TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Alikhan Sailekeyev 194438IVSB

Secure Ethereum Virtual Machine Compatible Smart Contracts Development Guideline for a Private Company

Bachelor's thesis

Supervisor: Valdo Praust
Msc

Tallinn 2022

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Alikhan Sailekeyev 194438IVSB

Turvaline Ethereumi virtuaalmasinaga ühilduv nutikate lepingute arendusjuhend eraettevõttes kasutamiseks

Bakalaureusetöö

Juhendaja: Valdo Praust
Msc

Tallinn 2022

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Alikhan Sailekeyev 194438IVSB

14.05.2022

Abstract

Smart contracts are set of computer code instructions that are properly executed in a decentralized way by a network of independent distrusting nodes without the intervention of a third party.

With the active growth of extensively used modern public blockchain networks that support the execution of self-written code, more new individuals are getting involved into the smart contracts' development. Given the fact that the uploaded smart contract's logic source code cannot be overwritten or replaced, the quality and security of final executable code must be ensured. New start-up companies that want to introduce new decentralized applications must know the best practice to develop reliable and secure smart contracts. Therefore, a development guideline needs to be designed for the use of company employees that would consider best practices that will meet the requirements of security and business needs.

In this research paper the author's final guideline mainly focuses on Solidity programming language and Ethereum Virtual Machine compatible blockchain networks. Throughout the research, the author examines and compares the existing smart contracts development best practices solutions to determine the best approaches. The main goal of the author is to cover the whole development aspects such as smart contracts' design, security, testing and deploy that would meet the business requirements.

This thesis is written in English and is 64 pages long, including 8 chapters, 13 figures and 4 tables.

List of abbreviations and terms

DAPP	Decentralized application
DEFI	Decentralized finance
DAO	Decentralized Autonomous Organisation
EVM	Ethereum Virtual Machine
ETH	Ether (native token of Ethereum network)
EIP	Ethereum Improvement Proposal
FTP	File Transfer Protocol
GDPR	Global Data Protection Regulation
HTTP	Hyper
ICO	Initial Coin Offering
IP	Internet Protocol
POW	Proof-of-Work
POS	Proof-of-Stake
POH	Proof-of-History
SMTP	Simple Mail Transfer Protocol
TCP	Transmission Control Protocol
Wei	The smallest denomination of ether

Table of contents

1 Introduction ... 10
1.1 Company description ... 10
1.2 The main problem ... 11
1.3 Motivation and Objectives ... 11
2 Background on Ethereum Blockchain and Smart contracts ... 13
2.1 Blockchain ... 13
2.1.1 Definition and characteristics of Blockchain ... 13
2.1.2 Blockchain architecture ... 14
2.2 Consensus Mechanism ... 15
2.3 Ethereum ... 15
2.3.1 Addresses ... 16
2.3.2 Ethereum Virtual Machine (EVM) ... 16
2.3.3 Smart contracts ... 17
2.3.4 Execution Fees ... 18
2.4 Blockchain explorers ... 20
2.5 General Data Protection Regulation ... 22
2.5.1 Scope, penalties, and key definitions of GDPR ... 22
3 State of the art ... 24
3.1 Literature Research ... 24
4 Methodology ... 29
5 Smart contracts Vulnerabilities and Data protection analysis ... 30
5.1 Real-world cases ... 30
5.1.1 The DAO hack ... 30
5.1.2 Solana x Wormhole hack ... 31
5.2 Existing development guidelines ... 32
5.2.1 Cytric's implementation of Secure Smart Contracts ... 33
5.2.2 Consensys' implementation of Secure Smart Contracts ... 34
5.3 Smart contracts and security ... 36
5.3.1 Security Challenges in Ethereum ... 37

5.3.2 Design Challenges and Patterns Usage 38
5.3.3 Security analyzer for testing 38
5.4 User data protection under GDPR 39
5.4.1 Data transparency 40
5.4.2 Data Minimization, Accuracy and Storage Limitation 40
5.4.3 Purpose limitation 40
5.5 Hardware wallets 41
6 Proposed Solution 43
6.1 Design 43
6.1.1 Development tools 43
6.1.2 Standards 44
6.1.3 Control and Management 45
6.1.4 Upgradeability 45
6.1.5 Data regulation 46
6.2 Security 46
6.2.1 Reentrancy 46
6.2.2 Balance limit 47
6.2.3 Emergency stop 47
6.2.4 Time management 48
6.2.5 Integer Overflow 48
6.2.6 Phishing with origin of the sender 48
6.2.7 Signature Replay 49
6.3 Testing 49
6.3.1 Truffle suite and Ganache 49
6.3.2 Security testing 50
6.4 Deploy 50
6.4.1 Truffle deploy 51
6.4.2 Securing deployer account private key 52
7 Discussion 53
8 Summary 54
References 55
Appendix 1 – Source codes 58
Appendix 2 – Non-exclusive licence for reproduction and publication of a graduation thesis 64

List of figures

Figure 1: Centralized vs decentralized 13
Figure 2: Blocks sequence 14
Figure 3: Blockchain architecture 14
Figure 4: Smart contracts working 17
Figure 5: Pattern: Checks-effects-interaction 58
Figure 6: Pattern: Mutex 58
Figure 7: Pattern: Balance limit 59
Figure 8: Pattern: Emergency stop 60
Figure 9: Pattern: Rate limit pattern 60
Figure 10: Pattern: Speed bump 61
Figure 11: Pattern: Ownable 62
Figure 12: Pattern: Roles 62
Figure 13: Pattern: Authorization 63

List of tables

Table 1: Ether denominations. 19
Table 2: Gas costs in gwei. 19
Table 3: Securty patterns of solidity development. 25
Table 4: Functional design patterns of solidity development. 26

1 Introduction

Since Bitcoin's debut in 2009 and its subsequent success in the cyber and financial field, the term of blockchain has piqued huge attention worldwide. The "disruptive" potential of this innovative system, the underlying technology of cryptocurrencies, has captivated industries as well as national governments [1][2]

The current level of worldwide digitalization and high network availability led to the existence of more global blockchain networks. The widespread and performance of computing devices are on a such level, that nowadays most users are capable of running the complex system of blockchain network on their own devices and provide contribution and support to the progressing software field of decentralized applications. Since the introduction of Ethereum blockchain network with the support of EVM, lots of individuals started inventing new applications and protocols using smart contracts as instructions of code that are executed among various machines.

However, decentralized applications development in context of blockchain networks is a relatively new concept, and many new ideas are being proposed even nowadays. Those promising projects evolve into start-up projects. And mostly, start-up companies hire young talents to implement their design into reality. The final code being executed in a decentralized way means that it must be secure and bug-free. Therefore, to achieve these basic and crucial needs, a well-structured secure smart contracts development guideline needs to be developed to help company employees and interns that additionally would meet the requirements of security measures and business needs.

1.1 Company description

The case study Company is situated in Kazakhstan and specializes in IT software development. This company offers various IT development services starting from e-commerce websites and ending with decentralized applications. The blockchain based applications development department has been founded in the end of 2020

Today, this department is developing a software for internal and outsource projects. The development team consists of people who are responsible for building front-end, back-end, and smart contracts solutions. Each member, in turn, is in charge of one or several projects at the same time.

1.2 The main problem

The author would like to admit that the blockchain developers are the main workers who have to manage the tasks of several different projects at the same time, while the other developers of other fields stay focused on their own specific solutions. This situation clearly shows that the company was in dire need of more talented and experienced specialists in blockchain technology field. Throughout the author's working period, he has seen the several waves of new interns who were hired to fill the gap of active developers. Hiring of new employees might be enough to resolve the problem of development speed, but not the development quality. One of the main problems of new interns were their lack of understanding of how smart contracts work and experience in development of optimized code. The security issues of blockchain networks and smart contracts vulnerabilities are a bit different than compared to security of other enterprise level applications.

1.3 Motivation and Objectives

By observing the situation regarding new interns' qualification and learning process, the author has come to the point, that a host company needs a development guideline for the developers of decentralized applications. This guideline would describe all the critical parts of designing, development, testing, and deployment that would meet all the business and security requirements.

Throughout the author's work experience at the blockchain based start-up company, the following points have been identified as necessary when developing DAPPs for enterprise needs:

- The main idea logic of decentralized application must be implemented directly in smart contracts

- The code and logic must be optimized as much as possible to reduce transactions cost

- The smart contract should not be constantly maintained in order to perform correctly. However, it depends on the core application idea

- The testing must be performed properly to avoid any possible bugs

- In case of big code and complex logic, the use feature of upgradeability must be considered

- The application must comply with user’s data protection regulations

The three objectives of this project are:

1. To observe the best existing EVM compatible smart contracts development practices.

2. To define what are the most important development aspects that would help avoid possible cyber threats.

3. To develop a guideline that will cover the design, security, testing and deploy stages of smart-contracts development for the use of company employees

2 Background on Ethereum Blockchain and Smart contracts

This chapter covers a number of key concepts that are important to comprehending this thesis, as well as some of the key technology discussed, such as blockchain and smart contracts.

2.1 Blockchain

2.1.1 Definition and characteristics of Blockchain

Blockchain is a peer-to-peer, ever-growing, cryptographically secure, hardly immutable, shared recordkeeping system in which each user, or so-called node, keep a copy of transaction records, that can only be updated via consensus when all parties involved in a network agree to update [1]. It is a technology that meets requirements such as decentralization, traceability, integrity, confidentiality, auditability, transparency, security [3][4]. When management is centralized on a single entity, the integrity of an information system is directly dependent on that entity, but in a decentralized system, the integrity of information is dependent on all members in the network where the system functions. As a result, one entity's unilateral will is insufficient to change a record. Figure 1 illustrates the difference between centralized and decentralized authorities.

