University of Zurich[UZH]

A Security Audit of the Blockchain Signaling Systems Protocol

Yulia Brun
Zurich, Switzerland
Student ID: 16-700-536

Supervisor: Bruno Bastos Rodrigues, Muriel Figueredo Franco and Prof. Dr. Burkhard Stiller
Date of Submission: August 5, 2020

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Bachelor Thesis – Communication Systems Group, Prof. Dr. Burkhard Stiller

Bachelor Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Abstract

Ethereum Smart Contracts unterstützen eine Vielzahl an Funktionen und erlauben es, Universalapplikationen zu erstellen. Da sie jedoch von Menschen erstellet werden, sind sie fehleranfällig. Werden Schwachstellen ausgenutzt, kann dies riesige finanzielle Verluste nach sich ziehen, wie etwa bei DAO und Parity Wallet. Darüber hinaus verunmöglicht die Unveränderbarkeit der Blockchain, dass Fehler behoben werden können, nachdem der Contract implementiert wurde. Daher haben wir eine Sicherheitsanalyse des Blockchain Signaling Systems (BloSS), das für die kooperative Abwehr von Distributed Denial-of-Service (DDoS) Attacken entwickelt wurde, durchgeführt.

Das Ziel der vorliegenden Studie ist es, für Ethereum Smart Contracts typische Schwachstellen zu identifizieren, unterschiedliche Tools für automatisierte Sicherheitsaudits anzuwenden, einen Sicherheitsaudit von BloSS Contracts durchzuführen und schliesslich die Ergebnisse zu analysieren und zu vergleichen. Im Rahmen einer Literaturrecherche haben wir sechs generelle Schwachstellen für die Blockchain und 41 Schwachstellen für Ethereum und Solidity identifiziert. Des Weiteren wurden fünf Sicherheitstools benutzt und deren Resultate hinsichtlich der Klassifizierung von Schwachstellen und richtig und falsch positiven Ergebnissen untersucht. Ausserdem hat der Sicherheitsaudit zu 57 Funden geführt, und jedem Fund wurde ein Risikolevel gemäss NIST Guide for Conducting Risk Assessments zugeordnet. Schliesslich wurden die Resultate des Sicherheitsaudits mit den Funden der Sicherheitstools verglichen.

Obwohl Sicherheitsaudits von Smart Contracts ein schnell wachsender Forschungsbereich sind, mangelt es an Standartisierung. Zum Zeitpunkt der Niederschrift dieser Bachelorarbeit gibt es weder eine universell anerkannte Klassifizierung von Schwachstellen, noch eine Anleitung für die Durchführung manueller Sicherheitsanalysen von Smart Contracts. Eine bessere Standartisierung würde jedoch die Kommunikation und den Vergleich unterschiedlicher Studien vereinfachen und junge Spezialisten bei ihren ersten Audits unterstützen.

Ethereum smart contracts support a wide range of functionality and allow creating general-purpose applications. However, created by humans, they are error prone. Exploited vulnerabilities can result in huge financial losses like in the case of the DAO and Parity wallet. Besides, due to the immutability of blockchain it is impossible to fix bugs once a contract is deployed. For this reason, we conducted a security analysis of the Blockchain Signaling System (BloSS) designed for cooperative defence against Distributed Denial-of-Service (DDoS).

The current study aims to identify vulnerabilities typical for Ethereum smart contracts, apply various tools for automated security audit, perform a manual security audit of BloSS contracts and, finally, analyse and compare the findings. Based on a literature research, we identified six general blockchain vulnerabilities and 41 Ethereum and Solidity vulnerabilities. Moreover, we used five security tools and analysed their results regarding the vulnerabilities classification and true and false positives. The security audit delivered a list of 57 findings. To each of them, a risk level based on NIST Guide for Conducting Risk Assessments was assigned. Finally, the results of the security audit were compared with the security tools findings.

Although security audit of smart contracts is a fast growing field of study, it lacks standardisation. At the time of writing, there is no universally acknowledged classification of vulnerabilities or a guide for conducting manual security analysis of smart contracts. However, an increase in standardisation would make the communication and comparison of different studies easier and support young professionals in their first audits.

Acknowledgments

I would like to thank my supervisors Bruno Rodrigues, Muriel Franco and Prof. Dr. Burkhard Stiller from the Communication Systems Group at the University of Zurich.

