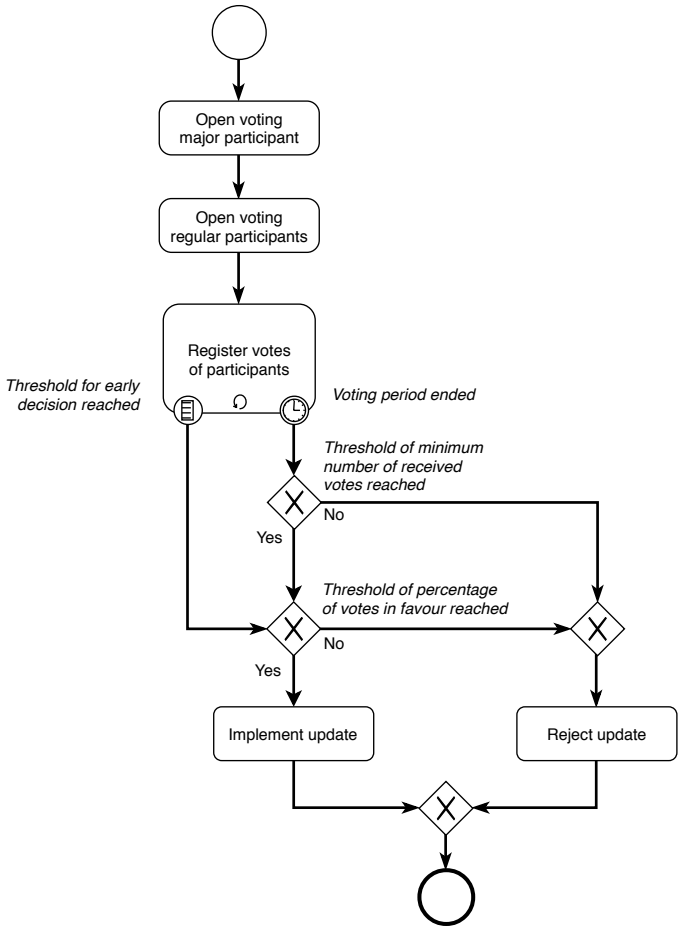

FIGURE 7.2: Process flow of a voting system for an asset registry

In case the threshold for an early decision is not reached before the end of the voting period, an additional condition needs to be met to implement the update. This condition is the *threshold of minimum number of received votes* and is used to make a widely supported decision. It prevents situations in which the decision is based on, for example, two out of eight votes. After all, withholding of voting does not imply that a voter agrees or disagrees with the update. This results in a conservative approach towards updates. Presumably, the ultimate decision to implement an update could be a majority vote. To allow the model to be generalised, it uses the *threshold percentage of votes in favour* to make the decision.

Summarising, the model can be adapted to the specific situation by means of the following parameters:

1. Voting period
2. Threshold for early decision
3. Threshold of minimum number of received votes
4. Threshold of percentage of votes in favour

7.2.3 Limitations

No Fair Stake

A significant limitation is that the decision-making process does not allow each participant to have a fair stake in the process. The model does not allow to include new participants in the voting process. Instead, this model requires a group of decision-makers who should be elected prior to the deployment of the smart contract. Furthermore, the model interferes with the requirement of anonymous use under the assumption that the identity of these decision-makers should be published to gain the trust of the other participants. The participants will not have a fair stake in terms of value at risk, hence this model should not be considered fair.

Boundary Cases

When this model would be used while taking into account the previous limitation, there is a second limitation to be found. Initially the voting power is distributed over a number of different participants. However, the model does not account for change of participants over time. For example, a participant could stop using the smart contract and could give or sell his Ethereum account to another decision maker. This allows the other decision maker to gain a higher stake in the voting process.

Another second issue that might arise is that the group of decision-makers could start to conspire. This would give them the opportunity to implement any update that would benefit them.

Next to the two major limitations above, the voting process has the same limitations that are the result of the voting process from section 7.1.

7.3 Case: Fungible Token Initial Coin Offering

7.3.1 Case Context

Many ICOs use a fungible token to raise capital for their blockchain project (Fenu et al., 2018). A smart contract that is deployed for the ICO creates a predefined number of fungible tokens and assigns these to the Ethereum account that deployed the smart contract, hereinafter referred to as ICO launcher. An Ethereum account holder can buy these tokens by sending Ether to the smart contract, in return, the smart contracts administrates that the buyer received a certain amount of tokens from the Ethereum account that launched the smart contract (Vogelsteller et al., 2018). Consequently, the number of tokens in possession of the launcher of the ICO reduces when more buyers start buying the token. A study by Catalini and Gans (2018) indicates that the ICO launchers do not sell all the tokens during the launch, the remaining tokens are sold by the ICO launcher in a later stage, for example, to finance operations and the team. Lines *#344* till *#358* of Listing 7.1 state the token distribution of VeChain (Etherscan, 2018b).

```
344     uint256 public constant totalSupply         = (10 ** 9) * (10
            ** 18); // 1 billion VEN, decimals set to 18
345
346     uint256 constant privateSupply              = totalSupply * 9
            / 100;  // 9% for private ICO
347     uint256 constant commercialPlan             = totalSupply *
            23 / 100; // 23% for commercial plan
348     uint256 constant reservedForTeam            = totalSupply * 5
            / 100;  // 5% for team
349     uint256 constant reservedForOperations      = totalSupply *
            22 / 100; // 22 for operations
350
351     // 59%
352     uint256 public constant nonPublicSupply     = privateSupply +
            commercialPlan + reservedForTeam + reservedForOperations
            ;
353     // 41%
354     uint256 public constant publicSupply = totalSupply -
            nonPublicSupply;
355
356
357     uint256 public constant officialLimit = 64371825 * (10 ** 18)
            ;
358     uint256 public constant channelsLimit = publicSupply -
            officialLimit;
```

LISTING 7.1: Token distribution VeChain ICO

7.3.2 Voting System Design

As previously described in section 3.3.2, smart contracts that implement ERC-20 compliant fungible tokens, administrate the balances of each Ethereum account in the smart contract. For similar reasons as in the case described in section 7.2, providing each Ethereum account with one vote results in an unfair voting process. Malicious participants could gain a higher stake in the voting process by dividing their tokens over multiple Ethereum accounts. However, due to the fact that ERC-20 tokens are fungible, thus are equal in value, the weight of each vote could be adjusted to the number of tokens owned by an Ethereum account. Dividing tokens

over multiple Ethereum accounts would then not result in a higher stake for a malicious participant.

A downside of this approach is that the ICO launcher owns a significant number of tokens which would allow him to steer the voting process. The distribution of votes can be defined as:

$$T = I + P,$$

where T is the total number of tokens, I number of tokens owned by the ICO launcher and P number of tokens owned by the public. If assumed that the decision is based on a simple majority vote, an inadequacy arises when $I > P$; the ICO launcher is able to control the voting process as he has the majority of the votes. In the example of VeChain, the ICO launcher has 59% of the tokens as defined in line #*352* of Listing 7.1. Situations in which $I < P$, but I is close to P, the ICO launcher could buy public tokens such that $I > P$, thereby obtaining a majority of the votes. It should be noted that the ICO launcher could distribute his tokens over a number of Ethereum accounts. Given that Ethereum accounts are anonymous and cannot be used to identify a participant, the total holdings of the ICO launcher cannot be calculated as the sum of the tokens held by the Ethereum accounts within a smart contract.

A method to prevent a participant from obtaining a majority of the voting power, is to set a limit on the maximum stake of an Ethereum account. Unfortunately, a disadvantage of this method is that a malicious participant could divide his tokens over multiple Ethereum accounts, thereby not exceeding the maximum stake. A more secure and conservative approach for the ICO launcher is to divide the voting process in two rounds, in the first round solely the ICO launcher is allowed to cast a vote and in the second round all participants that own public tokens with the exception of the ICO launcher. The ability of the ICO launcher to control the election is partially eliminated, an update is only implemented if the ICO launcher and the owners of the public tokens agree. Regardless of I and P, the ICO launcher has veto power; he has the authority to reject any update, but he is not able to implement an update without the consent of the participants. It prevents a malicious ICO launcher to implement bad code unless the owners of public tokens agree. The other way around, a malicious participant that owns a significant number of the public tokens will not be able to implement bad code, unless the ICO launcher agrees. Only in the situation that the ICO launcher buys a significant amount of the public tokens, the ICO launcher can control the election. Considering the usage of the smart contract and the role of the ICO launcher, this voting process is considered reasonable.

Figure 7.3 displays a voting process that is used for this case study. The implementation of a second round of votes increases the complexity of the process in comparison with models that only require a single round. Similar as the previous case studies, the first step in the process is for a participant to propose an update. The smart contract initiates the first voting round that allows the ICO launcher to cast his vote. In case the ICO launcher does not vote in favour of the update or fails to vote before the voting period has ended, the update is rejected. After the ICO launcher voted in favour of the update, the smart contract will open the second voting round. The second round of voting follows a similar process as described in section 7.2.

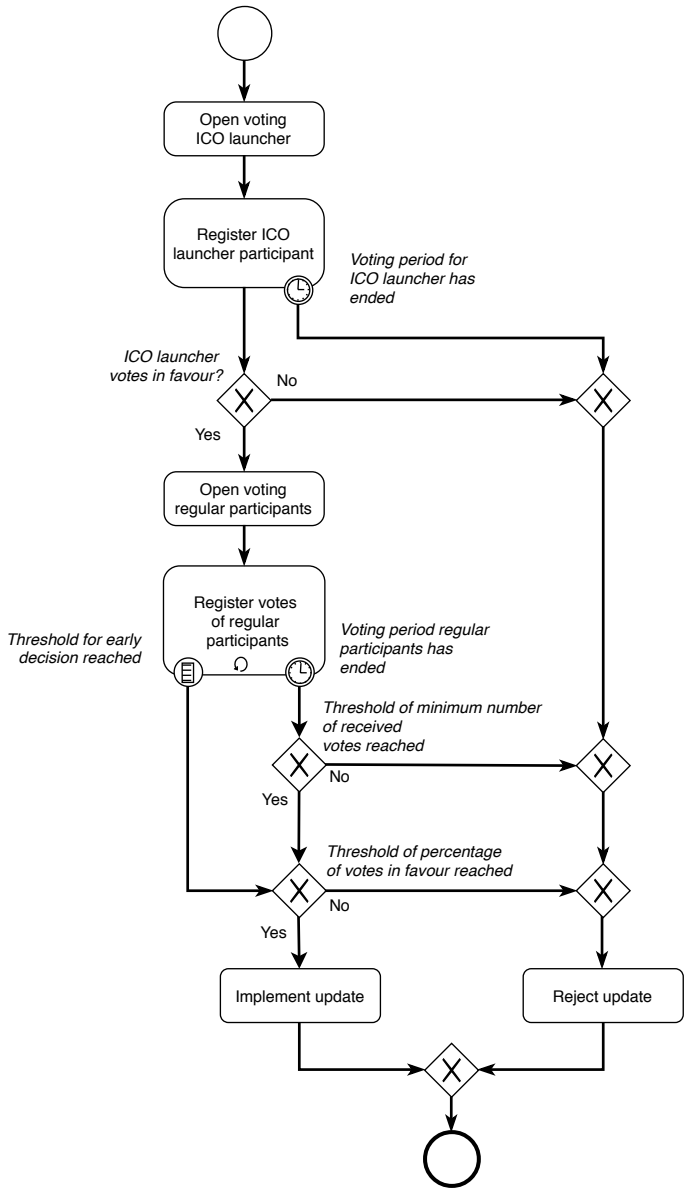

FIGURE 7.3: Process flow of a voting system for an ICO

The entire voting process can be tailored to the specific context by using four parameters:

1. Voting period ICO launcher
2. Voting period regular participant
3. Threshold for early decision
4. Threshold of minimum number of received votes
5. Threshold of percentage of votes in favour

7.3.3 Limitations

Voting Period

The current model implies that both rounds of voting are consecutive, which delays the decision-making process. Although not necessary, the rounds are implemented as consecutive in order to prevent regular participants from wasting their transaction fee in case the ICO launcher does not vote in favour. As a result of this, the total period until the final decision is increased. This is considered as a limitation of the model while it has a negative impact on the ability to implement updates for critical vulnerabilities.