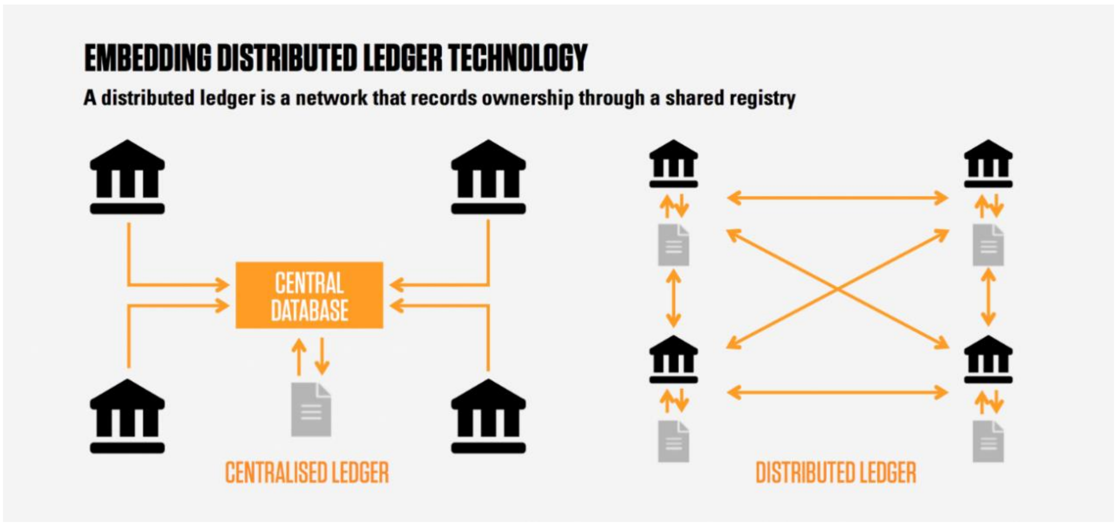
EMBEDDING DISTRIBUTED LEDGER TECHNOLOGY
A distributed ledger is a network that records ownership through a shared registry
CENTRAL DATABASE
CENTRALISED LEDGER
DISTRIBUTED LEDGER

Figure 1: Centralized vs decentralized. Source: coinspectator.com

Transactions and blocks are the two types of records that make up a blockchain. Multiple transactions are compacted and encoded in a Merkle tree in blocks and cryptographically approved. Each block has its own hash. These blocks are tightly connected with each other, where each block points to the hash of previous one, creating a long chain of these blocks, hence the name blockchain.

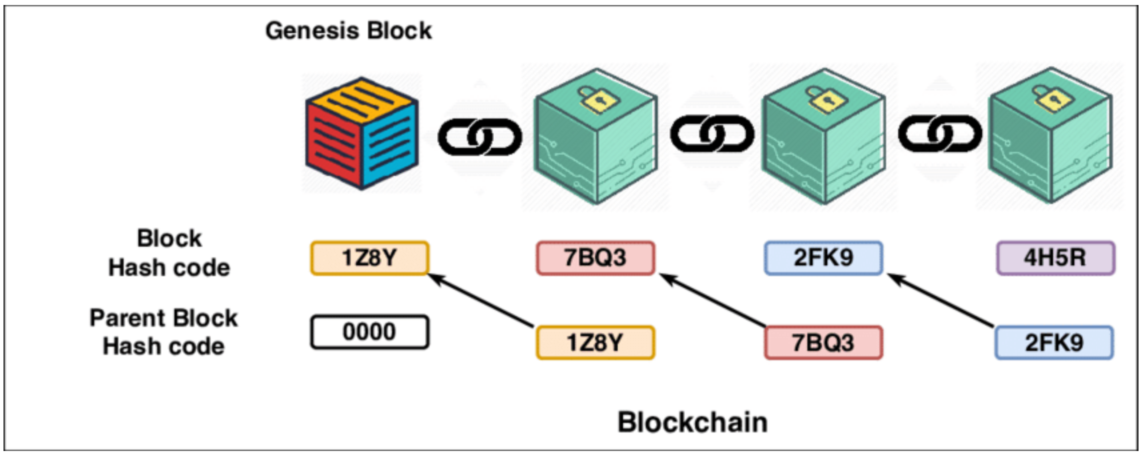

Figure 2: Blocks sequence. Source: [21]

This cyclical procedure validates the previous block's integrity back to the first block in the chain, known as the genesis block.

2.1.2 Blockchain architecture

As seen in the figure 3, blockchain can be thought of as a layer of a distributed peer-to-peer network running on top of the internet. It's similar to HTTP, FTP or SMTP, which all run on top of TCP/IP:

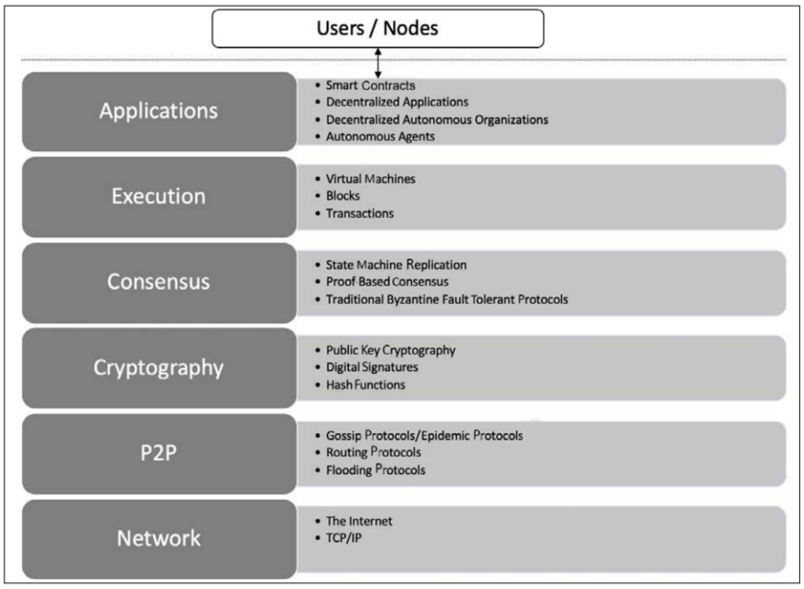

Figure 3: Blockchain architecture. Source: [1]

2.2 Consensus Mechanism

In distributed systems, consensus ensures that a state, value, or piece of data is correct and agreed upon by the majority of nodes. A consensus process ensures that this effort is carried out properly and independently of any interested parties, or, in the case of private permissioned networks, to meet other network objectives (such as centralized control).

In a nutshell, the consensus mechanism assures that each new block added to the blockchain is genuine, legitimate, and accepted by all nodes in the network. [5][6]

Proof-of-Work (PoW) is the most often used and well-known consensus method, and it is the essential underpinning of Satoshi Nakamoto's Bitcoin Blockchain that is also utilized by the Ethereum network. In a process known as "mining”, "nodes" solve hard, asymmetrical mathematical puzzles to produce new blocks, thereby demonstrating proof of work. Because this mathematical puzzle necessitates computing effort, the nodes must solve it as rapidly as possible in order to construct the new block and be rewarded for it. [5][7]

Other blockchain projects use a variety of different consensus algorithms. One of the most well-known is Proof-of-Stake (PoS), which is expected to replace PoW on the Ethereum platform by the second quarter of 2022. [8]

Proof-of-Stake is the fundamental mechanism that activates validators when enough stake is received. To become a validator on Ethereum, users must invest 32 ETH. Validators are assigned to produce blocks at random and are accountable for double-checking and confirming any blocks they do not make. The stake of the user is also used to incentivize positive validator activity. For example, a user can lose a portion of their share if they go offline (fail to validate) or lose their entire investment if they engage in wilful collusion. [9]

2.3 Ethereum

Ethereum [10] is a decentralized virtual machine that, at the request of users, executes programs known as contracts. Contracts are written in EVM bytecode [11], a Turing-complete bytecode language. A contract is a collection of functions, each of which is defined by a set of bytecode instructions. Contracts have the unique ability to send and receive ether (a cryptocurrency akin to Bitcoin [12]) from users and other contracts.

2.3.1 Addresses

The Ethereum blockchain is made up of account addresses, or so-called wallets, that may communicate with each other. Each account has a state and a twenty-byte address associated with it. An Ethereum address is a one-hundred-and-sixty-bit string that is used to identify an account. [10]

External accounts and contract accounts are the two types of accounts in Ethereum.

- **External accounts:** user accounts that are managed by private keys
- **Smart contract accounts:** Accounts that are governed by programming code that are known as smart contracts.

User account can send a message to any other external account, such as a simple value transfer, as well as to a contract, granting access to contract code functionalities. Contract accounts, on the other hand, are unable to initiate new transactions on their own. However, contract accounts can only initiate transactions in reaction to the receipt of additional transactions from an external account.

2.3.2 Ethereum Virtual Machine (EVM)

The Ethereum platform features the Ethereum Virtual Machine (EVM), a Turing-completely distributed virtual machine that runs scripts on the blockchain. Therefore, because the transaction is immutable, it is possible to develop a DAPP that works as programmed without the possibility of censorship, fraud, or third-party interference [10]. EVM is primarily a transaction-based state machine. This state machine just takes multiple inputs and transitions to a new state based on those inputs. The state has information stored at a particular point in time. In this way, the state machine acts as a computer and always remembers the state of something. If the EVM has a state change, it resembles a blank sheet, and that state changes each time something is written to the sheet. When the transaction is executed, the current state changes to the new state [13].

2.3.3 Smart contracts

Smart contracts are simply programs stored on a blockchain that run when predetermined conditions are met. They typically are used to automate the execution of an agreement so that all participants can be immediately certain of the outcome, without any intermediary's involvement or time loss [14].

Simple "if/when...then..." lines of instructions are written into code on a blockchain to make smart contracts work. When preset circumstances are satisfied and validated, the activities are carried out by a network of computers. When the transaction is complete, the blockchain is updated, which means the transaction can't be modified [6].