In particular, I am deeply grateful to Bruno Rodrigues for his help. He spent many hours in meetings with me and was always able to give me good advice and support me.

Contents

Abstract **i**

Acknowledgments **iii**

1 Introduction **1**

2 Background **5**

2.1 Blockchain 5

2.1.1 Main Characteristics 5

2.1.2 Bitcoin & Ethereum 6

2.1.3 Smart Contracts 7

2.2 Security Frameworks 7

2.2.1 The Confidentiality-Integrity-Availability Triad 7

2.2.2 NIST Guide for Conducting Risk Assessments 8

3 Related Work **11**

3.1 Security Audit & Vulnerabilities in Smart Contracts 11

3.2 Tools for Automated Security Audit 12

4 Blockchain Signaling System **15**

5 Methodology **17**

5.1 General Approach 17

5.2 Tools for Automated Security Audit 18

5.3 Security Analysis of Smart Contracts 19

6 Findings **23**

6.1 Tools for Automated Security Audit 23

6.2 Security Analysis of Smart Contracts 25

7 Discussion **49**

8 Conclusion and Final Considerations **55**

Bibliography **56**

Abbreviations **63**

Glossary **65**

List of Figures **65**

List of Tables **67**

A Comparison of the Security Audit Findings with the Results of the Security Audit Tools **71**

B Classification and Analysis of the Security Audit Tools Findings **77**

Chapter 1

Introduction

Blockchain provides a trustworthy, decentralized, and publicly available data storage making it an interesting opportunity for organizations to increase business agility and reduce costs by removing intermediaries in distributed applications (*e.g.,* by involving multiple and initially non-trusted stakeholders) [1]. The disintermediation characteristic allows, for example, two or more parties to conduct an exchange upon an agreement without requiring the presence of a third party acting as an intermediary. Further, the ability to run program code, which is managed and executed in a blockchain, has extended the possibilities for the development of new application areas on the blockchain.

It is precisely the important characteristic to disintermediate trust which arouses interest in the use of blockchain in applications areas beyond FinTech (Finance and Technology). For example, in the context of a Distributed Denial-of-Service (DDoS) cooperative defense, blockchain capabilities could be leveraged for signaling attacks as a mitigation requests across a blockchain network, and serve as an immutable platform for the exchange of mitigation services defined in smart contracts of different peers [2]. A cooperative network defense has many benefits, by utilizing other organization's resources the burden of the protection can be shared, and defense capabilities can be extended through the different protection systems participating in the distributed defense.

While many benefits are provided through a cooperative DDoS defense without a central element coordinating the defense, it also poses many challenges. For example, the lack of trust to defend other peers can be a hindrance to the design of such system. However, a reputation and incentives schemes, such as proposed in [3] can be deployed to establish trust in an environment composed of peers which may often compete against each other. Incentives are necessary to cover CAPital EXpenditures (CAPEX) to set up communication infrastructures, including additional hardware and software acquisition costs; and OPErating EXpenditures (OPEX) are incurred as soon as a mitigation service is in use. Still, there is no automated way to build a consensus on the quality of a mitigation service provided by peers in response to a mitigation request [4]. In other words, malicious peers seeking incentives could accept a mitigation request and not deliver a mitigation service to obtain the reward. Conversely, a peer that request a mitigation service could deny the acknowledgment of a valid mitigation service denying the reward to the mitigator.

The main goal of the current bachelor's thesis is to conduct a security analysis of the cooperative protocol Blockchain Signaling System (BloSS) defined in [5]. As blockchain and smart contracts are still rather new technologies, many developers lack knowledge and experience in these areas [6]. This could be the reason why at least one vulnerability was found in 46% of all Ethereum smart contracts existing in 2016 [7]. The problem is that if an attacker exploits a smart contract's vulnerability, it may lead to huge financial damage like in the case of the Decentralised Autonomous Organisation (DAO) attack in 2016 with an approximate loss of 60 million USD or Parity wallet which lost 31 million USD in 2017 [8]. Besides, in contrast to traditional software, it is impossible to debug smart contracts once they are deployed. As soon as smart contracts are on the blockchain, they become locked for changes due to the immutability of blockchain. Thus, developers cannot fix a bug or improve a contract. In order to do so, they would have to kill this contract and deploy another one. For this reason, it is important to perform security an audit of smart contracts before the deployment [6].