Fair Stake

It could be argued that the authority of an ICO launcher to solely reject an update could be seen as unfair. Given the specific context, we think it is reasonable for an ICO launcher to have this authority for the fact that the ICO launcher used the smart contract as a method to raise capital in order to finance their project. It is assumed that a participant that invests in these projects has a certain level of trust in the ICO launcher.

Boundary Cases

In regular conditions, the described model is expected to reach the desired outcome in a fair way. While participants are allowed to exchange tokens, i.e. a change of value at risk, boundary cases might occur after a certain time. A small number of participants could obtain a significant amount of the tokens. In these situations, the model could potentially result in unexpected and undesirable outcomes that the model does not account for. The model initially accounts for this in the situation of the ICO launcher, however, it is not taken into account for regular participants.

Consider a distribution of public tokens between three groups in the round for regular participants: participant A with 47%, participant B with 46%, and other small participants with 7% cumulative. Participant A and B do not necessarily need the consent of the other, they both only need to get a consent from the other small participants to reach a majority. Although the outcome is the result of a democratic decision-making process, it is arguable whether it should be considered as a desired situation.

Given that the voting system in this specific implementation consists of two separate voting rounds, a participant with a significant amount of the public tokens still requires the consent of the ICO launcher to approve an update.

Next to the limitations above, the limitations described in section 7.1 should also be considered for this voting process given that the same process is partially used.

7.4 Case: Fungible Token Energy Trading

7.4.1 Case Context

A number of publications on the possibilities of blockchain technology expect energy markets to benefit financially from smart contracts (Kounelis et al., 2017; Lamers,

2018). In this illustrative case study, a voting process is designed for an energy trading system, developed by Kounelis et al. (2017), that adopts fungible tokens to represent consumed and produced energy in a smart contract.

The system requires a physical device connected to a smart meter to be able to measure the amount of energy produced and consumed, and, to send this data to a smart contract that creates a virtual representation of the energy. The smart contract mints and rewards tokens to energy producers, in turn, consumers need to buy these tokens to be able to consume energy. Kounelis et al. (2017) envision these kinds of smart contracts to be deployed and managed by a consortium in which all participants of the electric system are united. The study lacks evidence about the organisation of the consortium, as consequence, in this thesis it is assumed that the organisation is not recorded in a smart contract. Additional assumptions are that only a single Ethereum account has the capability to apply changes to the smart contract and that the consortium is the only entity with access to this Ethereum account. The decision-making, in this case, is decentralised by means of the consortium, however, it is inefficient and prone to malicious use. Any participant within the consortium with access to the Ethereum account can interfere with the smart contract without official legitimate consent from the other participants united in the consortium.

7.4.2 Voting System Design

The voting system that could be applied to this case study is similar to the voting process model in section 7.2 with a major difference. In this case, all participants are included in the voting process and the number of fungible tokens held by an Ethereum account is used to apply a weight to a vote. This allows every participant to have a stake in the voting process. Fungible tokens have the same value and can, therefore, be used to determine the value at risk for each Ethereum account. It is assumed that, in contrast to the case study in section 7.3, no participant possesses a significant number of tokens, therefore, there is no need to divide the voting process into two rounds. While this process is equal to the process from section 7.2, it can be adjusted to the specific context with the same parameters.

7.4.3 Limitations

Boundary Cases

For the same reason as explained in section 7.3.3, this model could potentially reach a boundary case. Consider that the decision is based on a simple majority vote and a distribution of tokens between three groups in round B: participant A with 47%, participant B with 46%, and other small participants with 7% cumulative. Participant A and B do not necessarily need the consent of each other, they both only need to get a consent from the majority of the small participants. In contrast to the situation described in section 7.3.3, this implementation relies on a single round of votes. The two large participants could in these situations propose an update and persuade the small group of participants to give their consent.

This boundary condition could be prevented by setting a maximum limit on the weight of a vote. By setting a limit of, for example, 20%, large participants cannot easily reach a majority. In contrast to the previous case study in section 7.3, in this case the participants cannot divide the tokens over a number of Ethereum accounts because the accounts are linked to physical devices that measure the consumed and produced energy. In terms of the definition of fairness that is used in this study, this should not be considered as a democratic decision-making process.

Next to the limitation above, the voting system has the same limitations as the case study from section 7.3 given that the voting processes are similar.

7.5 Findings

The illustrative case studies indicate that the decision-making process could be implemented by means of a voting system. For three of the four case studies, a voting system could be designed that empowered participants to have a fair stake in the decision-making process. In these case studies, fairness was determined by means of value at risk: the weight of a vote of a participant is linear with the value at risk. There is no single process that suits all implementations of smart contracts.

When designing a feasible voting process, a key aspect that should be considered is whether the value at risk of a participant could change once the smart contract is deployed. In other words, does the smart contract require a method to determine the weight of a vote in a dynamic manner. The case studies indicate that two main drivers for a change in the weight of a vote are a dynamic group of participants and fungible tokens that can be traded with other participants. In the case study on the escrow service (section 7.1) neither the number of participants nor the value at risks change over time. This enables to implement a straightforward voting process in which no boundary cases are likely to occur. In the three other cases, the value at risk of a participant could change and therefore require a more complex voting process. To enable a fair process in terms of value at risk for these case studies, it is required that the smart contract is capable of determining the relative values for each individual Ethereum account. The case studies indicate that this is possible for fungible tokens (section 7.3; section 7.4) and not for non-fungible tokens (section 7.2).

In the case study on Fungible Token Initial Coin Offerings (section 7.3), it was evident that one participant would have a majority. In that specific case it was not necessarily an issue for the fact that the Ethereum accounts of this participant were known prior to the deployment of the smart contract. These Ethereum accounts were then used to implement an additional round of voting in which only the major participant is allowed vote. While considering the role of the participant with the majority, the ICO launcher, the additional round of votes is reasonable. Unfortunately, this case reveals a major limitation of voting-based processes in smart contracts: participants could gain a significant stake in the voting process without the knowledge of the other participants. In the case studies on fungible tokens (section 7.3; section 7.4) it became clear that the tokens can be exchanged with other participants. This allows a malicious participant to increase his tokens holdings and thereby his stake in the voting process. Thus, initially the model might perform as expected, but over time a centralisation of the voting power cannot be prevented. A problem here is that a participant can use multiple accounts to interact with the smart contract and that these accounts cannot be linked to the individual participant.

Next to the limitation described above, a decision-making process based on voting has a number of other limitations depending on the situation. As shown in Table 7.1, two major limitations can be found in each case study. First, each participant should have sufficient knowledge of the programming language in order to make a well-informed decision. Second, voting-based processes depend on the input of participants and thus require a certain amount of time.

TABLE 7.1: Overview of limitations found in case studies

Limitation	**Escrow service**	**Asset registry**	**Initial coin offering**	**Energy trading**
Lack of knowledge	×	×	×	×
Time consuming	×	×	×	×
No fair stake (initially)		×		
Boundary cases		×	×	×

Chapter 8

Solution Design

Introduction

This chapter provides a detailed description of the solution design and provides the answer to the following two research questions:

5. *What is a design for smart contracts that can be updated in a decentralised manner?*
6. *Does the designed artefact work as desired?*

The design consists of two key elements: a decision-making process to reach consensus on an update and a technical element to facilitate the ability to update a smart contract.

The chapter starts with a generalisation of the voting process that was extracted from the case studies from chapter 7, after which the technical element is discussed in detail. The two sections that follow will discuss the financial implications, limitations of the artefact. The last two sections cover the verification of the requirements and the validation of the results.

8.1 Generalised Decision-Making Process Model

Figure 8.1 presents the generalised model for the decision-making process which was extracted from the individual illustrative case studies. The model is initiated by a participant who proposes an update for the smart contract. Then, the model makes a distinction on the fact whether a single participant has a significant stake. This participant, hereinafter referred to as a major participant, is a participant that would be able to solely control the outcome in a single round voting system, i.e. he who has a stake higher than the *threshold of percentage of votes in favour*. In case it is evident that there is a major participant whose addresses are known, the model requires an extra round of votes in which only that participant is allowed to vote, as shown in section 7.3. On the one hand, it enables the major participant to reject any update, on the other hand, it does not enable the major participant to force any update. The Ethereum address, or addresses, of the major participant, need to be identified and stated in the smart contract in order to grant the votes, therefore, this should be determined by the developers prior to the deployment of the smart contract.

In case there is no major participant, the model implements a single round of votes during which regular participants are allowed to vote. This part of the process shows resemblance with the process described in section 7.2. Once the smart contract is open to receive votes from regular participants, there are two paths to the next step in the voting process: either the voting period has ended, or the *threshold for early decision* is reached. An early decision refers to a situation in which a certain

number of participants voted either in favour or not in favour. It allows the update to be implemented earlier. The number set as threshold is at least equal to the *threshold of percentage of votes in favour* of the total number of participants. In some cases, the number could be set higher for the fact that participants could switch side by casting a new vote.

If the voting period has ended and the *threshold for early decision* has not been reached, the process requires a certain percentage of votes received to make a widely supported decision. This is implemented to prevent that a decision to update the smart contract is based on a small group of the participants, as explained in section 7.2. Depending on whether the *threshold of percentage of votes in favour* is reached, the update is either implemented or rejected. Thus, the *threshold of percentage of votes in favour* allows to set what percentage of the votes need to be in favour to accept an update. A full democratic percentage of this would be more than 50%: a simple majority. However, it can be argued that a minimal majority also means that close to 50% of the participants do not agree with the update. This thesis does not engage in a study on the exact percentage that should be used.