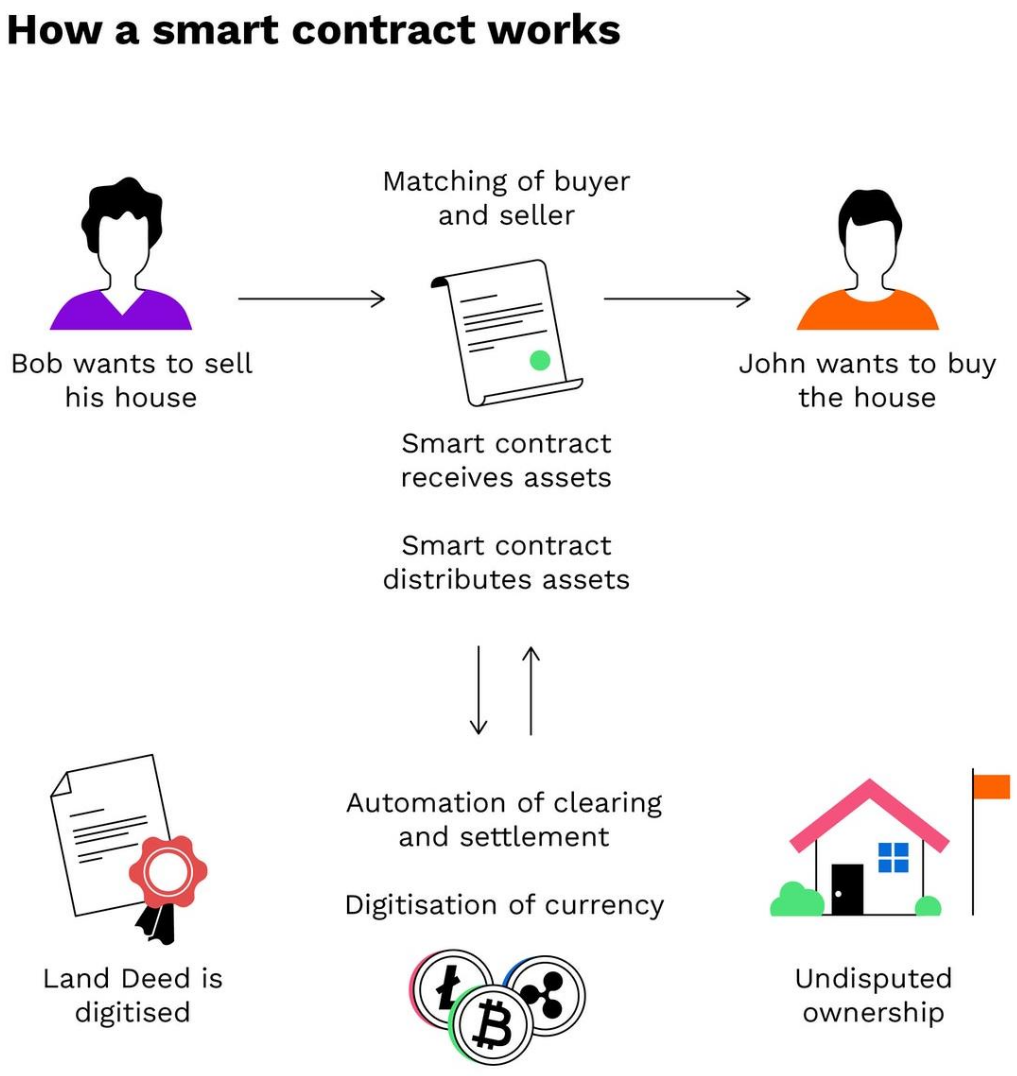
How a smart contract works
Matching of buyer and seller
Bob wants to sell his house
John wants to buy the house
Smart contract receives assets
Smart contract distributes assets
Automation of clearing and settlement
Digitisation of currency
Land Deed is digitised
Undisputed ownership

Figure 4: Smart contracts working: Source [bitpanda.com]

Users transmit transactions to the Ethereum network to construct new contracts, perform contract functionalities, and send ether to contracts or other users. All transactions are recorded on the blockchain, which is a public, append-only data structure. The condition of each contract and the balance of each user are determined by the sequence of transactions on the blockchain.

The settlement of digital relationships through the blockchain is great for decreasing the danger of breach of any rule specified in an agreement, boosting the security of value online transactions, because Ethereum smart contracts basically follow the set of instructions predefined by a developer. This is because the blockchain bears responsibility for adhering to the rules of the partnership since the smart contract, once launched on the blockchain, becomes independent of the parties' will, obeying nothing but their self-execution instructions under the circumstances embedded therein [1].

Because contracts have financial value, it is critical to ensure that they be carried out accurately. Conflicts in contract execution (due to failures or attacks, for example) are resolved using a consensus procedure based on PoW problems. When the adversary does not control the bulk of the network's processing capacity, contract execution is ideally right.

The smart contract can then be coded by a developer, though firms that use blockchain for business are increasingly providing templates, web interfaces, and other online tools to make smart contract construction easier. Therefore, the widespread of blockchain use in different fields.

2.3.4 Execution Fees

All miners in the Ethereum network should ideally execute each function call. The execution costs paid by users who call functions motivate miners to execute such work. Execution costs guard against denial-of-service attacks, which occur when an adversary attempts to slow down the network by requesting time-consuming calculations.

The cost of executing code is described in terms of gas and gas price, and the product represents the cost paid by the user. The transaction that initiates the invocation defines the gas limit that the user is ready to pay as well as the price per unit of gas. In general, the greater the price per unit, the more likely miners will opt to complete the transaction. Each EVM operation consumes a particular quantity of gas [1][11], and the overall charge is determined by the miners' entire sequence of operations. Gwei is the unit of measurement for gas. It's easier to utilize 10 gwei instead of 0.00000001 ETH because certain transaction fees might be modest. Other units of measurement are shown in the following table

Table 1: Ether denominations. Source [ethdocs.org]

#	Unit	Wei Value	Wei
1	wei	1 wei	1
2	Kwei (babbage)	1e3 wei	1,000
3	Mwei (lovelace)	1e6 wei	1,000,000
4	Gwei (shannon)	1e9 wei	1,000,000,000
5	microether (szabo)	1e12 wei	1,000,000,000,000
6	milliether (finney)	1e15 wei	1,000,000,000,000,000
7	ether	1e18 wei	1,000,000,000,000,000,000

Unless an exception is thrown, miners process a transaction until it completes normally. If the transaction succeeds, the remaining gas is returned to the caller; otherwise, the transaction's whole gas allocation is lost. When a computation uses all of the allocated gas, it throws an "out-of-gas" exception, and the caller loses all of the gas. An adversary seeking to launch a denial-of-service assault (for example, by running a time-consuming function) should set aside a large amount of gas and pay the ether associated with it. Miners will execute the transaction if the attacker picks a gas price that is consistent with the market; otherwise, if the price is too low, miners will not execute the transaction.

As a result, gas is a metric for computing effort. A predetermined amount of gas is allotted to each EVM operation. See table below for some examples

Table 2: Gas costs in gwei. Source [eips.ethereum.org]

#	Value	Operation	Gwei	Description
1	0x01/0x03	ADD/SUB	3	Addition/Subtraction operation
2	**0x02/0x04**	MUL/DIV	5	Multiplication/Division operation
3	0x16/0x17/0x18/0x19	AND/OR/XOR/NOT	3	Bitwise operations
4	0x56	JUMP	8	Unconditional jump

5	0x31	BALANCE	400	Get balance of the given account
6	0x40	BLOCKHASH	20	Get the hash of one of the 256 most recent complete blocks.
7	**0x55**	SSTORE	5000/20 000	Storage operation
8	**0xf1**	CALL	25 000	Message-call into an account
9	**0xf0**	CREATE	32 000	Create a new account with associated code.
10	**0x3b**	EXTCODESIZE	700	Get size of an account's code.

When it comes to gas prices, the way contracts are written has a direct impact on how well they function in terms of gas usage. Users may be subjected to unreasonably costly contracts if blockchain developers are sloppy or uninterested about gas economy. Solidity developers, on the other hand, who strive to optimize the code a lot might make it tough to read. As a result, programmers must strike a compromise between their concern about gas usage and code readability [15].

2.4 Blockchain explorers

Since blockchain is a decentralized and open system, it means that everybody can become a node and start exploring the stored data in the network. However, not every user will want or be able to become a member of the whole infrastructure as a full node due to hardware limitations. For that kind of situations, ordinary users and developers use blockchain explorers, which act as reliable API for the decentralized network with the functionality that allow them to examine and analyze recently added blocks and transactions along with the source code of uploaded smart contracts.

When it comes to the choosing the suitable explorer tool, the first thing that comes into mind is Etherscan. The reason for that is that was the only tool the author had real experience working with. However, it would also be good to mention an open-source alternative named BlockScout.

Etherscan

Etherscan is the oldest and most popular Ethereum explorers among other alternatives. Of course, it is a matter of taste, but here are some advantages of this explorer:

- The search can be performed using user account address, transaction hash, block number, tags, labels, and smart contracts address.
- It provides all the needed information about the pending and approved transactions with the sender, receiver, gas limit, price, and consumption.
- It offers developer services, such as an Application Program Interface (API) for acquiring smart contract code.
- Developers and companies can verify the source code of their contracts and make it transparent, so users could be convinced about the code quality on their own
- The Graphical-User-Interface (GUI) of the website is user-friendly and look clean

BlockScout

BlockScout is open source blockchain explorer that is not limited only to Ethereum, since it allows the switching between multiple networks. This project had its first beta version released in 2018. Here are the benefits:

- It allows the switching between variety of blockchain networks
- As a consequence of user inquiries, it provides block-oriented information.
- It is open project improved by community.

Security

Security tools are used to analyze smart contracts code statically and automatically. The author named Durieux, in his paper, selected nine automated security tools and analyzed a huge amount of Ethereum smart contracts. The two tools with the best results, either individually or in combination, were as follows [16]:

- **Mythril**: it is based on taint analysis, concolic analysis and control flow checking of the EVM bytecode, to search for data that make possible the discovery of vulnerabilities in the Solidity smart contracts.
- **Slither**: it is a static analysis framework that converts Solidity smart contracts into an intermediate representation called SlithIR to runs a suite of vulnerability

detectors and print visual information about security failures of the smart contracts.

2.5 General Data Protection Regulation

The General Data Protection Regulation (GDPR) is the world's most stringent privacy and security law. Despite the fact that it was designed and passed by the European Union (EU), it imposes duties on organizations anywhere that target or collect data about EU citizens. On May 25, 2018, the regulation went into effect. Those who break the GDPR's privacy and security regulations will face stiff penalties, with fines exceeding tens of millions of euros [17]. The strength of GDPR has seen it commended as a dynamic way to deal with how individuals' very own information ought to be taken care of.

2.5.1 Scope, penalties, and key definitions of GDPR

Scope

At the core of GDPR is private information. Extensively this is data that permits a living individual to be straightforwardly, or by implication, distinguished from information that is accessible. This can be something self-evident, like an individual's name, area information, or a reasonable online username, or it very well may be something that might be less immediately clear: IP locations and treat identifiers can be considered as private information.

Under GDPR there's additionally a couple of exceptional classifications of delicate individual information that are given more noteworthy insurances. This individual information incorporates data about racial or ethnic beginning, political feelings, strict convictions, enrollment of worker's organizations, hereditary and biometric information, wellbeing data and information around an individual's sexual coexistence or direction.

The vital thing about what comprises individual information is that it permits an individual to be distinguished - pseudonymized information can in any case fall under the meaning of individual information. Individual information is so significant under GDPR in light of the fact that people, associations, and organizations that are either 'regulators' or 'processors' of it are covered by the law.

Even though GDPR was proposed to European countries, it can be used or applied in other countries as well. The GDPR applies to companies that process the personal data of EU residents or citizens. However, it does not matter if your company is situated outside the EU or not. GDPR will apply to any company as long as they process and collect the data of European people.

Penalties

The fines for violating the GDPR are very high. There are two tiers of penalties, which max out at 20 million euro or 4% of global revenue (whichever is higher), plus data subjects have the right to seek compensation for damages.

3 State of the art

This chapter outlines one of the research paper's key objectives: an in-depth examination of the design, development, and deployment of Solidity smart contracts for business reasons. To avoid unethical practices, gas consumption difficulties, and security vulnerabilities, these patterns will be applied to corporate employees for the production of reusable smart contracts.

3.1 Literature Research

The use of well-known and well-studied approaches can be used to conduct a literature review. The following four types of reviews were examined for this study:

- **Narrative Literature Review**: This type of review is frequently used for the theoretical underpinning of articles, dissertations, theses, and course completion papers, but it does not utilize complex or thorough search tactics.

- **Systematic Mapping Study**: this word refers to studies that attempt to determine the present state of a topic in broad terms. These studies aim to determine who writers publish the most, as well as the institutions, years of publishing, and research methodologies used.

- **Systematic Literature Review**: it is used in scientific research to acquire, select, and assess the results of relevant studies using systematic and explicit approaches.

- **Multivocal Literature Review**: The same as previous literature reviews. However, the fundamental distinction between an MLR and an SLR or an SM is that, whereas SLRs and SMs solely employ academic peer-reviewed publications as input, MLRs also incorporate grey literature sources such as blogs, white papers, and online pages [18].

The style of literature review chosen by the author for this project was **Systematic Literature Review** (SMS) with the combination of **Multivocal Literature Research** (MLR) because it was thought to be the most suited for the nature of this research paper and its major focus – EVM compatible smart contracts. Ethereum's approach of blockchain and smart contracts is still relatively new technology. Since Ethereum has been launched in 2015, the author collected the needed information from sources published after the year of Ethereum initial release that contained enough references and citations.

Because most produced smart contracts currently belong to DAPPs that operate in the finance sector, conducting many transactions of significant monetary worth [19], the key element to be examined in this article was development security inside a company. As a result, smart contract development must be rigorous in order to minimize company financial losses due to code faults. This paper uses a variety of data sources, starting from internet resources and ending with academic books that were based on Solidity, smart contracts and Ethereum.

The authors Wohrer and Zdun, in on of their academic papers, performed a data analysis where they pointed out some developmental design patterns that would correspond to security needs of smart contracts [20].

Table 3: Security patterns of solidity development. Source [20]

#	Category	Pattern	Problem	Solution	Example Contract
1	Security	Checks-Effects-Interaction	Reentrancy attack. External calls from attackers can alter contract’s execution logic	To decrease the attack surface of a contract being modified by its own externally called contracts, follow a suggested functional code order in which calls to external contracts are always the final step.	CryptoKitties
2		Emergency Stop	Contract must be stopped in case of bug exposure	Include an emergency stop feature in the contract that may be used by a trusted person to halt sensitive functions.	Augur/REP
3		Speed Bump	When numerous heavy logic tasks are executed at the same time, a smart contract might fail	Extend the time it takes to complete sensitive tasks in order to prevent fraud.	TheDAO

4		Rate Limit	When numerous heavy logic tasks are executed at the same time, a smart contract might fail	Set a limit on how many times a job may be completed in a certain amount of time.	etherep
5		Mutex	Reentrancy attack. External calls from attackers can alter contract’s execution logic	To prevent an external call from re-entering its caller function, a mutex must be used.	Ventana Token
6		Balance Limit	Contract’s collected funds can be lost if attacked by hackers	Limit the maximum amount of cryptocurrency that can be put at risk in a contract.	CATToken

The authors used **Grounded Theory** technique to collect their data. Grounded theory (GT) is a research approach that focuses on the development of theory that is 'grounded' in evidence that has been collected and analysed in a methodical manner [21].

Moreover, they used the same technique with the combination of **Multivocal Literature Research** in another article that extended their previously setup security design patterns. This article introduces more functional design patterns on top of existing security principles, which is also a very important part in smart contracts that are focused for business needs. Some of those patterns are very critical when it comes to management of business applications such as control, authorization, lifecycle, and maintenance, since the company is responsible for their application, and they must have some management possibilities.

Table 4: Functional design patterns of solidity development. Source [22]

#	Category	Pattern	Problem	Solution	Example Contract
1	Action and Control	Pull payment	It is possible for money transfers to fail.	Let payment receivers withdraw the funds.	CryptoKitties
2		State machine	Multi-scenario logic is used in several DAPPs.	Contracts should be adaptable to a variety of phases and transitions.	DutchAuction
3		Commit and Reveal	Data secrecy is required in	Give data transactions	ENS Registrar

			several contracts.	secrecy for a specific amount of time before revealing the information.	
4		Oracle (Data provider)	Off-chain data is required by some contracts.	External data can be accessed by sending a request to an off-chain data carrier (mostly Chainlink).	Etheroll
5	Authorization	Ownership	Any account calls the execution of any contract method.	Restrict the access to some specific sensitive methods only to the contract owner	Euphoria Lottery
6		Access Restriction	Contract's collected funds can be lost if attacked by hackers	Limit the maximum amount of cryptocurrency that can be put at risk in a contract.	Etheroll
7	Lifecycle	Mortal	After a certain amount of time, some contract procedures are no longer required.	Implement self-destruction functionality into a contract	GTA Token
8		Automatic Deprecation		Determine the time interval for a contract method to be deprecated.	Polkadot
9	Maintenance	Data Segregation	When a contract is updated, data must be migrated to the new version.	Separate contract data in a separate contract to avoid data migrations when a new version is made.	SAN Token
10		Satelite	Contract updates are hindered by the immutability of deployed contracts.	By deploying auxiliary contracts (proxies), you can go around the blockchain's immutability and add additional	LATP Token

				functionality to a contract.	
11		Contract Register	Contract participants need to know its latest version.	Allow participants to look up the address of the most recent version of a contract.	Tether Token
12		Contract Relay		Create a contract that operates as a proxy for all requests, forwarding them to the most recent version of the contract.	Numeraire

4 Methodology

The main hypothesis of the author is that in order for company employees to develop more reliable and secure code for decentralized applications, the well-structured guideline must be developed that could point to the most important parts for business needs. Author will propose a solution that will be implemented in the practical part of this work.

The paper aims to provide theoretical understanding of how well designed and properly secured smart contracts for enterprise solutions must be developed. The author followed the qualitive research methodology. The steps for establishing a methodology framework for this study were as follows:

- Explain the core characteristics of blockchain security and what kind of limitations smart contracts have
- Explore the real-world attacks performed on smart contracts to identify the
- Investigate the security design patterns problems of smart contracts
- Describe and show how the testing and deployment of smart contracts must be accomplished.
- Describe the method of how to keep the private file in a safe way

5 Smart contracts Vulnerabilities and Data protection analysis

This chapter's goal is to cover the real-world attack cases and how the deficiency of security and data regulation impacts the company and its customers.

5.1 Real-world cases

5.1.1 The DAO hack

5.1.1.1 What is a DAO?

DAO stands for "Decentralized autonomous organization" and it is a blockchain-based cooperative that is collectively owned by its members and has rules that are defined and performed by code instructions. DAOs are decentralized management structures that replace centralized administration with a techno-democratic method in which investor-stakeholders vote on decisions. DAOs are constructed on top of blockchains and whose transactions are viewable on the underlying blockchain protocol. While The DAO was an early implementation of decentralized autonomous governance, decentralized autonomous models continue to be prominent in blockchain-related applications, particularly among DeFi platforms. Bitcoin can be considered as a very first implementation of DAO system [23].

The following describes how a DAO works:

- The smart contracts that will administer the organization are written by a group of people.
- There is an initial funding period during which people contribute to the DAO by purchasing ownership tokens – this is known as a crowd sale or an Initial Coin Offering (ICO) – in order to provide it with the resources it requires.
- When the financing period ends, the DAO becomes operational.
- People can then make ideas to the DAO for how the money should be spent, and members who have bought in can vote on whether or not these proposals should be approved.

5.1.1.2 The attack

The Slock company has established a crowdfunding campaign for a project dubbed "The DAO". The DAO has its own token on Ethereum blockchain network named "TheDAO token" that has been deployed on 30th of April of 2016. Every token is a smart contract, and the address of this specific token is *0xbb9bc244d798123fde783fcc1c72d3bb8c189413* [24]. The crowdfunding campaign was a huge success, raising 12.7 million Ether, which was worth $150 million at the time. However, an attacker discovered a flaw in the code that allowed a recursive withdraw function to be called without checking the current transaction's settlement. As a result, the attacker began the attack by making a tiny contribution and requesting withdrawal using a recursive code. This allowed him to withdraw about $70 million from the crowdfund [25]. The proceedings then took an intriguing turn. The Ethereum Foundation issued a warning to the attacker, threatening to halt the attack and freeze the account. The attacker replied that he was playing according to the contract, and that any intervention via a soft or hard fork would be a breach of contract, which he might pursue in court. However, he was able to halt the attack. Later, the Ethereum Foundation used a hard split to reclaim the funds, however this choice raised many questions about smart contract autonomy. This arduous work resulted in two Ethereum networks – Ethereum and Ethereum classic – as well as a great deal of debate.

5.1.2 Solana x Wormhole hack

5.1.2.1 What is a Solana?

Solana the same blockchain platform for decentralized and scalable applications as Ethereum. However, in comparison to other blockchains, Solana is substantially faster in terms of transaction processing and has significantly cheaper transaction costs. Its architecture is based on Proof of History (POH) that is used for confirming the order of events and the passage of time between them. Combined with consensus algorithms such as Proof of Work and Proof of Stake helps Solana to reach higher transaction speeds [26]. The project is open-source and was founded in 2017 and currently managed by the Solana Foundation in Geneva, with the blockchain developed by Solana Labs in San Francisco.

Solana network's smart contracts are written in another programming language named Rust rather than Solidity [27].

5.1.2.2 What is a Wormhole?

Wormhole is a communication bridge between Solana and other top decentralized finance networks, basically a way to move crypto assets between different blockchains. Existing projects, platforms, and communities are able to move tokenized assets seamlessly across blockchains and benefit from Solana's high speed and low cost. With the advent of a variety of high-value chains of varying shapes and sizes, interoperability between these currently segregated networks has become increasingly critical. Wormhole is a cross-chain messaging protocol secured by a network of Guardian nodes that tries to fill this gap. These guardians sign off on transfers between chains [28].

5.1.2.3 The attack

On February 2nd, the Wormhole bridge was hacked. The attacker took use of unpatched Solana Rust contracts, which were manipulated into crediting 120k ETH as having been deposited on Ethereum, allowing the hacker to mint the equivalent in wrapped whETH (Wormhole ETH) on Solana.

Wormhole's bridge is made up of two smart contracts, one for each chain. There was one smart contract on Solana and one on Ethereum in this attack example. Wormhole is an Ethereum bridge that accepts an Ethereum token, locks it inside a contract on one chain, and then issues a parallel token on the other chain. The transaction provided by the attacker contained legitimate signatures from the guardians and the overall processed went accordingly to smart contracts instructions.