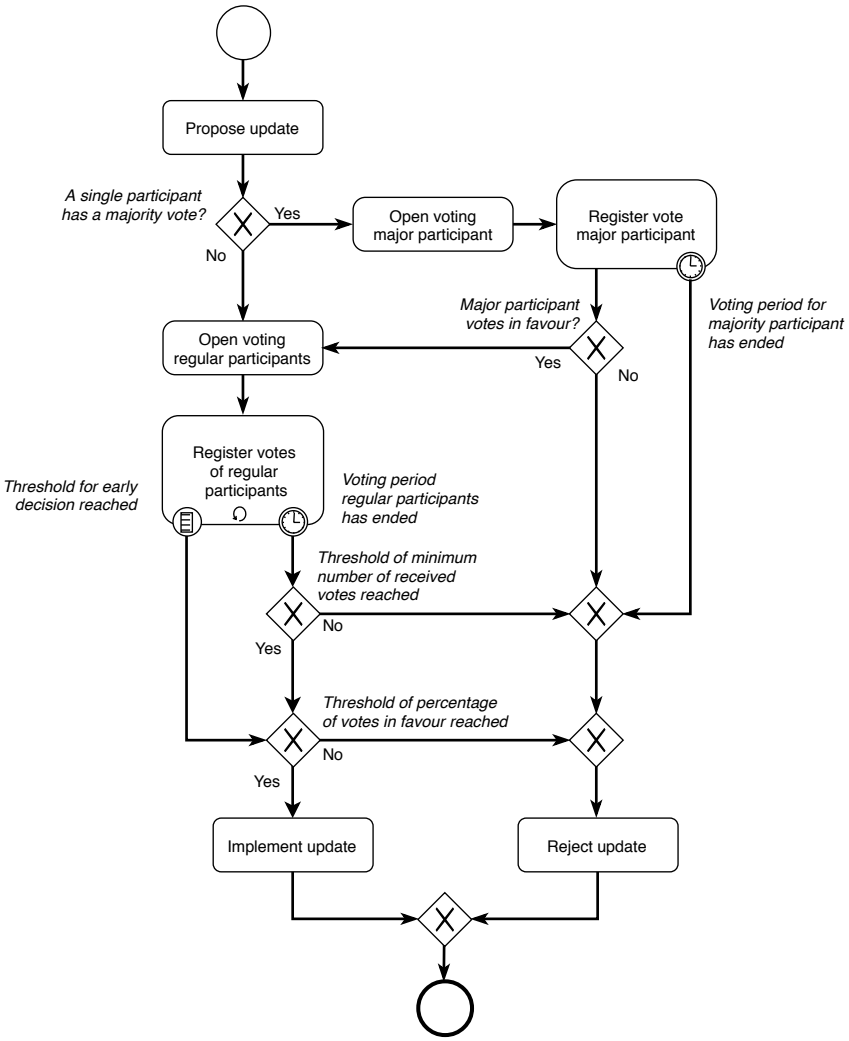

FIGURE 8.1: Generalised decision-making process

Summarising, the model allows to be tailored to the specific context by means of five parameters for a two-round voting system, and four parameters for a single round voting system. The parameters for the single round voting system are:

I *Voting period for regular participants*
This parameter is used to set the term during which regular participants are allowed to vote.

II *Threshold for early decision*
The decision to update can be made before the end of the voting period. The number should at least be the *threshold of percentage of votes in favour* of the total number of participants, or higher.

III *Threshold of minimum votes reached*
This threshold allows to set a minimum number of required votes upon which the decision to update is based for regular participants. The design takes a conservative approach in the decision-making process, the absence of a vote from a participant does not imply that the participant agrees on the update.

Two-round voting systems can be adjusted by the previous parameters with the inclusion of:

IV *Voting period for major participant*
This parameter is used to set the term in which the major participant is allowed to vote. The update is rejected when the major participant fails to vote within this term.

8.2 Technical Design

Figure 8.2 shows a high-level architecture of the smart contracts and their components. The solution design divides the required functionality over two separate deployed smart contracts, whereby the first smart contract serves as a "proxy" to the second smart contract "controller". The main methods needed to fulfil the initial purpose of the smart contract are thus implemented in the controller.

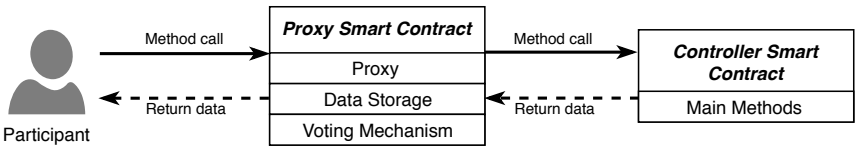

FIGURE 8.2: Overview smart contract update mechanism

8.2.1 Proxy Smart Contract

The proxy acts as a gateway to the functionality of the controller, furthermore, it implements features for data storage and orchestrating the voting process. The existing solution design described in section 5.2 suggested that a proxy smart contract could be used as a gateway to the core smart contract, however, an analysis showed that the implementation of that design was not considered as feasible due to limitations of Solidity. This solution design takes a similar approach while avoiding the limitations of Solidity.

Proxy

As said, the methods needed to fulfil the initial purpose of the smart contract are implemented in the controller. To call these functions, the proxy implements a fallback function that executes a custom implementation of `delegatecall()`. As explained in section 3.2.4, the fallback function of the proxy is triggered when a non-existent method is invoked. The custom implementation of `delegatecall()` allows to call methods from the controller in the context of the proxy, thereby avoiding the key obstacle of a regular `delegatecall()` of not returning data (section 5.2). The solution design relies on assembly code to implement the custom of `delegatecall()` function (Elena and Spagnuolo, 2018). Assembly is a fine-grained and low-level code that can be used in Solidity and that directly targets the EVM (Ethereum Foundation, 2018i). Listing 8.1 provides an example of the custom implementation that allows to call methods from an external smart contract and to return the results.

The assembly code first determines the next free memory slot that can be used to store the result, this pointer is stored by EVM on the position `0x40`. The location is then used to store the transaction call data. After that, the function executes a `delegatecall()` and overwrites the previously stored transaction call data with the returned data.

The custom implementation of this function contains a variable called `proxydestination` which states the address to which smart contract the `delegatecall` is executed. By implementing a method that allows to update the value of `proxydestination`, an update can be realised. It allows a user to point the `delegatecall()` to a new instance of the controller, thereby updating the smart contract. As a result, solely the controller can be updated, not the proxy. Therefore, it is recommended to implement most of the functionality in the controller. The address of the proxy is static and is used for every transaction to interact with the smart contracts. An update of the controller does not change the address of the proxy.

```
1   function () public payable {
2       address proxyaddr = proxydestination;
3       assembly {
4           let freememorystart := mload(0x40)
5           calldatacopy(freememorystart, 0, calldatasize())
6           let success := delegatecall(not(0), proxyaddr,
                freememorystart, calldatasize(), freememorystart, 0)
7           returndatacopy(freememorystart, 0, returndatasize())
8           switch success
9           case 0 { revert(freememorystart, returndatasize()) }
10          default { return(freememorystart, returndatasize()) }
11      }
12  }
```

LISTING 8.1: Fallback method with a custom delegatecall()

Data Storage

Although the proxy seems to be a novel approach to implement an update mechanism, it creates a major challenge when it comes down to development. The challenge is to ensure a compatible memory layout of the proxy and the controller. In a compiled smart contract, class variables, among other things, point to the memory slot in which the EVM stores the value (Wood and Ethereum Foundation, 2018). As

TABLE 8.1: Memory layout example

Memory Slot	Proxy	Controller	Controller (Compatible)
0	proxyDestination	variableA	proxyDestination
1		variableB	variableA
2			variableB

the proxy executes functions of the controller in the context of its own, the memory slots point to the wrong locations.

Consider the two smart contracts, *Proxy* and *Controller*. *Proxy* contains a variable `proxyDestination`, which is used to save the address of the *Controller*. In *Controller* two random variables are defined: `variableA` and `variableB`. In this case, when trying to retrieve `variableA` via *Proxy* from *Controller*, the proxy returns `proxyDestination`. This happens because whenever the *Proxy* retrieves `variableA` from memory slot 0 of *Controller*, the EVM returns memory slot 0 of its own memory, which returns `proxyDestination`.

Solving this requires that the memory slots used by *Controller* are not overlapping with the memory slots used by *Proxy*. Technically this is done by defining the variables used by *Proxy* in a separate "contract class" (*ProxyData*). Inheriting *ProxyData* contract class in the *Controller* contract class before compiling and deploying, automatically ensures that the used memory slots do not point to memory slots used by *Proxy*. Using the previous example, the memory slots are then used as shown in the column "Controller (Compatible)" from Table 8.1. It should be noted that all data that needs to be stored, including data of the *Controller*, is stored in the Proxy.

Keeping the memory layouts compatible can result in major challenges over time when a number of updates are implemented. A useful method to mitigate these risks is by storing persistent data in mappings, referred to as eternal storage (Lemble and Ethereum Foundation, 2018). Each mapping stores multiple key-value pairs for a specific data type, such as `bool` and `string`, and always refers to the same point. Additional get, set and delete functions are required to store data in the mapping. Listing 8.2 shows an example of the functions to use an address mapping, it should be implemented in the proxy smart contract.

Although storing variables using this approach is not how variables supposed should be stored, it allows dynamic usage of variables without analysing the memory slots prior to an implementation. It thus reduces the risks of implementing faulty updates that corrupt the memory layout.

Albeit the fact that the memory slot poses a major challenge during the development of the smart contract, there is a crucial benefit: data does not need to be migrated after an update.

```
1     mapping(bytes32 => string)      private stringStorage;
2
3     function getString(bytes32 _key) external view returns (string)
          {
4         return stringStorage[_key];
5     }
6
7     function setString(bytes32 _key, string _value) external {
8         stringStorage[_key] = _value;
9     }
10
11    function deleteString(bytes32 _key)  external {
12        delete stringStorage[_key];
13    }
```

LISTING 8.2: Data storage implementation for string

Voting Mechanism

The third core element of the proxy smart contract are features to orchestrate the voting process. A number of functions are required to support the voting process as identified in section 8.1. These functions are for a user to propose an update, to allow participants to vote, to make the decision, and to activate the update.

First, it implements a method that allows participants to propose an update. In order to implement a reliable and secure voting mechanism, the participant should deploy the update, i.e. a new controller, on the blockchain, prior to invoking the method. This ensures that code for the update is immutable and thus cannot be changed during the voting process. The method to propose an update triggers an event to notify the participants of the smart contracts. After an update is proposed, this method is temporarily disabled to ensure that updates do not interfere with each other. Interfering updates could lead to corrupt data storage.

Once an update is proposed, the proxy starts registering votes. A participant can cast a vote by sending a transaction to the proxy that invokes the method to register his vote. The method in the proxy determines the weight of the vote using a method implemented by the developers, for instance, the number of fungible tokens hold by the Ethereum account of the participant. A participant with multiple Ethereum accounts is granted one vote per Ethereum account. In case a previous vote from the Ethereum account has been registered, the vote is updated with the new vote. The method that registers the votes of the participants computes the current outcome after each invocation in order to allow the voting mechanism to take the decision to update before the voting period has ended. It is key to notify the participants about an early decision to allow them time to implement the correct functions to interact with the updated controller. This could be done by broadcasting the block number during which the updated controller becomes active.