The attacker seemed to use the *complete_wrapped* function which is called whenever someone creates whETH on Solana network. A *transfer_message*, which is essentially a message signed by the guardians that specifies which token to mint and how much, is one of the inputs that this method accepts. The problem itself lied in another smart contract and was based on signature verification.

The *load_instruction_at* function was utilized by the Wormhole contracts to ensure that the Secp256k1 function was called first. According to GitHub's internal commits, the load instruction at function was deprecated to be used on January 13th because it did not check that signature verification was being conducted by a whitelisted system address [29].

5.2 Existing development guidelines

This section considers solidity guideline development repositories driven by community and evaluate the design similarities between them. That analysis will help to understand how the development guideline usually looks like. The selection process was simple. The author chose the first 2 candidates from search result that corresponded to GitHub repositories. The author decided to search for git repositories because of the possible contributions made from other people to improve the relevancy of the provided materials.

5.2.1 Cytric’s implementation of Secure Smart Contracts

This repository was the first search result, and it looks simple enough. It has the following 3 sections [30]:

- Development guidelines
- Program analysis
- Learn EVM

Development guidelines

This section has its own subsections that describe 3 stages of development: design (before development), implementation (during development), and deployment (after development).

Those sections describe the main and important concepts of Solidity development in a simple form. This part of repository discusses about the following points:

- **Design**: documentation, code optimization, contracts upgradeability
- **Implementation**: Function composition, Inheritance, Events, Dependencies, Testing, Solidity version
- **Deployment**: This part describes the best practices to do after the deployment, but does not discuss about best practices while deployment process

Also, this section of repository describes the checklist regarding the interaction with the crypto tokens and high-level processes that are recommended to be followed while developing smart contracts

Program Analysis

In this section, the author examines the possibility of program code analysis using some of the existing tools as examples and recommendations. Three of the following distinctive testing and program analysis techniques were proposed:

- Static analysis with **Slither**. All the paths of the program are approximated and analysed at the same time, through different program presentations (e.g. control-flow-graph)

- Fuzzing with **Echidna**. The code is executed with a pseudo-random generation of transactions. The fuzzer will try to find a sequence of transactions to violate a given property.

- Symbolic execution with **Manticore**. A formal verification technique, which translates each execution path to a mathematical formula, on which on top constraints can be checked.

Each technique has advantages and pitfalls, and will be useful in specific cases:

Technique	**Tool**	**Usage**	**Speed**	**Bugs missed**	**False Alarms**
Static Analysis	Slither	CLI & scripts	seconds	moderate	low
Fuzzing	Echidna	Solidity properties	minutes	low	none
Symbolic Execution	Manticore	Solidity properties & scripts	hours	none*	none

5.2.2 Consensys' implementation of Secure Smart Contracts

Consensys repository has more contributors and therefore, more materials and examples. The learning materials consists of 4 learning categories: General Philosophy, Development recommendations, Attacks, Security tools [31].

General philosophy

Smart contract programming necessitates a different approach to engineering than it may be accustomed to. Because the cost of failure is high and adjustments can be difficult, it's more related to hardware programming or financial services programming than web or mobile development. As a result, defending against known vulnerabilities is insufficient. Instead, the developers will need to master the new following development philosophy:

- Prepare for Failure

- Stay up to Date

- Keep it Simple
- Rolling out
- Blockchain Properties
- Simplicity vs. Complexity

Development recommendations

The development recommendations are split into six categories.

Category	Description
General	Guiding principles that should be kept in mind during development.
Precautions	Principles that prevent attacks in general or avoid excessive damage in the worst-case scenario.
Solidity-specific	Helpful tips when building smart contracts in Solidity - including interesting quirks.
Token-specific	Recommendations to honour when dealing with or implementing tokens
Documentation	Guidelines on how to properly document smart contracts and the processes surrounding them.
Deprecated	Vulnerabilities that were applicable in the past but can be reasonably excluded nowadays.

Attacks

The following is a list of known attacks which you should be aware of and defend against when writing smart contracts.

Category	Description
Reentrancy	Intra- and inter-function reentrancy attacks and potentially faulty solutions to them.
Oracle Manipulation	Manipulation of external data providers and potential solutions to oracle security issues.
Frontrunning	A definition and taxonomy around frontrunning and related attacks.
Timestamp Dependence	Attacks relating to the timing of a transaction.
Insecure Arithmetic	Integer overflows and underflows.

Denial of Service	Denial of service attacks through unexpected reverts and the block gas limit.
Griefing	Attacks relating to bad faith players around a smart contract system.
Force Feeding	Forcing Ether to be sent to smart contracts to manipulate balance checks.
Deprecated/Historical	Attacks that are part of Ethereum's history and vulnerabilities that have been fixes on a (Solidity) compiler level.
More	Where to find more information about vulnerabilities and weaknesses.

Security tools

This section is about tools that can detect vulnerabilities or help developers maintain a high code quality to reduce the likelihood and impact of vulnerabilities.

Category	**Description**
Visualization	These tools are aimed at visualizing, EVM bytecode, smart contracts, and their control flow graphs.
Static and Dynamic Analysis	Tools that employ various means of program analysis to find vulnerabilities and weaknesses.
Classification	Resources attempting to classify vulnerabilities and weaknesses in smart contracts.
Testing	Tools for running, measuring, and managing smart contract related tests
Linters and Formatters	Any tools that highlight code smells and make smart contract code adhere to format standards.

5.3 Smart contracts and security

On Ethereum, smart contracts are often authored in a high-level language and then compiled into EVM bytecodes. Solidity is the most well-known and widely utilized, and it is even employed in other blockchain platforms. Solidity is a contract-oriented high-level programming language with JavaScript-like syntax [32].

A smart contract analysis carried out by Bartoletti and Pompianu [19] shows that Bitcoin and Ethereum primarily focus on financial contracts, which are critically important in business field. The direct handling of the assets means that the flaws are more likely to be relevant to the security and have greater financial consequences that the errors on typical applications, as evidenced by the DAO attack on Ethereum described on section 5.1.1

5.3.1 Security Challenges in Ethereum

For a variety of reasons, security is a top priority in Ethereum programming. Based on author’s development experience, the following limitations of Ethereum blockchain can be specified:

- **Unfamiliar execution environment**. Ethereum is not like any other centrally managed execution environment, whether mobile, desktop, or cloud. Developers aren't used to their code being run by an anonymous, mutually distrusting network of profit-driven nodes

- **EVM stack**. The Ethereum stack (the Solidity compiler, the EVM, the consensus layer, and so on) is still in development, with security flaws being revealed all the time

- **Contracts immutability**. Patching a deployed contract is not possible. It means that if a bug is found after a smart contract deploy, then users’ security, who have interacted with the contract, will be in danger, since no authorities are able to alter the deployed contract source code

- **Attackers who are financially motivated by anonymity**. When compared to other forms of cybercrime, smart contract exploitation provides more profits (the prices of cryptocurrencies have been rapidly increasing), easier cashing out (ether and tokens are instantly marketable), and a reduced chance of punishment due to anonymity.

- **The rate of development is extremely fast**. Companies that use blockchain technology try to release goods quickly, since they want to be the first to adopt existing centralized application in a decentralized world or come up with a new innovative DAPP. By doing that, they typically sacrifice the security of a final product.

- **High-level language that isn't up to par**. Some think that Solidity encourages programmers to use risky development techniques [33][34][35].

In this paper, the author will ignore the security flaws of Ethereum core system and will concentrate on security from the perspective of a smart contract developer who use Solidity as his main programming language.

5.3.2 Design Challenges and Patterns Usage

Understanding how smart contracts are used and executed may aid smart contracts designers in developing new decentralized applications, through their designs, avoid vulnerabilities such as those discussed below. A basic summary of the usual design patterns that are intrinsically frequent or practical when talking about the codification of smart contracts is provided in the following points.

5.3.3 Security analyzer for testing

Every piece of application code must be tested to verify the proper work of the base functionality. However, before doing manual testing and code reviews, the developer should use automatic vulnerability detection tools, to find common and obvious security pattern flaws.

By examining some academic works [36][37][38] and relying on previous development experience, the author has chosen 2 security analysis tools that suit best the business needs. Initially, Smartcheck and Oyente were also on the author's radar. However, it seemed that Smartcheck left out being outdated and abandoned by developers. Oyente also left the comparison out since it only works with Solidity up to version 0.4.19.

5.3.3.1 Mythril

Mythril is an EVM bytecode security analysis tool. It finds security flaws in EVM-compatible blockchains' smart contracts. It means that the smart contracts not necessarily need to be written using Solidity. It detects a variety of security flaws using symbolic execution, SMT solution, and taint analysis. It's also utilized in the MythX security analysis platform (together with other tools and approaches) [39].

5.3.3.2 Securify 2.0

Securify 2.0 is a continuation of tool Securify that has been described in [40]. The Ethereum Foundation and ChainSecurity have cooperated to create Securify 2.0, a security scanner for Ethereum smart contracts. Securify's main research was carried out at ETH Zurich's Secure, Reliable, and Intelligent Systems Lab. The main features of this tool are following:

- It supports 37 different vulnerabilities
- Uses Datalog to implement a unique context-sensitive static analysis.
- Analyzes contracts written in Solidity version equal or higher than 0.5.8.

5.3.3.3 Tools performance

To compare the two above chosen tools, the author decided to use publicly available smart contracts with vulnerabilities obtained from not-so-smart-contracts repository on GitHub and SWC Registry. These sources contain a variety of smart contracts with potential weaknesses. Here what the author has to say:

Mythril

Mythril recognized array out of bounds access, arithmetic overflows and underflows, poor randomization, and unprotected ether withdrawal. However, it didn’t consider an outdated compiler version, and solidity coding best practices such as deprecated functions, state variables default visibility.

Securify 2.0

Securify 2.0 didn’t perform numerical analysis; hence it did not identify arithmetic overflow and underflow, a problematic randomness, or unprotected ether withdrawal. However, it did detect out-of-bounds array access and transaction order dependency. Overall, Securify 2.0 discovered more vulnerabilities than Mythrill.

5.4 User data protection under GDPR

User’s data security is very important to data owners. Companies are also interested in secure data storage because of the high penalties applied by GDPR. Based on regulations of GDPR and Blockchains systems basic characteristics and limitations, the following points are defined regarding user data protection:

5.4.1 Data transparency

Since blockchain is an open and transparent system, all the stored data is publicly accessible. It means that every user can get access to any information without any extra authorization and authentication. GDPR also proposes the another understanding of user's data transparency. However, by analyzing both versions of definition of "transparency", it can clearly be seen said that they mean different understandings of publicity and availability. Blockchain's "transparency" means that stored data is fully public to anyone, while GDPR insist that the data is also public and available, but only to the initial owner

5.4.2 Data Minimization, Accuracy and Storage Limitation

GDPR require the applications to allow user to modify his provided data. The user can alter the data handling by recalling the contest on which the data can be processed with further possibilities to completely erase the data from service provider's storage. However, the blockchain by the core design doesn't allow the modification of stored data because integrity is a strong part of Blockchain technology. That is why the blockchain does not meet the user's data control requirement of GDPR. However, Ethereum blockchain, gives opportunity to alter the data stored in smart contracts, which makes it possible to follow the needed regulations. But the ordinary transactions executed in Ethereum blockchain are immutable.

5.4.3 Purpose limitation

Sometimes the personal information can be used not for the purposes it has been collected for. Since Ethereum's Blockchain smart contracts can be deployed by any authority, the regulation of data collection gets almost impossible to track. That is why, depending on the use of the specific smart contract, the personal data must be collected for specified, explicit and legitimate purposes and not further processed in a manner that is incompatible with those purposes. This principle depends on who is the data manager, since it is the controller who decides the purposes of the processing. Each smart contract code must therefore be reviewed uniquely including the purposes for which it is built.

5.5 Hardware wallets

A portable hardware wallet is physical device used to assist clients with overseeing and store cryptographic forms of money in a safe way. Hardware wallets are decentralized, expecting clients to assume total command and responsibility for private keys (recovery phrase). During the signing process of transaction, an equipment wallet will just utilize the private keys to sign in an offline environment to stay away from openness to online assaults. There are lots of companies that provide ready solutions like Ledger, Trezor, SecuX, ELLIPAL, Keystone, etc. For this paper, the author decided to review a Keystone portable wallet project as an example. The choice was made based on author's hand on experience with that device.

Keystone hardware wallet is an offline device that store the private credentials of blockchain account. The Keystone developers do not collect the users' personal credentials. However, they track the usage from their mobile companion applications with the intent to make the service better to the end user [41]. The official webpage states that the Keystone companion app collects data only for debugging and performance or stability improvement purposes. Here are the benefits of this portable wallet:

Air-gapped design

This design means that no credential information is being transmitted using wireless or wired data-transfer and communication technologies such as Bluetooth, NFC, LTE, USB, etc. Since your portable device is never connected to the internet and it does not transmit any data and emit signals, it makes it hard for attackers to intercept the data and get access to user's credentials. Therefore, the Keystone wallet uses a QR code approach, where user scans and verifies every transaction that he sees on mobile companion application.

Secure element storage and open-source firmware

Keystone Hardware Wallet utilizes a bank-grade Secure Element to produce true random numbers, derive private and public keys, sign transactions, and protect private keys from being leaked if an attacker has physical access to the device. Keystone Hardware Wallet is the first hardware wallet that has open-source Secure Element firmware code. Having that, allows you to verify all core cryptographic operations such as how private keys are generated and contained entirely within the Secure Element.

Self-destruct mechanism

Portable wallets are meant to be always near the owner, and it makes them physically vulnerable if it is lost. To prevent an attacker from performing a side-channel attack from succeeding if you lose your device, the Keystone wallet has been assigned with a self-destruct mechanism. Upon detection of disassembly, the mechanism will wipe the private keys and any sensitive information so that an attacker cannot extract it from your device.

6 Proposed Solution

This section aims to practically demonstrate the proposed guideline that covers the design, security, testing, and deploy parts of smart contracts development that would correspond to business needs.

6.1 Design

6.1.1 Development tools

The following tools were chosen according to their relevance and this paper's host company employees' experience with them. Those tools will be good to be familiar with in order to perform well as a blockchain developer. These choices are following:

- **Remix** or **Visual Studio Code** as Integrated Development Environment for Solidity programming language. The familiarity with both is preferred. For the beginning purposes the author would suggest using Remix because it is online and has built-in debugger all covered in good-looking and user-friendly Graphical-User-Interface (GUI)

- **Truffle** as Solidity Developmental Environment. This tool was chosen on the choice basis made by the author and host company blockchain development leader.

- **JavaScript** scripting language and **NodeJS** runtime environment are chosen because of popularity of those technologies' usage in Front-End applications because it is natively supported in modern web browsers.

- **Web3.js** is a JavaScript set of libraries that let you use HTTP, IPC, and WebSocket to communicate with a local or distant blockchain node [38]. It supports the connection to all networks that has a compatibility with Ethereum Virtual Machine (EVM) [42].

- **OpenZeppelin** is the standard for secure blockchain applications. The core team have developed ready-to-use solidity libraries and smart contracts that have been properly tested and comply with all the highest standards of secure decentralized software development [43].

- **Ganache** is a personal blockchain that enables the building of Ethereum distributed applications quickly. Ganache can be used throughout the development cycle, allowing you to build, deploy, and test your DAPPs in a secure and predictable environment. In a nutshell, it's a local blockchain emulator [44].

- Developers need to follow the **design patterns** mentioned in Tables 3 and 4, where the reasons for their selection were explained

6.1.2 Standards

Developers might have to develop smart contracts with functionality that has already been implemented and standardized by community. In order to develop applications following established concepts, develop must stick to Ethereum standards and use existing libraries and smart contracts that have proved to be reliable and secure. Since most of the projects and start-ups interact with tokens, the company employees must be familiar with the different type of token standards.

Here is the list of existing and widely used and approved standards that a company employee must be familiar with:

- **ERC-20 -** An open standard for fungible (interchangeable) tokens such as voting tokens, staking tokens, and virtual currencies.

- **ERC-721 -** A non-fungible token interface, such as a deed for artwork or a music.

- **ERC-777** is a new token standard that improves on ERC-20.

- **ERC-1155** is a token standard that allows both fungible and non-fungible assets to be included [45].

Moreover, in case if a developer has to implement and deploy one of the tokens of type listed above, then the use of OpenZeppelin libraries must be ensured. The team behind OpenZeppelin have already developed a tokens implementation that follow the established standards.

6.1.3 Control and Management

Problem: If there are no restrictions, any Ethereum account or contract address can call public methods from other contracts. Mostly, business projects require some administration from the company authorities. It's understandable that determined procedures in a contract should only be called from the contract owner's address.

Solution: This technique may be used to tackle that challenge in a reusable fashion every time the contract owner has to perform a sensitive task.

6.1.3.1 Ownership

This pattern is simple enough. Demonstrated in Appendix 1.7, this pattern allows to specify the owner of the smart contract, who can execute the specific methods that are restricted to be run by owner address only.

6.1.3.2 Roles

This design pattern gives you an opportunity to distinct sorts of addresses (users) to have different roles. In Appendix 1.8 an example of smart contract code enables the design of reusable methods for adding and removing addresses from certain roles, as well as determining whether or not a particular address has a specified role

6.1.3.3 Multiple ownership authorization

Problem: Having all the access granted to one address is very risky, since private key of owner address might be hacked due to variety of reasons. That is why, like in centralized apps, in the realm of DAPPs, it's only logical that some duties are carried out by a group of people rather than a single owner address. This eliminates single points of failure in a decentralized manner.

Solution: Appendix 1.9 illustrates design code example that allows contract methods to be run only if a set of participants has given their permission.

6.1.4 Upgradeability

Problem: Contracts on Ethereum blockchain are immutable by the nature. It means that after deployment, nobody has an opportunity to change the code of the smart contract. Therefore, it is critically important to fix all the bugs before final release of your code on blockchain network. However, no company is safe from bugs occurrence in their applications. If a serious bug is found after official launch of smart contract, then the consequences are unpredictable and mostly are negative.

Solution: Use upgradeability pattern and deploy proxy that would interact with separate contract with implementation of main business logic.

OpenZeppelin provides a good documentation and ready-to-use libraries that would help any developer to start making their smart contracts mutable and fixable in case of failures.

6.1.5 Data regulation

Based on the outcomes of section 5.3, the author suggests that the following points must be followed and considered by a company to comply with Global Data Protection Regulation:

- Keep the needed data requirements as simple as possible. Collect only that data, that will fully be used by the contract and nowhere else
- Do not store the user's private information that could identify a user such as name, age, geolocation, IP address, document ids, etc.
- The data usage and purposes must be documented and clearly stated in the source code or frontend
- Give user an opportunity to alter his own data by calling a method from the contract

6.2 Security

This section is going to describe critical security vulnerabilities with the proper solution that have to be considered properly to develop reliable and secure smart contracts. Following subpoints examine only serious security flaws that might harm the logic of smart contracts used for enterprise needs.

6.2.1 Reentrancy

Problem: Smart Contracts on EVM Blockchains can invoke other contracts to perform their duties by performing external calls, which is the foundation for reentrancy attacks, in which a called contract fraudulently alters the caller contract's existing internal state. **Solution**: Following patterns below help to solve this vulnerability.

6.2.1.1 Checks-effects-interaction

The design illustrated in Appendix 1.1 provides a systematic code instructions ordination every time external calls are needed to avoid reentrancy. The caller contract performs validations (checks) first, then any necessary internal state modifications (effects), and finally the external call (interaction)

6.2.1.2 Mutex

This pattern, like checks-effects-interaction, aims to eliminate reentrancy attacks, but makes it possible using different approach. By looking at Appendix 1.2, it can be seen that contract initializes a Boolean variable that acts as a lock for external calls. To make code reusable, this logic is defined as modifier that can be applied to several different methods at the same time. When a contract calls an external contract, the locker is engaged before the call and deleted when the call is completed.

6.2.2 Balance limit

Problem: Due to their popularity of cryptocurrencies original intent to manage valuable transactions, financial concerns became critical in Ethereum smart contracts. As a result, Solidity programmers should constantly keep this condition in mind, especially when developing a contract for a business need. As “TheDAO” attack has shown, little bug in a smart contract can lead to a big financial loss that is followed with a huge drop of company’s reputation.

Solution: The balance limit pattern aids in the function demonstrated in Appendix 1.3, by allowing the setting of a monetary amount threshold to monitor and validate operations that cannot exceed a specified limit.

6.2.3 Emergency stop

Problem: Testing is essential in Solidity programming. Smart contracts developers have to strictly follow the Test-Driven-Development (TDD) technique and pay a huge attention to testing while development process. It helps to eliminate defects and failures in final application before the final deploy. However, they can nonetheless occur.

Solution: The design pattern in Appendix 1.4 allows a contract's operation to be paused in an emergency, such as the discovery of flaws or security vulnerabilities in important activities or methods.