If no decision is taken before the end of the voting period, the proxy counts the votes in favour of the update. Using these values, the smart contract takes the final decision by reflecting the values on the required conditions from the parameters in section 8.1. The proxy determines a block number in which the update will become active. Once the blockchain reaches that block number, the proxy automatically enforces the update by changing the `proxydestination`.

8.3 Financial Implications

Implementing an update mechanism and a data storage will increase the costs of using the updatable smart contract as each transaction that invokes methods from the logic smart contract requires extra computational operations. Another aspect that increases the costs is the voting process; the process relies on transactions in order for participants to cast their vote.

For the calculation of costs below, an average gas price of $24.13 * 10^9$ Wei is used. This was the average price over the last 365 days, from the 16^{th} of August 2017 till 15^{th} of August 2018 (Etherscan, 2018d). Over the same period, the average price of 1 Ether was $556.56. It should be noted that 1 Ether is equal to $1 * 10^{18}$ Wei. Furthermore, it should be noted that both prices can fluctuate significantly, for example, the maximum price of Ether over the last 365 was $1385.02 and the lowest price was $223.14. The price trend of Ether is shown in Figure A.2 in Appendix A.

8.3.1 Costs Proxy

To estimate the costs of adopting a proxy mechanism, an implementation of a proxy smart contract and a controller smart contract were deployed on the Ethereum blockchain and analysed. The implementation allowed to set four different data types: `string`, `unsigned integer`, `address`, and `boolean`. It should be noted that the implementation did not use the data storage method described in section 8.2.1 in order to determine the net costs of only the proxy.

Table 8.2 shows a comparison of the total transaction gas consumption between a direct invocation of a method and an invocation via the proxy. The results indicate that the proxy mechanism increases the total gas consumption for the methods with at least 3.12%. As such, the average costs increment for the four methods approximately is $0.014. Given this minor change, this thesis does not engage in an in-depth analysis of this cost increment.

TABLE 8.2: Comparison of transaction gas consumption of proxy

Method	Gas consumption direct call	Gas consumption call via proxy	Cost increase by proxy
setString()	33,297	34,335	3.12%
setUint()	26,603	27,629	3.86%
setAddresss()	28,155	29,181	3.64%
setBoolean()	26,925	27,951	3.81%

8.3.2 Costs Data Storage

To estimate the costs for using the data storage, a basic implementation of the smart contract was deployed on the blockchain. The implementation was similar to the Listing 8.2 and did not use a proxy fallback method in order to ensure that the cost increase is only the result of the data storage approach. The implementation supported to store four data types: `string`, `unsigned integer`, `address`, and `boolean`.

By using the data storage approach, the total gas consumption of a transaction increases between 3.83% and 5.83%, as shown in Table 8.3. As a result, the average cost increment for using the methods is approximately $0.020. Given this minor change, this thesis does not engage in an in-depth analysis of this cost increment.

TABLE 8.3: Comparison of transaction gas consumption of data storage

Method	Gas consumption without data storage	Gas consumption with data storage	Cost increase by data storage
setString()	33,017	34,283	3.83%
setUint()	26,603	28,155	5.83%
setAddresss()	28,227	29,754	5.41%
setBoolean()	26,845	28,353	5.62%

8.3.3 Costs Voting

The decision-making process relies on transactions, not merely to allow a participant to propose an update, also for participants to register their vote. As a result, the entire decision-making process can become expensive depending on the number of participants. The participant who wants to propose an update solely needs to pay in order to deploy the new controller smart contract. Depending on the size of the logic smart contract, this could lead to significant costs for the participant. Additionally, he needs to pay for a transaction to propose the update to the other participants.

Once an update is proposed, each participant is required to place a transaction in order to cast a vote. In the most basic implementation of a function that allows registering a boolean `inFavour` in a data storage, each transaction will cost at least 42,095 gas, worth $0.565.

8.4 Limitations

8.4.1 Decision-Making Process

A major limitation of a voting-based decision-making process is that the process can consume a certain amount of time depending on the voting period(s) set. In case an update is proposed in order to solve security vulnerabilities, this period allows for a malicious participant or other users to exploit the vulnerability. In a two-round voting system, the total time of the process can be reduced by opening the two rounds simultaneously, although this approach could result in the fact that regular participants waste their fees (section 7.3).

Second, each participant is required to understand the programming language of smart contracts in order to make a well-considered choice. This is not solely a limitation of a voting-based system, it holds for any decision-making process that relies on the input of participants. Alternatively, a group of decision-makers could be elected to which the participants could delegate their vote, although, it would contradict the anonymous nature blockchain.

The third limitation of the generalised decision-making process is that it is not feasible to be applied in every implementation of smart contracts, for example, implementations that use non-fungible tokens (section 7.2). The decision-making process relies on the value at risk of a participant to allocate voting power over the participants, hence, it requires that the value at risk for each participant can be computed and compared.

Last, due to the fact that participants can exchange fungible tokens, boundary cases such as centralisation of the voting power, can occur over time. Initially the model might perform as expected, but it cannot be guaranteed that it will perform as expected in a later stage. The underlying issue here is that a participant can use

multiple accounts to interact with the smart contract. The addresses of these accounts cannot be linked to the individual participant, thus the smart contract is not capable of detecting centralisation of voting power.

8.4.2 Adoption Strategy

As a result of the state machine principle of blockchain, the update of a smart contract follows a big bang adoption strategy; the update is enforced during a single block by changing the proxy destination. It could result in situations in which an uninformed participant places a transaction that is not compatible with the current implementation. Although the participant only loses the paid transaction fee for a reverted transaction, it could lead to problematic results depending on the implementation. From the perspective of a participant, the solution lacks user-friendliness as each individual participant is required to update their manual process or client-side application. In an ideal situation, a client-side application is provided that automatically ensures that the transactions are compatible with the smart contract or notifies that the transaction is incompatible before a transaction is placed.

For the technical-design, parallel and phased update approaches were considered to improve the user-friendliness. It was found that in order to support this, two different versions of the state of a smart contract need to be maintained: one of the updated version and one of the outdated version. This not only results in higher transaction fees due to a higher number of computations, but it also faces difficulties when both states are not the same. As the updates change logic and thus the results of the smart contract, the states will become different over time. Neither of the states can be considered as the real one, they will both be valid results of the different versions of the implemented logic.

8.4.3 Downtime

During the block that the update of the smart contract is executed, i.e. `proxy-destination` is updated, the smart contract is open to receive any incoming transactions. As a result, during that specific block number, the smart contract is vulnerable for a transaction-order dependence attack (section 4.1). That is, three transactions can be processed in a block, however, the order in which they are included is unknown. A transaction of a participant to the smart contract could either be directed to the previous version or to the new version, there is no assurance that the proxy destination is updated prior to the transaction of the smart contract. Although the smart contract does stay open, it is suggested that the block number in which the update is implemented should be considered as downtime. Participants should be prevented from sending transactions in that block or all incoming transactions triggering the fallback function should be ignored. A fallback method that provides the proxy functionality could be programmed such that it is disabled during the block number of the update.

8.4.4 Update Publicly Visible

The voting process is started after a participant proposed an update In order for the other participants to make a well-informed decision, the source code of the update should be published. It could be argued that this is not desirable for updates that implement a patch to solve a security vulnerability. A malicious user could analyse

the source code of the update to find and exploit the security vulnerability while the decision-making process is in progress.

8.4.5 Hard Fork

A blockchain can become the victim of a hard fork. A hard fork means that nodes in the network mine on top of different chains of blocks (section 2.3.7). This could happen at any time when nodes do not agree with each other. The impact of a hard fork on the solution design depends on the timing of the hard fork and on which transactions are processed. A hard fork results in two or more different chains of blocks. When it occurs during the voting process, the chains do not necessarily include the same transactions, thus they can include different votes. As a result, the outcome of the voting process potentially differs between the blockchains.

The main issue of a hard fork is that there will be different valid chains of blocks, thus different versions of what is considered "reality". An implication of this is that the smart contract will exist on two different blockchains and therefore could have different data. Hence a hard fork might result in conflicting instances. This thesis does not engage in exploring measures and solutions to mitigate hard forks.

8.5 Verification of Requirements

This section verifies whether the designed artefact satisfies the requirements for a solution design presented in chapter 6.

Table 8.4 presents an overview of the individual requirements and their level of satisfaction in the design. Eight out of the nine requirements were considered as a "must have". The results show that only five of them have been satisfied completely, the remaining three are partially satisfied. One requirement was stated as a "could have" for the solution design.

For each requirement, an explanation of the satisfaction is presented below.

Requirement 1: ***The smart contract must be updated after a consensus has been reached amongst the participants.***
The solution design partially satisfies this requirement to the extent that a different version of the controller smart contract can be implemented (section 8.2.1). After a consensus has been reached, the update of the controller smart contract will automatically be implemented. However, the design does not support a complete update of the smart contract due to the fact that the code in the proxy cannot be changed. This leads to the conclusion that this requirement is not completely satisfied. It is recommended to implement as much logic as possible in the controller smart contract.

Requirement 2: ***The smart contract must facilitate in a decision-making process to reach a consensus.***
This requirement is satisfied by the voting mechanism that is implemented in the proxy smart contract (section 8.2.1).

Requirement 3: ***Participants of the smart contract must be able to propose an update of the smart contract.***
Both the generalised decision-making process and the technical design support for a participant to propose an update. The generalised decision-making process uses

this step in order to start the voting process (section 8.1). This is implemented in the voting mechanism in the proxy smart contract (section 8.2.1).

Requirement 4: ***The smart contract should allow participants to patch critical vulnerabilities near real-time.***
The generalised decision-making process only allows the update to be implemented after a voting process (section 8.1). Considering that the participants need to cast their vote individually, the entire process consumes a certain amount of time. This means that the critical vulnerability will be exploitable for the period of the voting process. Hence it can be concluded that this requirement is not satisfied.

During the design of the artefact, a potential solution to satisfy the requirement was identified. A promising solution seems to be to allow participants to temporary disable the main functionality of the smart contract except the voting system. Assuming that the critical functionality is to be found in the main functionality, a malicious participant will then not be able to exploit the vulnerability. Once the main functionality is disabled, the participant could propose a new update that, after reaching a consensus, is implemented and that activates the main functionality. Key to this solution is to prevent malicious participants from misusing the disable functionality. Further research on this potential solution is required to determine its feasibility.