6.2.4 Time management

Problem: Some methods in smart contracts might require an execution at specific time period because of business logic. Controlling time operations, for example, is quite prevalent in Solidity contracts in the gaming or gambling sectors

6.2.4.1 Rate limit

By restricting the frequency of how many times a certain job may be completed over a defined period, pattern in Appendix 1.5 aims to regulate and manage time

6.2.4.2 Speed Bump

Some of the key purposes of the Ethereum platform, which is critical to modernize and speed operations, include automating jobs and removing intermediaries. Nonetheless, it is occasionally more convenient to perform some sensitive duties more slowly. Appendix 1.6 demonstrates a pattern that makes it possible to slow down task execution in accordance with contract constraints.

6.2.5 Integer Overflow

Problem: An integer overflow happens in computer programming when arithmetic operations attempt to produce a numeric value that is outside of the range that can be represented with a given number of digits - either higher than the maximum or lower than the lowest representable value.

Solution: When using Solidity version of 0.8.0 or higher, then the arithmetic overflows by default throws and error and reverts the transaction. However, in case of development with a lower compiler version, the use of **SafeMath** library must be considered. That library is included in OpenZeppelin’s repository as well.

6.2.6 Phishing with origin of the sender

Problem: An attacker can convince user to call a contract’s method on his behalf via attacker’s contract that acts as a bridge. It can be harmful because a malicious contract can deceive the owner of a contract into calling a function that only the owner should be able to call. That happens a lot among new developers, because they assume that msg.sender and tx.origin are similar keywords.

Solution: The **tx.origin** refers to the initial external account that initiated the original transaction, but the **msg.sender** refers to the last account that invoked the function (which might be external or another contract account). When a method has to validate that the transaction has been initiated directly by the user, then the msg.sender has to be used instead of tx.origin.

6.2.7 Signature Replay

Problem: Sometimes it might sound good to use a pattern of signing messages off-chain and transfer it further to the contract that needs that signature before performing some specific function. This technique is helpful when you need to reduce the transaction numbers performed by user and the overall gas consumption per each transaction. However, the same signature can be used to execute a contract method more than once. If the signer's goal was to authorize a transaction just once, this may be dangerous.
Solution: Sign messages with the nonce and address of the smart contract to make each transaction hash unique.

6.3 Testing

Testing is the most important part in smart contracts development because any bugs found after a final deploy become impossible to fix. That is why code logic testing has to be done in two environments, local and public test networks.

6.3.1 Truffle suite and Ganache

For the local testing, there are bunch of tools that help writing tests easier. The author had a real experience writing tests using built-in automated testing framework tools of **Truffle** suite. It requires **Ganache** virtual blockchain environment to be installed and configured. Truffle allows you to write test cases using both Solidity and JavaScript languages.

6.3.1.1 Writing Tests in JavaScript

Truffle provides a strong foundation for writing JavaScript tests by using the Mocha testing framework and Chai for assertions. It is advised to see Mocha's official documentation if you're not comfortable with developing unit tests in Mocha.

6.3.1.2 Writing Tests in Solidity

As.sol files, Solidity test contracts coexist with Javascript tests. They will be added as a distinct test suite per test contract when the truffle test is run. These contracts keep all of the advantages of Javascript testing. When writing test using this approach, the following points must be taken into account:

- Solidity tests should not extend from any contract. This reduces the number of tests you have to run and provides you total control over the contracts you build.
- No assertion library should be used in solidity testing contract because Truffle comes with a default assertion library. However, this can be changed at anytime

6.3.2 Security testing

There are tons of vulnerability analyzer tools available for smart contracts written in Solidity. However, the use of these tools does not fully guarantee the absence of bugs in final code. They only help to automize the security analysis part of smart contracts development. Therefore, any of these tools must be used only as additional checking before committing the code for audition.

Based on the analysis in section 5.5 the author suggests using either Mythril or Securify 2.0 security analyzer tools.

6.4 Deploy

Before putting company's funds and reputation on the line, contracts should have a long and thorough testing phase. The following points have to be ensured before deployment:

1. Have a complete test suite with 100% coverage (or close to it)
2. Deploy it on your local blockchain network node such as Ganache
3. Deploy your contract on a public test network
4. Make a public beta testing, involving other users to interact with your contract

6.4.1 Truffle deploy

Since the author chose Truffle as main smart contracts development framework, the deploy part will also be maintained using it.

6.4.1.1 Migrations

Migrations are scripts written in JavaScript that aid in the deployment of Ethereum contracts. These files are in charge of staging your deployment activities, and they're created with the expectation that your deployment requirements may evolve over time. You'll write fresh migration scripts as your project grows to keep up with the blockchain's progress. To run the migrations (deploy), execute the following command

```
truffle migrate
// or
truffle deploy
// both of the commands perform the same actions
```

This will perform all migrations in the “migrations” directory of your project. Migrations are, at their most basic level, a set of controlled deployment scripts. If your migrations were previously successful, truffle migrate will begin execution with the most recent migration and only run freshly generated migrations. If no fresh migrations are available, truffle migrate will do nothing. You may use the —reset option to start again with all of your migrations. However, in order to keep track of deployed smart contracts, an additional contract named Migrations.sol must be deployed initially.

Initial migration

To utilize the Migrations functionality in Truffle, a “Migrations” contract must be deployed. This contract must have a specified interface, but you are allowed to change it as you see fit. This contract will be deployed as the first migration in most projects and will not be modified again.

Upgradeability

Truffle deploy can also be used to deploy upgradeable contracts. Since, upgradeability is achieved using proxy contracts, truffle will need additional functionality to automatically deploy additional contracts as proxy and check your source contracts for upgradeability errors. The author recommends using the combination truffle deploy with the combination of OpenZeppelin upgradeable contracts. For that purpose, the OpenZeppelin team developed an additional dependency for Truffle framework and can be accessed via this link.

Configuration

The *truffle_config.js* file is configured to setup the contracts compilation and configure deploy settings like on what specific blockchain network to migrate contracts and from what account address. The most important part in that file is the user's private key or mnemonic phrase. It is prohibited, from developers' concepts to hardcode any sensitive information directly inside of config files. That is why the private key of main accounts must be stored separately and injected into the configuration while the script execution time. The author would suggest at least using environmental variables to load sensitive information from one single file that would be well-encrypted and stored.

6.4.2 Securing deployer account private key

It is very crucial for the business company to keep the private keys of their main decentralized accounts under control. The best approach of keeping the private keys safe, by the author's own opinion, is by using hardware wallets.

Being reviewed in section 5.5, the use of Keystone Pro is recommended, or any other alternative of portable hardware wallets can also be implemented.

In case if company can't afford or does not trust hardware wallets, the private key file must be well-encrypted and saved at least in 2 storages that are physically far from each other to prevent a single point of failure.

7 Discussion

This chapter will discuss the findings that were collected and examined in the previous chapter.

Most of the development guidelines and best-practices have almost the same structure. They recommend variety of techniques and tool for design, development, testing and deploy steps and how to secure code from vulnerabilities. The existence of such guidelines clearly points to the fact that EVM compatible smart contract is a relatively new concept and that it needs more clarification and research. The author's proposed practical solution also followed the same structure established by community to ensure the full coverage of development process.

Moreover, the author reviewed the GDPR rules in the use with smart contracts and noticed that other development recommendations do not consider any kind of data protection regulation. It is a very crucial part for every business company that collects data of its users. Blockchain's open nature is a critical issue when it comes to user's data protection. Design of Ethereum blockchain assure that smart contract's storage information will always be available for read purpose, which makes it impossible to provide the private data using authorization.

8 Summary

This paper analyzed the known smart contracts vulnerabilities and risky solidity development patterns to build a secure guideline that covers all the aspects of smart contracts development with the best practices for the business needs.

In theoretical part it defined and compared the existing security guidelines. Then, it investigated the theoretical design of blockchain technology to understand what kind of limitations it has and how the future applications would harm from it. Also, the use GDPR has been considered to examine the compatibility of smart contracts environment with existing data protection rules.

In practical part the author combined all the theory and analysis results to build a secure smart contract development guideline for use in a private company. In that guideline the whole development cycle from designing till final deploy has been discussed. The author’s chosen methods and tools were based on paper reviews, performed analysis, and the own experience in smart contracts field.

Moreover, the author recommends reviewing the development guideline every year to make additional changes in the structure of paper to be up to date. Also, it is advised for the developers to always monitor the latest resource to find more vulnerabilities, possible attacks, and new design patterns.

References

[1]. Imran Bashir, Mastering Blockchain - Third Edition, 2020.
[2]. Nomura Research Institute, “Survey on blockchain technologies and related services”, 2016.
[3]. Tao Zhang, Zhigang Huang - Blockchain and central bank digital currency.
[4]. Mohamed Torky, Aboul Ella Hassanien, “Integrating Blockchain and the Internet of Things in Precision Agriculture: Analysis, Opportunities, and Challenges”, 2020.
[5]. Parma Bains, “Blockchain consensus mechanisms: a primer for supervisors”, 2018.
[6]. Tapscott, Don and Alex Tapscott, “Blockchain revolution: how the technology behind bitcoin and other cryptocurrencies is changing the world”, 2018.
[7]. Ammous, Saifedean, “The bitcoin standard: the decentralized alternative to central banking”, 2018.
[8]. Proof-of-stake (PoS) | ethereum.org. https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/ [Online; accessed 01-March-2022]
[9]. The Ethereum Oracle: An Interview with DARMA Capital’s Andrew Keys On The Future Of Ethereum. https://www.gfinityesports.com/cryptocurrency/ethereum-oracle-interview-andrew-keys-darma-capital-future-eth/ [Online; accessed 01-March-2022]
[10]. Buterin Vitalik, “Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform”, 2014.
[11]. Gavin Wood, “Ethereum: a secure decentralised generalised transaction ledger”, 2014.
[12]. Satoshi Nakamoto, “Bitcoin: a peer-to-peer electronic cash system”, 2008.
[13]. Antonopoulos, Andreas M and Gavin Wood, “Mastering ethereum: building smart contracts and dapps”, O’Reilly Media, 2018.
[14]. What are smart contracts on blockchain? - IBM. https://www.ibm.com/topics/smart-contracts. [Online; accessed 05-March-2022]
[15]. Zou, Weiqin et al, “Smart contract development: Challenges and opportunities”, In: IEEE Transactions on Software Engineering, 2019.
[16]. Durieux, Thomas et al, “Empirical Review of Automated Analysis Tools on 47,587 Ethereum Smart Contracts”. In: arXiv preprint arXiv:1910.10601, 2019.
[17]. What is GDPR, the EU’s new data protection law?. https://gdpr.eu/what-is-gdpr/ [Online; accessed 02-March-2022]
[18]. Vahid Garousi, Michael Felderer, Mika V. Mäntylä, „The need for multivocal literature reviews in software engineering: complementing systematic literature reviews with grey literature”, 2017.
[19]. M. Bartoletti and L. Pompianu, “An empirical analysis of smart contracts: Platforms, applications, and design patterns”, 2017.