TABLE 8.4: Satisfaction requirements

Requirement	**Priority**	**Satisfaction**
Requirement 1: *The smart contract must be updated after a consensus has been reached amongst the participants.*	Must	Partially satisfied
Requirement 2: *The smart contract must facilitate in a decision-making process to reach a consensus.*	Must	Satisfied
Requirement 3: *Participants of the smart contract must be able to propose an update of the smart contract.*	Must	Satisfied
Requirement 4: *The smart contract should allow participants to patch critical vulnerabilities near real-time.*	Could	Not satisfied
Requirement 5: *Participants must be able to use the smart contract anonymously.*	Must	Satisfied
Requirement 6: *Each participant must have a fair stake in the decision-making process on the acceptance of a proposed update.*	Must	Partially satisfied
Requirement 7: *The smart contract must have a static address on which it can be reached.*	Must	Satisfied
Requirement 8: *Participants must be able to invoke methods of the smart contract in one call.*	Must	Satisfied
Requirement 9: *The solution design must be implemented in a smart contract.*	Must	Satisfied

Requirement 5: ***Participants must be able to use the smart contract anonymously.***
A participant can cast a vote by placing a transaction from his Ethereum account (section 8.2). Hence, the artefact does not require a participant to provide any information about his identification.

Requirement 6: ***Each participant must have a fair stake in the decision-making process on the acceptance of a proposed update.***
This requirement was studied by designing decision-making processes for four different illustrative case studies (chapter 7). The results show that the satisfaction of this requirement depends on the use case of a smart contract. For implementations in which the value at risk cannot be computed or compared, a fair stake in the decision-making process is not possible (section 7.2). The requirement is satisfied in implementations in which the value at risk for each Ethereum account is equal or allows to be computed, such as, escrow smart contracts and fungible-token smart contracts (section 7.1; section 7.3; section 7.4).

To conclude, the satisfaction of this requirement depends on the definition of a fair stake, which was the value at risk in this thesis. The generalised process that was presented in this chapter can thus only be applied in situations in which the value at risk is fixed, i.e. does not change after deployment, or when the (relative) value can be determined within the smart contract itself. Other limitations of the model are that it requires each participant to have sufficient knowledge about the programming language and that the process might consume a long time to reach a decision. Furthermore, boundary cases might arise in situations that the value at risk of the participants can change after the deployment of the smart contract. This could eventually lead to a centralised distribution of the voting power.

Requirement 7: ***The smart contract must have a static address on which it can be reached.***
The address of the proxy smart contract does not change after an update; hence the solution design satisfies this requirement (section 8.2.1).

Requirement 8: ***Participants must be able to invoke methods of the smart contract in one call.***
The implementation of a proxy smart contract allows a participant to interact with the smart contract in a single step (section 8.2.1). A participant is only required to send a transaction to the proxy smart contract, no other steps are required. As a result, this requirement is satisfied by the solution design.

Requirement 9: ***The solution design must be implemented in a smart contract.***
The voting mechanism is enforced by a smart contract (section 8.2.1). The smart contract is used to propose an update, to allow participants to vote, and it computes the result of the voting rounds. Furthermore, the smart contract automatically ensures that the update is implemented after a consensus has been reached. As a result, a participant is ensured that the update is implemented.

8.6 Validation

The solution design was validated by means of interviews with blockchain experts from different business sectors. The interviewees include blockchain experts from a Dutch bank, a software company, a start-up and a research institute. Due to different backgrounds, expertise and knowledge of blockchain, each interview had to adopt a different angle. First all interviewees were provided with a brief description of the study to introduce on what grounds it was performed. The first three of the four interviewees were asked to express their view on both the technical aspect and the generalised decision-making process. Given the limited time that the interviewees had available, the interviews did not address the individual case studies explicitly. The fourth interview had a much broader and abstract focus on the governance of smart contracts in general, as such, some individual aspects were discussed during the interview but not all of them.

8.6.1 Validation Results

A general perspective that was shared by the interviewees is that updatable smart contracts benefit safer adoption of smart contracts. One interviewee mentioned that smart contracts are closely tied to business processes and that these business processes tend to change over time. Another interviewee mentioned that updatable smart contracts might provide additional level of assurance that smart contracts are safe in use.

The solution design is a hybrid approach towards updatable smart contracts. As previously identified in the verification (section 8.5), the solution design only allows to update the controller smart contract; the functionality in the proxy smart contract is static and cannot be changed. To that extent, the solution design cannot be considered as a full updatable smart contract as stated by an interviewee.

While the technical design, and more specifically the assembly code for the custom `delegatecall()`, requires a deep understanding of Solidity, this aspect of the solution design could not be validated in-depth. Two interviewees acknowledged the limitations of the default implementation for `delegatecall()`. They agreed that a custom implementation, one that overcomes the limitations, should work in the manner that it was explained during the interviews. One interviewee highlighted the issue of data storage directly after the custom `delegatecall()` was explained. This person confirmed that the data storage approach as presented in this thesis is indeed a feasible option.

After the validation of the technical design, the interview proceeded with the validation of the generalised decision-making process. Initially, this part of the interview led to a few intensive and sometimes ideological discussions about blockchain, which led to the conclusion that there is not yet a consensus on what the key characteristics of blockchains are or should be. Once explained that the scope of this study was on public blockchains in which participants could not be trusted, such as Ethereum, the decision-making process was discussed.

The interviewees agreed that a voting-based system that adds a weight component to a vote, is a feasible solution to facilitate a decision-making process. It is the most logical and straightforward implementation. The value at risk principle is considered as a usable and implementable approach for the weight of a vote. The limitations including the boundary cases that were identified in this study were all acknowledged, however, this study failed to cover all of them. One interviewee pointed out that the two-round voting system initially might perform well, but that

it does not account for situations in which a new major participant occurs, or that a major participant no longer has a majority. This is a valid remark about the model that should be considered during the implementation. The underlying issue to solve here is that a participant potentially has multiple Ethereum accounts that together result in a majority. Unfortunately, there is no method to identify which Ethereum account belongs to which participant. This prevents the smart contract from determining whether a new participant has a majority, or when a major participant no longer has a majority. A second limitation of a two-round voting model, as mentioned by another interviewee, is that a bug in favour of the major participant cannot be solved without the consensus of a major participant. He argued that the model thus introduces a trusted party.

According to another interviewee, the decision-making process can be harmful in situations that two or more groups of participants use different parts of the smart contract. Consider a situation in which a group of participants with 80% of the voting power only need methods *A()* and *B()*, and a group of participants with 20% of the voting power that use *C()* and *D()*. Participants of the first group could propose and enforce an update that removes methods *C()* and *D()*. A potential solution to this problem might be to divide the functionality over two different smart contracts such that the groups do not use the same smart contract.

When it was asked whether the solution design could be applied in different blockchains, the answers were not all in line. One interviewee argued that especially the technical design is specifically targeted at Ethereum and that better solutions are probably available for other types of blockchain. As he stated, Ethereum is build in a world where each participant can trust no one and as a result of this, each Ethereum smart contract is processed within his own environment. Other types of blockchains, for example one that is specifically aimed at consortia, are developed in which multiple smart contracts are executed within the same environment. As such, these blockchains could allow to take a different approach at updating. Another interviewee indicated that the likelihood of applying the technical design on another blockchain should not be considered as possible. However, the decision-making process is more abstract and is therefore more likely to be applicable when using other blockchains. Only the method to determine the value at risk could require adjustment.

Chapter 9

Conclusion and Future Work

9.1 Conclusion

This study adopted the Design Science in Information Systems Research methodology on a study towards updatable smart contracts in order to give an answer to the following research question:

Research Question: *How could smart contracts be updated in a decentralised manner?*

Six subquestions were defined that together serve as an answer to the main research question. The answers to these subquestions are presented below.

Subquestion 1: *What is the current state of the art of blockchain technology?*

A literature study was conducted and presented in chapter 2 in order to answer this subquestion. Although Bitcoin marked the start of digital currencies, the underlying technology is not entirely new. A blockchain is a transaction-based system that collects transactions into blocks and stores it on a distributed network of computers. Multiple types of blockchains exist, these can be divided into three main types: public, consortium and private blockchains (section 2.1.3). Public blockchains are, in contrast to private blockchains, publicly accessible and writable. Consortium blockchains combine characteristics of public and private blockchains. The integrity and validity of the blocks in the chain are ensured by the computers in the network, called miners or nodes. Using a consensus algorithm, the miners in the network decide which miner is allowed to add the next block to the chain (section 2.2). In this thesis, the algorithms Proof of Work, Proof of Stake and Delegated Proof of Stake are covered. By using algorithms and cryptography these types of systems ensure the decentralisation, persistence, anonymity, transparency of the recorded transactions. While blockchains generally consist of a number of miners, there is no central entity that has the authority to validate all transactions. As a result of this, multiple attacks, such as race attacks, are possible on a blockchain (section 2.3).

Subquestion 2: *What is the current state of the art of smart contracts?*

As described in chapter 3, smart contracts are programs that are stored and executed on a blockchain. Once stored on the blockchain, each smart contract has its own address on which it can be reached. The code on this address is immutable, this ensures that the smart contract will, given a certain state, always behave in a consistent manner. For this study, the scope was narrowed down to only Ethereum smart contracts. Ethereum smart contracts are written in Solidity after which they are compiled to EVM code in order to deploy them on the blockchain (section 3.2). Transactions are not only used to exchange value but also to invoke methods of a smart contract. To prevent abuse of the network, each method invocation and each

transaction requires a certain fee which is based on the number of computations and the amount of data. Use cases of smart contracts can be found in escrow services, finance, the energy sector and IoT (section 3.3).

Subquestion 3: *What are existing security vulnerabilities of smart contracts?*

Both academic literature and non-academic literature describe multiple vulnerabilities of smart contracts (chapter 4). A recent study was focussed on the categorisation and severity of smart contract vulnerabilities. The results of this study are shown in Table 4.2. It lists 22 vulnerabilities, of which 14 are marked as Solidity vulnerabilities, two as EVM, and six as Blockchain. Vulnerabilities range from minor errors made by programmers to more abstract vulnerabilities such as unpredictable states. These vulnerabilities are to be found in existing smart contracts on the Ethereum blockchain. Two studies, for which custom tools were developed, discovered that a high number of existing smart contracts contain vulnerabilities. One study showed that 8,833 out of the 19,366 analysed smart contracts contain at least one vulnerability (section 4.2.1). The second study concluded that 4,905 Ether, worth $2.6 million, is stored on vulnerable smart contracts (section 4.2.2).

Subquestion 4: *What are design requirements for smart contracts that can be updated in a decentralised manner?*

Based on two briefly described existing solution designs and on the current state of the art of blockchain technology, smart contracts, and its vulnerabilities, nine requirements were identified for a solution design (chapter 6). These requirements were divided into functional requirements and non-functional requirements, as presented in section 6.1 and section 6.2. Additionally, a rationale was provided for each requirement. A major decision with considerable impact on a solution design is the definition of "a fair stake". In this study, fairness is defined as the value that a participant has stored on the smart contract, i.e. value at risk. The second requirement that had a high impact is the requirement on anonymous use.

Subquestion 5: *What is a design for smart contracts that can be updated in a decentralised manner?*

A design that allows to update smart contracts in a decentralised manner is composed of a decision-making process and a technical ability to implement it (chapter 8). The decision-making process allows to reach a consensus on whether an update will be implemented or not. This study explored a voting system that sets the weight of a vote equal to the value at risk of a participant. The process was extracted from four illustrative case studies based on existing smart contract implementations (section 7). In three of the four case studies, it allowed the participants to have a fair stake in terms of value at risk. To provide the ability to implement an update, the solution design adopted a proxy smart contract that acted as a gateway to the smart contract with the logic (section 8.2.1). Additionally, this proxy smart contract facilitates the decision-making process. As a consequence of the taken approach, a custom method is needed to store data in a secure way (section 8.2.1). The method that was adopted stores the data in predefined arrays per data type. An analysis of the costs of the proxy smart contract indicated that the transaction fee increases with at least 3.12% for the tested functions, for the data storage the cost increment is at least 3.83% (section 8.3). The solution design has a number of limitations that affect the effectiveness (section 8.4).

The limitations of the decision-making process are:

1. Participants might lack knowledge of the programming language to make a well-informed decision.
2. It is not applicable when the value at risk cannot be determined.
3. Centralisation of voting power cannot be prevented in all situations.
4. The process is time consuming.

The limitations of the technical design are:

1. The big bang adoption of an update might result in invalid transactions.
2. One block downtime.
3. The source code of an update must be publicly visible.
4. A hard fork might result in conflicting instances.

Subquestion 6: *Does the designed artefact work as desired?*

It can be concluded that the designed artefact does not work as desired. The verification on the satisfaction of the requirements showed that the designed artefact does not satisfy all requirements of a solution design (section 8.5). First, the presented design does not allow to update the code of the proxy smart contract, therefore not all components can be updated. Second, the design does not facilitate the necessary measures to patch critical vulnerabilities. Last, the artefact only provides participants with a fair stake in a few situations, but not in all situations.

The results of this study were validated by means of interviews with industry experts (section 8.6). The experts confirmed the need for updatable smart contracts and agreed that the solution design, in theory, should work in the problem context. However, they also confirmed that it is impractical due to the limitations. The validation did not consider the technical design to the full extent, the interviewees had little knowledge about the concepts of the Assembly implementation of the proxy. The general concept of a proxy smart contract, including the separation of data and logic into two smart contracts, was perceived as a feasible solution.

The interviewees confirmed that the voting system seems to be a feasible method for the decision-making process. However, additional limitations were identified by experts next to the ones initially identified in this study. The first key finding is that the decision-making process in combination with smart contracts does not allow to dynamically identify a participant with a majority. The second key finding is that two groups of participants theoretically could use different functionality provided by a smart contract. In those occasions, the decision-making process can have negative implications for the smaller group of participants.

9.2 Contribution

Prior to this study, only information on the vulnerabilities of smart contracts could be found in academic literature. No article provided a holistic overview of the concept of smart contracts. In this thesis, we presented a synthesis on the current state of the art of blockchain technology and smart contracts aiming to close this gap. We did not prove or suggest that these smart contracts are insecure by definition, however,

considering the list of vulnerabilities and the nature of blockchain, it is clear that safe use of these programs cannot be guaranteed. This thesis presented an overview of vulnerabilities that might occur within smart contracts and the security measures to minimise the risks. Albeit the fact that these programs potentially store high-value assets, they lack sufficient methods to guarantee safe usage. Any vulnerability that is found in a smart contract is permanent and irreversible.

The main goal of this study was to investigate whether a solution could be designed that would allow to update smart contracts in a decentralised manner, in order to facilitate safe usage of smart contracts. A major gap between existing solution designs and the character of blockchain technology is the level of decentralisation. Existing solution designs set one entity in control of the updates, this interferes with the decentralised nature of blockchain. We have tried to narrow this gap by studying a decision-making process that would allow each individual participant of a smart contract to have a fair stake. Although the solution design did not satisfy all the requirements, it still provides valuable insights to narrow down the gap. The study itself could be considered as a feasibility study and the insights can be used in future research to develop an improved design.

9.3 Discussion

This thesis showed that a research topic within the field of blockchain technology and smart contracts is challenging. During the interviews for the validation of the artefact, we found that the field is moving fast but without a clear direction. As expected, each expert had a different view of the characteristics of blockchains. What we did not expect was that they all favoured a different type of blockchain that we discussed in section 2.1.3. The influence of the type of blockchain on a design for a decision-making process seems to be significant. This became clear during an interview with a start-up company that develops blockchains and smart contracts in order to automate cross-organisational processes. They argued that these organisations already had a certain level of trust between each other given the fact that they do business together. This would allow them to set a single entity in control of the updates; the other entities could simply check if that single entity is compliant to their rules. Unfortunately, this approach could not be applied in this study while it does not allow anonymous use and does not provide participants a fair stake.

Blockchain technology and smart contracts evolve at a high pace. A consequence of this high pace is that the information upon which this study is based could be outdated to some extent. If one would start this study with a similar approach again at the end of this study, October 2018, the solution design and intermediate results might differ. For example, the list of smart contract vulnerabilities could have become longer during the period of this study. Probably the most significant change would be found in the technical design that was presented in this thesis. At the start of this study, the combination of a proxy smart contract with a controller smart contract was considered as the best solution. Recent developments focussed on the technical aspects of updatable smart contracts, provide much more information on this approach (Spagnuolo and ZeppelinOS, 2018; Trail of Bits, 2018). The information could lead to different decisions during the design phase of the study. Despite the high pace in developments, we have no indication that other studies have tried to solve the challenge of a complete approach towards updatable smart contracts. We are the first ones that have considered a decision-making process, which is to our belief remarkable. This fact is hard to explain, especially given the area in which

decentralisation and transparency are important. A reason for this could be that the maturity of smart contracts has not yet reached a level to start thinking about these kinds of problems; the technical implementation is already challenging.

Although this study is based on existing implementations of smart contracts, it would be interesting to perform research on the adoption of smart contracts within businesses. We feel that there is a gap between implementations within businesses and implementations described academic and non-academic literature. As such, if we were to perform research again, we would have taken a slightly different approach than we took in this study. First of all, the study should be split into two separate studies, one study on the decision-making process and one on the technical implementation. A reason for this is that the scope of the decision-making process should be more general than the scope for the technical implementation. Second, we would have performed a more extensive analysis of the areas in which smart contracts are adopted. The scope of this study was specifically on Ethereum smart contracts: a system with a public and anonymous character. However, we have seen that not all cases specifically rely on the same characteristics of the Ethereum blockchain. Thus, we would start to get an overview of existing implementations and to classify these implementations. As the decision-making process depends on the type of implementation, one could take a more specific scope by using the classification of implementations.

In recent years, blockchain gained tremendous attention. Some people consider it to be a hype while others say that it will become a part of our digital infrastructure. The impact that blockchain will have is neither black nor white, it is much more nuanced (Iansiti and Lakhani, 2017). The term blockchain surely contains a certain amount of hype, companies seem to be doing "blockchain projects" just to show the outside world that they are an innovative company, meanwhile, they do not seem to understand the impact, benefits, disadvantages and risks. In many of these projects, the issue could be solved with many other technologies and not merely with blockchain. While considering the characteristics of (public) blockchains, it is clear that the real value of blockchain is in cross-organisational and collaborative community projects. The technology allows to use a central database with a high degree of availability and integrity, while not setting a single entity in full control of the data.

9.4 Limitations

This study was designed to explore a design for smart contracts that can be updated in a decentralised manner. The study has the following limitations.

First of all, the decision-making process that was the result of this study is highly theoretical, it is only based on four illustrative case studies. It could be argued that the limited number of case studies results in a less accurate and less generalisable process. More case studies could have been done to increase the validity of the result. They could, for example, reveal additional limitations or other boundary cases that the previous case studies did not have.

Second, this study did not go through the full design science cycle as discussed in section 1.4. Specifically, the treatment implementation and implementation evaluation were not part of this research due to time limitations. The validation of the solution design revealed new boundary cases and limitations that were not yet identified. We believe that testing both the decision-making process and technical design

in practice could reveal more boundary cases and limitations. The next cycle in design science methodology could consider all findings and then ultimately result in improved and more secure models.

Third, this study is only concerned with smart contracts on the Ethereum blockchain. Hence, the results of this study, specifically the technical design, have a limited applicability. Although Ethereum is currently the most used blockchain for smart contracts globally, it is possible that over time new and improved blockchain platforms for smart contracts arise. Looking at the pace on which the developments in the area of blockchain and smart contracts go, it is likely that this will happen.

Last, given the rapid development in the area of blockchain technology and smart contracts, this study had to consult a relatively large number of non-academic sources. This holds for example for the study on the current state of the art of smart contracts, which mainly relied on the documentation of Ethereum. A downside of the use of non-academic sources is that the foundation and evidence of the information are not always provided or weak. An example of such source is Etherscan (2018d), which was consulted to retrieve the average gas price. The source fails to describe the exact method that was used to determine the average gas price.

9.5 Future Research

Based on this study, we identified a number of potential areas for future research. These areas do not only emerge from the solution design, but also from more general high-level aspects of blockchain and smart contracts.

The first topic for future research would be to design an artefact similar to this study, but without the requirements of anonymous use and fair stake. These two requirements restricted the design of the solution significantly and resulted in a number of limitations. Ignoring these requirements would allow to investigate whether delegate voting is a feasible concept to streamline the decision-making process. Delegate voting would create the opportunity for people without knowledge of smart contracts to trust their vote to someone else. Additional research is needed to get an improved understanding of the applicability and feasibility of this concept in other cases.

Another topic for further research is to get a clear overview of the requirements for a solution design. The requirements that were identified in this study originated from the general characteristics of blockchain and smart contracts found in the preceding literature studies. A different approach to determine the requirements would be to gather them from existing use cases and implementations. This could increase the validity of the requirements and the applicability of solution design. Before this approach can be adopted, it is important to gather sufficient information on the use cases and implementations. As shortly described in the discussion (section 9.3), there seems to be a gap between use cases of smart contracts in literature and actual implementations. Researching this gap would result in a better understanding of the advantages, disadvantages and risks.

The third topic for future research originates from the requirement on the need to fix critical vulnerabilities within a short period. We did not find a solution to satisfy this requirement without violating the other requirements. This topic could be researched more in-depth to determine whether there are viable solutions that we did not identify.

Bibliography

Abdellatif, T. and Brousmiche, K.-L. Formal Verification of Smart Contracts Based on Users and Blockchain Behaviors Models. *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–5, 2018.

Abrams, L. Evrial Trojan Switches Bitcoin Addresses Copied to Windows Clipboard, 2018, URL: https://www.bleepingcomputer.com/news/security/evrial-trojan-switches-bitcoin-addresses-copied-to-windows-clipboard/ *Accessed on:* 2018-03-19.

Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., De Caro, A., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y., Muralidharan, S., Murthy, C., Nguyen, B., Sethi, M., Singh, G., Smith, K., Sorniotti, A., Stathakopoulou, C., Vukolić, M., Cocco, S. W., and Yellick, J. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 30:1—–30:15, New York, NY, USA, 2018. ACM.