[20]. Wohrer, Maximilian and Uwe Zdun, “Smart contracts: security patterns in the ethereum ecosystem and solidity”, 2018.
[21]. Helen Noble, Gary Mitchell, “What is grounded theory?”, 2016.
[22]. Wohrer, Maximilian and Uwe Zdun, “Design patterns for smart contracts in the ethereum ecosystem”, 2018.
[23]. What is a DAO? An attempt to clarify a complex idea. http://web.archive.org/web/20201101015803/https://daobase.org/what-is-a-dao/. [Online; accessed 03-March-2022]
[24]. TheDAO Token on Etherscan. https://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413. [Online; accessed 03-March-2022]
[25]. The story of the DAO, and how it shaped Ethereum. https://www.coininsider.com/what-happened-to-the-dao/. [Online; accessed 03-March-2022]
[26]. Anatoly Yakovenko, “Solana: A new architecture for a high performance blockchain”, 2017.
[27]. History | Solana Docs. https://docs.solana.com/history. [Online; accessed 03-March-2022]
[28]. FAQs - Wormhole. https://docs.wormholenetwork.com/wormhole/faqs. [Online; accessed 03-March-2022]
[29]. Kelvinfichter’s post on twitter. https://twitter.com/kelvinfichter/status/1489041221947375616?s=21. [Online; accessed 03-March-2022]
[30]. Ethereum secure smart contract development guideline. crytic/building-secure-contracts: Guidelines and training material to write secure smart contracts. https://github.com/crytic/building-secure-contracts. [Online; accessed 03-March-2022]
[31]. Ethereum Smart Contract Best Practices. https://consensys.github.io/smart-contract-best-practices/about/. [Online; accessed 03-March-2022]
[32]. Anonymous, “Solidity documentation”, 2022. https://buildmedia.readthedocs.org/media/pdf/solidity/develop/solidity.pdf [Online; accessed 03-March-2022]
[33]. M. Alharby and A. van Moorsel, “Blockchain-based smart contracts: A systematic mapping study”, Fourth International Conference on Computer Science and Information Technology, 2017.
[34]. I. Wöhrer and U. Zdun, “Smart contracts: Security patterns in the Ethereum ecosystem and solidity”, In the 2018 International Workshop on Blockchain Oriented Software Engineering, 2018.
[35]. The bug which the DAO hacker exploited was not merely in the DAO itself, 2016. https://redd.it/4opjov [Online; accessed 03-March-2022]
[36]. Gernot Salzer, Monika Di Angelo, “A Survey of Tools for Analyzing Ethereum Smart Contracts”, 2019.
[37]. Mauro C. Argañaraz, Mario M. Berón, Maria J. Varanda Pereira, Pedro Rangel Henriques, “Detection of Vulnerabilities in Smart Contracts”, 2020.
[38]. Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, Martin Vechev, “Securify: Practical Security Analysis of Smart Contracts”, 2018.
[39]. Securify v2.0. https://github.com/eth-sri/securify2 [Online; accessed 05-March-2022]

[40]. ConsenSys/mythril: Security analysis tool for EVM bytecode. Supports smart contracts built for Ethereum, Hedera, Quorum, Vechain, Roostock, Tron and other EVM-compatible blockchains. https://github.com/ConsenSys/mythril [Online; accessed 05-March-2022]

[41]. Privacy Promises - Keystone. https://keyst.one/keystone-hardware-wallet-privacy-promises [Online; accessed 15-March-2022]

[42]. web3.js - Ethereum JavaScript API. https://web3js.readthedocs.io/en/v1.7.3/#web3-js-ethereum-javascript-api [Online; accessed 15-March-2022]

[43]. OpenZeppelin | About. https://openzeppelin.com/about/. [Online; accessed 15-March-2022]

[44]. Ganache | Overview - Truffle Suite. https://trufflesuite.com/docs/ganache/. [Online; accessed 15-March-2022]

[45]. Ethereum Development Standards. https://ethereum.org/en/developers/docs/standards/. [Online; accessed 15-March-2022]

Appendix 1 – Source codes

1.1 Checks-effects-interaction

```
pragma solidity ^0.8.0;

contract checksEffectsInteraction {
	bool public publicSaleEnded = false;

	function endPublicSale() public {
		// Check
		require(this.timestamp >= publicSaleEnd);
		require(!publicSaleEnded);
		// Alter the state
		publicSaleEnded = true;
		// External contract call
		externalContract.externalMethod();
	}
}
```

Figure 5: Pattern: Checks-effects-interaction

1.2 Mutex

```
pragma solidity ^0.8.0;

contract MutexPattern {
	bool private locked = false;
	modifier nonReentrant() {
		require(!locked, “No re-entrancy allowed”);
		locked = true;
		_;
		Locked = false;
	}
	function someMethod() nonReentrant external returns (uint256) {
		msg.sender.call()
		return 1;
	}
}
```

Figure 6: Pattern: Mutex

1.3 Balance limit

```
pragma solidity ^0.8.0;

import “@openzeppelin/contracts/access/Ownable.sol”

contract BalanceLimit is Ownable {
	uint256 public maxLimit = <LIMIT>;

	function setLimit(uint256 newLimit) external onlyOwner {
		maxLimit = newLimit;
	}

	modifier limitedPayable() {
		require(this.balance <= limit, “Limit has already been
	exceeded”);
		_;
	}

	function depositETH() public payable limitedPayable {
		// transfer some ETH
	}
}
```

Figure 7: Pattern: Balance limit

1.4 Emergency stop

```
pragma solidity ^0.8.0;
import “@openzeppelin/contracts/access/Ownable.sol”

contract EmergencyStop is Ownable {
	bool public paused = false;

	modifier whenPaused {
		require(!paused)
		_;
	}
	modifier whenNotPaused {
		require(paused)
		_;
	}
	function toggleContractPaused() public onlyOwner {
		paused = !paused;
	}
	function deposit() public payable whenNotPaused {
		// some code
	}
	function withdraw() public view whenPaused {
		// some code
	}
}
```

Figure 8: Pattern: Emergency stop

1.5 Rate limit pattern

```
pragma solidity ^0.8.0;

contract RateLimit {
	uint lastExecution = time.timestamp;

	modifier enabledEvery(uint t) {
		if (time.timestamp >= lastExecution) {
			lastExecution = time.timestamp + t;
			_;
		}
	}

	function f() public enabledEvery(1 minutes) {
		// some code
	}
}
```

Figure 9: Pattern: Rate limit pattern

1.6 Speed bump

```
pragma solidity ^0.8.0;

contract SpeedBump {
  struct Withdrawal {
    uint amount;
    uint requestedAt;
  }
  mapping (address => uint) private balances;
  mapping (address => Withdrawal) private withdrawals;
  uint constant WAIT_PERIOD = 7 days;
function deposit() public payable {
    if(!(withdrawals[msg.sender].amount > 0)) {
      balances[msg.sender] += msg.value;
    }
  }
function requestWithdrawal() public {
    if (balances[msg.sender] > 0) {
      uint amountToWithdraw = balances[msg.sender];
      balances[msg.sender] = 0;
      withdrawals[msg.sender] = Withdrawal({ amount: amountToWithdraw,
requestedAt: now });
    }
  }
function withdraw() public {
    if(withdrawals[msg.sender].amount > 0 && now >
withdrawals[msg.sender].requestedAt + WAIT_PERIOD) {
      uint amount = withdrawals[msg.sender].amount;
      withdrawals[msg.sender].amount = 0;
      msg.sender.transfer(amount);
     }
  }
}
```

Figure 10: Pattern: Speed bump

1.7 Ownable

```
pragma solidity ^0.8.0;

contract Ownable {
	address private _owner;

	constructor() {
		_setOwner(msg.sender);
	}

	function owner() public view virtual returns (address) {
		return _owner;
	}

	modifier onlyOwner() {
		require(owner() == _msgSender(), "Ownable: caller is not the
owner");
		_;
	}

	function transferOwnership(address newOwner) public onlyOwner {
		require(newOwner != address(0), "Ownable: new owner is the
	zero address");
		_setOwner(newOwner);
	}
	function _setOwner(address newOwner) private {
		address oldOwner = _owner;
		_owner = newOwner;
	}
}
```

Figure 11: Pattern: Ownable

1.8 Roles

```
library Roles {
	sturct Role {
		mapping (address => bool) bearer;
	}

	Function add(Role storage role, address account) internal {
		require(!has(role, account), “Roles: account already has a 
role”);
		role.bearer[account] = true;
	}

	Function remove(Role storage role, address account) internal {
		require(has(role, account), “Roles: account does not have a 
role”);
		role.bearer[account] = false;
	}

	Function has(Role storage role, address account) internal view 
returns (bool) {
		Require(account != address(0), “Roles: account is the zero 
address”);
		Return role.bearer[account];
	}
}
```

Figure 12: Pattern: Roles

1.9 Multiple ownership authorization

```
pragma solidity ^0.8.0;

contract Authorization {
	uint256 public nonce = 0;
	uint256 public threshold;
	mapping(address => bool) isOwner;
	address[] public ownersArr;

	constructor(uint256 _threshold, address[] _owners) {
		if(_owners.length > 10 || _threshold > _owners.length || 
_threshold == 0) {
			throw;
		}

		for (uint256 i=0; i < _owners.length; i++) {
			isOwner[_owners[i]] = true;
		}
		ownersArr = _owners;
		threshold = _threshold;
	}

	Function execute(uint8[] sigV, bytes32[] sigR, bytes32[] sigS, 
address destination, uint256 value, bytes data) {
		if(sigR.length != threshold) {
			throw;
		}

		If(sigR.length != sigS.length || sigR.length != sigV.length) 
		{
			throw;
		}

		bytes32 txHash = sha3(byte(0x19), byte(0), this, 
	destination, value, data, nonce);
		address lastAdd = address(0);

		for(uint256 i=0; i < threshold, i++) {
			address recovered = ecrecover(txHash, sigV[i], 
	sigR[i], sigS[i]);
			if(recovered <= lastAdd || !isOwner[recovered]) {
				throw;
			}
			lastAdd = recovered;
		}
```

Figure 13: Pattern: Authorization

Appendix 2 – Non-exclusive licence for reproduction and publication of a graduation thesis[1]

I Alikhan Sailekeyev

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis "Secure Ethereum Virtual Machine Compatible Smart Contracts Development Guideline for a Private Company", supervised by Valdo Praust
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

14.05.2022

1 The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.