Atzei, N., Bartoletti, M., and Cimoli, T. A Survey of Attacks on Ethereum Smart Contracts (SoK). In Maffei, M. and Ryan, M., editors, *Principles of Security and Trust*, pages 164–186, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

Axiom Zen. Smart Contract CryptoKitties, 2017, URL: https://etherscan.io/address/0x06012c8cf97bead5deae237070f9587f8e7a266d#code *Accessed on:* 2018-07-24.

Bag, S., Ruj, S., and Sakurai, K. Bitcoin Block Withholding Attack: Analysis and Mitigation. *IEEE Transactions on Information Forensics and Security*, 12(8):1967–1978, 2017.

Bahga, A. and Madisetti, V. K. Blockchain Platform for Industrial Internet of Things. *Journal of Software Engineering and Applications*, 09(10):533–546, 2016.

Banafa, A. IoT and Blockchain Convergence: Benefits and Challenges, 2017, URL: https://iot.ieee.org/newsletter/january-2017/iot-and-blockchain-convergence-benefits-and-challenges.html *Accessed on:* 2018-06-26.

Barcelo, J. . User Privacy in the Public Bitcoin Blockchain. *(Working paper)*. 2014.

BBC News. CryptoKitties craze slows down transactions on Ethereum, 2017, URL: https://www.bbc.com/news/technology-42237162 *Accessed on:* 2018-07-24.

Bentov, I., Lee, C., Mizrahi, A., and Rosenfeld, M. Proof of Activity: Extending Bitcoin's Proof of Work via Proof of Stake. *Cryptology ePrint Archive*, 452(3):1–19, 2014.

Biryukov, A., Khovratovich, D., and Pustogarov, I. Deanonymisation of clients in Bitcoin P2P network. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 15–29, 2014.

Bitcoin Project. Contract, 2018a, URL: https://bitcoin.org/en/developer-guide#contracts *Accessed on:* 2018-04-13.

Bitcoin Project. Bitcoin Improvement Proposal, 2018b, URL: https://github.com/bitcoin/bips/blob/master/README.mediawiki *Accessed on:* 2018-06-01.

Bitcoin Project. Block Chain Overview, 2018c, URL: https://bitcoin.org/en/developer-guide#block-chain-overview *Accessed on:* 2017-04-16.

Bitcoin Project. Proof of Work, 2018d, URL: https://bitcoin.org/en/developer-guide#proof-of-work *Accessed on:* 2018-04-16.

Bitcointalk Forum. Re: Fake Bitcoins?, 2011, URL: https://bitcointalk.org/index.php?topic=36788.msg463391#msg463391 *Accessed on:* 2018-02-15.

Buterin, V. Selfish Mining: A 25% Attack Against the Bitcoin Network, 2013, URL: https://bitcoinmagazine.com/articles/selfish-mining-a-25-attack-against-the-bitcoin-network-1383578440/ *Accessed on:* 2018-02-16.

Buterin, V. The Meaning of Decentralization, 2017, URL: https://medium.com/@VitalikButerin/the-meaning-of-decentralization-a0c92b76a274 *Accessed on:* 2018-07-06.

Buterin, V. Proof of Stake FAQ, 2018, URL: https://github.com/ethereum/wiki/wiki/Proof-of-Stake-FAQ *Accessed on:* 2018-04-16.

Castaldo, L. and Cinque, V. Blockchain-Based Logging for the Cross-Border Exchange of eHealth Data in Europe. In Gelenbe, E., Campegiani, P., Czachórski, T., Katsikas, S. K., Komnios, I., Romano, L., and Tzovaras, D., editors, *Security in Computer and Information Sciences*, pages 46–56, Cham, 2018. Springer International Publishing.

Catalini, C. and Gans, J. . Initial Coin Offerings and the Value of Crypto Tokens. *NBER No. 24418 (Working paper)*. 2018.

Christidis, K. and Devetsikiotis, M. Blockchains and Smart Contracts for the Internet of Things. *IEEE Access*, 4:2292–2303, 2016.

Clack, C. D., Bakshi, V. A., and Braine, L. Smart Contract Templates: foundations, design landscape and research directions. *Computing Research Repository*, abs/1608.0: 1–15, 2016.

CoinMarketCap. Cryptocurrencies Market Capitalization, 2018, URL: https://coinmarketcap.com/ *Accessed on:* 2018-05-03.

ConsenSys Diligence. Mythril, 2018a, URL: https://github.com/ConsenSys/mythril#searching-from-the-command-line *Accessed on:* 2018-04-25.

ConsenSys Diligence. Known Attacks, 2018b, URL: https://consensys.github.io/smart-contract-best-practices/known_attacks/ *Accessed on:* 2018-05-22.

ConsenSys Diligence. Recommendations for Smart Contract Security in Solidity, 2018c, URL: https://consensys.github.io/smart-contract-best-practices/recommendations/ *Accessed on:* 2018-05-25.

ConsenSys Diligence. Software Engineering Techniques, 2018d, URL: https://consensys.github.io/smart-contract-best-practices/software_engineering/ *Accessed on:* 2018-05-31.

Conti, M., Kumar, S., Lal, C., and Ruj, S. A Survey on Security and Privacy Issues of Bitcoin. *IEEE Communications Surveys Tutorials*, 2017.

Coplien, J. O. Software design patterns: Common questions and answers. *The Patterns Handbook: Techniques, Strategies, and Applications*, pages 311–320, 1998.

CryptoKitties. Collect and breed digital cats!, 2018, URL: https://www.cryptokitties.co/ *Accessed on:* 2018-07-24.

Destefanis, G., Marchesi, M., Ortu, M., Tonelli, R., Bracciali, A., and Hierons, R. Smart contracts vulnerabilities: a call for blockchain software engineering? *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 19–25, 2018.

Dika, A. *Ethereum Smart Contracts : Security Vulnerabilities and Security Tools*. Master thesis, Norwegian University of Science and Technology, 2017.

Douceur, J. R. The Sybil Attack. In Druschel, P., Kaashoek, F., and Rowstron, A., editors, *Peer-to-Peer Systems*, pages 251–260, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

Elena, N. and Spagnuolo, F. Proxy Patterns, 2018, URL: https://blog.zeppelinos.org/proxy-patterns/ *Accessed on:* 2018-07-19.

Entriken, W., Shirley, D., Evans, J., Sachs, N., and Ethereum Foundation. EIPS eip-721, 2018, URL: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-721.md *Accessed on:* 2018-07-24.

Ethereum Foundation. Opcodes, 2017, URL: https://github.com/ethereum/pyethereum/blob/develop/ethereum/opcodes.py *Accessed on:* 2018-04-18.

Ethereum Foundation. Solidity Documentation, 2018a, URL: https://solidity.readthedocs.io/en/v0.4.22/ *Accessed on:* 2018-04-17.

Ethereum Foundation. Ethereum Contract ABI, 2018b, URL: https://github.com/ethereum/wiki/wiki/Ethereum-Contract-ABI *Accessed on:* 2018-04-18.

Ethereum Foundation. Yellowpaper Repository, 2018c, URL: https://github.com/ethereum/yellowpaper/commits/master *Accessed on:* 2018-04-18.

Ethereum Foundation. Common Patterns, 2018d, URL: http://solidity.readthedocs.io/en/v0.4.21/common-patterns.html *Accessed on:* 2018-04-25.

Ethereum Foundation. Ethereum Contract ABI, 2018e, URL: https://github.com/ethereum/wiki/wiki/Ethereum-Contract-ABI *Accessed on:* 2018-04-20.

Ethereum Foundation. Security Considerations, 2018f, URL: http://solidity.readthedocs.io/en/latest/security-considerations.html *Accessed on:* 2018-05-02.

Ethereum Foundation. Safety, 2018g, URL: https://github.com/ethereum/wiki/wiki/Safety *Accessed on:* 2018-05-22.

Ethereum Foundation. Solidity by Example, 2018h, URL: http://solidity.readthedocs.io/en/v0.4.21/solidity-by-example.html *Accessed on:* 2018-07-23.

Ethereum Foundation. Solidity Assembly, 2018i, URL: http://solidity.readthedocs.io/en/v0.4.21/assembly.html *Accessed on:* 2018-08-06.

Ethereum Foundation. White Paper, 2018j, URL: https://github.com/ethereum/wiki/wiki/White-Paper *Accessed on:* 2018-04-13.

Ethereum Foundation. Introduction to Smart Contracts, 2018k, URL: http://solidity.readthedocs.io/en/v0.4.21/introduction-to-smart-contracts.html *Accessed on:* 2018-04-20.

Etherscan. Ethereum Transaction (Parity InitWallet), 2017a, URL: https://etherscan.io/tx/0x05f71e1b2cb4f03e547739db15d080fd30c989eda04d37ce6264c5686e0722c9 *Accessed on:* 2018-04-16.

Etherscan. Ethereum Transaction (Parity Selfdestruct), 2017b, URL: https://etherscan.io/tx/0x47f7cff7a5e671884629c93b368cb18f58a993f4b19c2a53a8662e3f1482f690 *Accessed on:* 2018-04-16.

Etherscan. Contracts With Verified Source Codes Only, 2018a, URL: https://etherscan.io/contractsVerified *Accessed on:* 2018-04-17.

Etherscan. Smart Contract VeChain Token, 2018b, URL: https://etherscan.io/address/0xd850942ef8811f2a866692a623011bde52a462c1#code *Accessed on:* 2018-07-18.

Etherscan. Ethereum (Ether) Historical Prices, 2018c, URL: https://etherscan.io/chart/etherprice *Accessed on:* 2018-08-16.

Etherscan. Ethereum Gas Price Tracker, 2018d, URL: https://etherscan.io/gastracker *Accessed on:* 2018-08-16.

Eyal, I. and Sirer, E. G. Majority Is Not Enough: Bitcoin Mining Is Vulnerable Ittay. *International conference on financial cryptography and data security*, pages 436–454, 2014.

Fenu, G., Marchesi, L., Marchesi, M., and Tonelli, R. The ICO phenomenon and its relationships with ethereum smart contract environment. *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 26–32, 2018.

Finney, H. Best practice for fast transaction acceptance - how high is the risk?, 2011, URL: https://bitcointalk.org/index.php?topic=3441.msg48384#msg48384 *Accessed on:* 2018-02-13.

Frantz, C. K. and Nowostawski, M. From institutions to code: Towards automated generation of smart contracts. *Proceedings - IEEE 1st International Workshops on Foundations and Applications of Self-Systems, FAS-W 2016*, pages 210–215, 2016.

Garay, J., Kiayias, A., and Leanoardos, N. The Bitcoin Backbone Protocol: Analysis and Applications. *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 9057:281–310, 2015.

Gasser, U., Budish, R., and West, S. M. Multistakeholder as Governance Groups: Observations from Case Studies. *SSRN Electronic Journal*, 7641, 2015.

Gervais, A., Karame, G. O., Capkun, V., and Capkun, S. Is Bitcoin a Decentralized Currency? *IEEE Security and Privacy*, 12(3):54–60, 2014.

Hahn, A., Singh, R., Liu, C. C., and Chen, S. Smart contract-based campus demonstration of decentralized transactive energy auctions. *2017 IEEE Power and Energy Society Innovative Smart Grid Technologies Conference, ISGT 2017*, 2017.

Hevner, A. R., March, S. T., Park, J., and Ram, S. Design Science in Information Systems Research. *Design Science in IS Research MIS Quarterly*, 28(1):75–105, 2004.

Iansiti, M. and Lakhani, K. R. The Truth About Blockchain. *Harvard business review*, (January-February), 2017.

Idelberger, F., Governatori, G., Riveret, R., and Sartor, G. Evaluation of Logic - Based Smart Contracts for Blockchain Systems. 1(July):1–17, 2016.

Karame, G. O. and Androulaki, E. Double-Spending Fast Payments in Bitcoin. *Acm Ccs*, pages 906–917, 2012.

Kiayias, A. and Panagiotakos, G. Speed-Security Tradeoffs in Blockchain Protocols. *Cryptology ePrint Archive*, pages 1–19, 2015.

King, S. and Nadal, S. PPCoin: Peer-to-Peer Crypto-Currency with Proof-of-Stake. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16*, pages 1–27, 2012.

Knirsch, F., Unterweger, A., Eibl, G., and Engel, D. Privacy-Preserving Smart Grid Tariff Decisions with Blockchain-Based Smart Contracts. In Rivera, W., editor, *Sustainable Cloud and Energy Services: Principles and Practice*, pages 85–116. Springer International Publishing, Cham, 2018.

Kounelis, I., Steri, G., Giuliani, R., Geneiatakis, D., Neisse, R., and Nai-Fovino, I. Fostering consumers' energy market through smart contracts. *Energy and Sustainability in Small Developing Economies, ES2DE 2017 - Proceedings*, pages 0–5, 2017.

Kroll, J. a., Davey, I. C., and Felten, E. W. The Economics of Bitcoin Mining, or Bitcoin in the Presence of Adversaries. *The Twelfth Workshop on the Economics of Information Security (WEIS 2013)*, 2013(Weis):1–21, 2013.

Kshetri, N. Blockchain's roles in strengthening cybersecurity and protecting privacy. *Telecommunications Policy*, 41(10):1027–1038, 2017a.

Kshetri, N. Can Blockchain Strengthen the Internet of Things? *IT Professional*, 19(4), 2017b.

Kwon, Y., Kim, D., Son, Y., Vasserman, E., and Kim, Y. Be Selfish and Avoid Dilemmas: Fork After Withholding (FAW) Attacks on Bitcoin. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 195–209, 2017.

Lamers, D. *Possibilities for blockchain in the energy transition*. Master thesis, University of Twente, 2018.

Lamport, L., Shostak, R., and Pease, M. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

Lemble, A. and Ethereum Foundation. ERC930 - Eternal Storage Standard, 2018, URL: https://github.com/ethereum/EIPs/issues/930 *Accessed on:* 2018-07-19.

Lin, I.-C. and Liao, T.-C. A Survey of Blockchain Security Issues and Challenges. *International Journal of Network Security*, 1919(55):653–659, 2017.

Luu, L., Chu, D.-H., Olickel, H., Saxena, P., and Hobor, A. Making Smart Contracts Smarter. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16*, pages 254–269, 2016.

Matsuo, S. How formal analysis and verification add security to blockchain-based systems. In *Formal Methods in Computer Aided Design*, pages 1–4, 2017.

Mavridou, A. and Laszka, A. *Tool Demonstration: FSolidM for Designing Secure Ethereum Smart Contracts Anastasia*, volume 1. Springer International Publishing, 2018.

Nakamoto, S. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008.

Nazarkin, G. DealMate Smart Contract, 2018, URL: https://gist.github.com/gnazarkin/c3006c7cec3e2842c2d840f5408bd920 *Accessed on:* 2018-07-23.

Nikolic, I., Kolluri, A., Sergey, I., Saxena, P., and Hobor, A. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. *Computing Research Repository*, 2018.

Oraclize Limited. Oraclize - blockchain oracle service, enabling data-rich smart contracts, 2018a, URL: http://www.oraclize.it/ *Accessed on:* 2018-04-26.

Oraclize Limited. Oraclize Documentation, 2018b, URL: https://docs.oraclize.it/#home *Accessed on:* 2018-04-26.

Parity Technologies. Security Alert, 2017, URL: https://paritytech.io/security-alert-2/ *Accessed on:* 2018-04-16.

Parity Technologies. A Postmortem on the Parity Multi-Sig Library Self-Destruct, 2018, URL: http://paritytech.io/a-postmortem-on-the-parity-multi-sig-library-self-destruct/ *Accessed on:* 2018-04-16.

Peters, G. W. and Panayi, E. *Understanding Modern Banking Ledgers Through Blockchain Technologies: Future of Transaction Processing and Smart Contracts on the Internet of Money*. 2016.

Popper, N. A Hacking of More Than $50 Million Dashes Hopes in the World of Virtual Currency, 2016, URL: https://www.nytimes.com/2016/06/18/business/dealbook/hacker-may-have-removed-more-than-50-million-from-experimental-cybercurrency-project.html *Accessed on:* 2018-04-24.

Prnewswire. Propy Announces World's First Real Estate Purchase on Ethereum Blockchain, 2018, URL: https://www.prnewswire.com/news-releases/propy-announces-worlds-first-real-estate-purchase-on-ethereum-blockchain-300528640.html *Accessed on:* 2018-02-19.

Schreiber, F. and Alexandre, P. *Sybil*. Warner Books New York, 1974.

Spagnuolo, F. and ZeppelinOS. Smart Contract Upgradeability using Eternal Storage, 2018, URL: https://blog.zeppelinos.org/smart-contract-upgradeability-using-eternal-storage/ *Accessed on:* 2018-09-10.

Szabo, N. Smart Contracts: Building Blocks for Digital Markets, 1996, URL: http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html *Accessed on:* 2018-04-16.

Taul, S. No DAO funds at risk following the Ethereum smart contract 'recursive call' bug discovery, 2016, URL: https://blog.slock.it/no-dao-funds-at-risk-following-the-ethereum-smart-contract-recursive-call-bug-discovery-29f482d348b *Accessed on:* 2018-05-06.

Trail of Bits. Contract upgrade anti-patterns, 2018, URL: https://blog.trailofbits.com/2018/09/05/contract-upgrade-anti-patterns/ *Accessed on:* 2018-09-10.

Tranter, W. H., Taylor, D. P., Ziemer, R. E., Maxemchuk, N. F., and Mark, J. W. On Distributed Communications Networks. In *The Best of the Best: Fifty Years of Communications and Networking Research*, pages 692–. Wiley-IEEE Press, 2007.

Tschorsch, F. and Scheuermann, B. Bitcoin and Beyond : A Technical Survey on Decentralized Digital Currencies. *IEEE Communications Surveys & Tutorials*, 18(3): 2084–2123, 2016.

Ubitquity. The First Blockchain-Secured Platform for Real Estate Recordkeeping, 2018, URL: https://www.ubitquity.io/ *Accessed on:* 2018-07-24.

VeChain. About, 2018, URL: https://www.vechain.com/#/about *Accessed on:* 2018-02-14.

Vogelsteller, F., Buterin, V., and Ethereum Foundation. EIPS eip-20, 2018, URL: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md *Accessed on:* 2018-05-03.

Wieringa, R. *Design Science Methodology for Information Systems and Software Engineering*. Springer Berlin Heidelberg, 2014.

Wood, G. Ethereum: a secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, pages 1–32, 2014.

Wood, G. and Buchanan, A. Advancing Egalitarianism. In *Handbook of Digital Currency: Bitcoin, Innovation, Financial Instruments, and Big Data*, pages 385–402. Elsevier Inc., 2015.

Wood, G. and Ethereum Foundation. Ethereum: a secure decentralised generalised transaction ledger, 2018, URL: https://ethereum.github.io/yellowpaper/paper.pdf *Accessed on:* 2018-04-18.

Zheng, Z., Xie, S., Dai, H., Chen, X., and Wang, H. An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends. *Proceedings - 2017 IEEE 6th International Congress on Big Data, BigData Congress 2017*, pages 557–564, 2017.

Appendix A

Average price Gas and Ether

Figure A.1 shows the average gas price between the 16^{th} of August, 2017 and the 15^{th} of August, 2018. Figure A.2 provides an overview of the Ether price for the same period. The price of Ether is volatile as can be seen in the latter figure. The differences between the highest and lowest values are significant and occurred in a relatively short time. The cause of the volatility might be a combination of different factors, such as increased media attention and high number of ICOs.

FIGURE A.1: Average historical Gas price (Etherscan, 2018d)

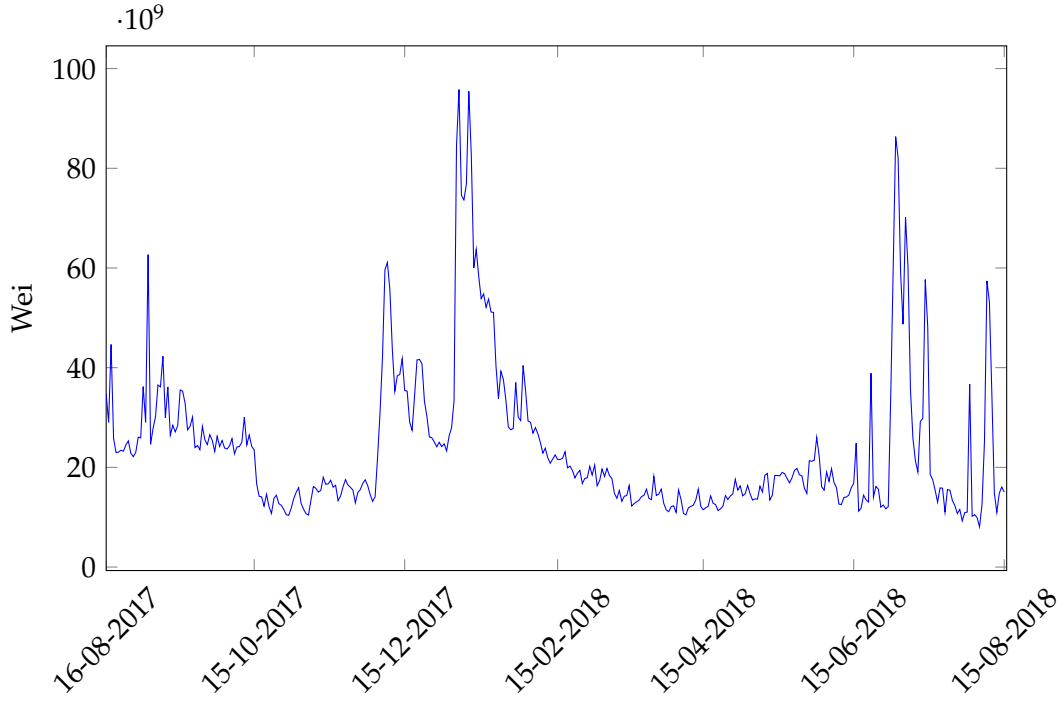

FIGURE A.2: Historical prices Ether (Etherscan, 2018c)