In addition to conducting a security analysis of BloSS, this bachelor's thesis aims to provide an overview of the vulnerabilities typical for Ethereum smart contracts. Besides, our goals include using and comparing different tools for automated security audit, analysing and documenting the findings of the security analysis and the security tools results.

The current project consists of four major steps. First of all, we aim to gain a fundamental knowledge of blockchain, Ethereum, smart contracts and security frameworks such as Confidentiality-Integrity-Availability (CIA) triad and National Institute of Standards and Technology (NIST) Guide for Conducting Risk Assessments. This step also includes an extensive literature research on vulnerabilities in Ethereum smart contracts and tools for automated security audit. In the second step, we apply previously found security tools to smart contracts in BloSS in order to get the first impression of their security level. Then, a manual security audit of the smart contracts will be performed. In this step, we use CIA triad and a list of previously identified vulnerability types and apply NIST Guide for Conducting Risk Assessments in order to estimate the risk level of the findings. Finally, we analyse and compare the security tools findings and the results of the manual security audit.

The bachelor's thesis is organised as follows. The next chapter contains the background information about blockchain and security frameworks. Then, the related work on smart contracts vulnerabilities and security audit tools are presented. Chapter 4 provides a short introduction to BloSS and the signaling process. In the following chapter, we describe our methodology. The findings are presented in Chapter 6. In Discussion, we analyse the security tools findings regarding true positives and true negatives and compare them with the results of the security audit. Besides, we discuss advantages and disadvantages of different tools. In Chapter 8, we describe the difficulties we faced during this project and the limitations of the current study. The thesis contains also a list of abbreviations and a short glossary. Finally, there are two appendices. The first one contains a table matching the security tools findings to the security audit results. Appendix B consists of five tables. The first four present our analysis of the security tools findings with regard to vulnerabilities classification and identification of true positives. The last table contains links to Securify's reports. The screenshots of the security audit tools findings and their error messages can be found in our GitHub repository (`https://github.com/ytyuri/`

Bachelor-s-Thesis---Screenshots/tree/master).

Chapter 2

Background

2.1 Blockchain

Blockchain offers a significant change for a variety of industries where the disintermediation of trust makes sense. Through a decentralized and immutable data storage, it also enables the enforcement and verification of exchange assets — when supporting smart contracts — while changing existing areas by promoting the disintermediation of processes involving multiple stakeholders. By removing third parties, less operational costs and higher business agility are expected.

In general, a blockchain is a list of sequential blocks, which are linked to each other using block hashes. Blocks consist of two parts: a block header and a block body. The first one contains different information about the block including a timestamp, a block hash pointing to the previous block and a hash value for all transactions in the block. The transactions themselves are stored in the block body [9]. They are hashed in a Merkle tree, which is a binary tree where a non-leaf node's hash results from the hashes of its children. Due to this structure, any inconsistency in the Merkle tree would become visible in the blockchain [10].

Blockchains use digital signatures based on asymmetric cryptography for transaction validation. Every blockchain participant has a private and a public key. Private keys are used for signing transactions, while public keys allow their verification [9].

2.1.1 Main Characteristics

The key characteristics of a blockchain include decentralisation, transparency, data availability, immutability and data integrity. *Decentralisation* results from the distributed nature of the blockchain network. All participants (nodes) have the same copy of the ledger. This helps to increase *transparency* and ensure *data availability* [11]. Due to the hashes linking the blocks to each other, it is impossible to alter a transactional record once

it was added to a block and the block was committed to the blockchain. Thus, the transactions on the blockchain become *immutable* after a commit. Finally, the immutability of transactions guarantees the *data integrity* [12].

Nevertheless, blockchains can have a different level of decentralisation, immutability etc. depending on their type. Three blockchain types are usually distinguished: private, consortium and public blockchains [12, 9]. Both private and consortium blockchains restrict participation in consensus process are, therefore, called *permissioned*. In a *private* blockchain, only members of an organisation can take part in the consensus process. Therefore, it is the most centralised blockchain type. *Consortium* blockchains are more decentralised than private ones as their consensus process is controlled by a pre-selected group of individuals. Finally, in a *public* blockchain, any user can join the consensus process. Thus, they are *permissionless* and the most decentralised ones. Public blockchains also have often a high level of transparency as all the records are visible to any user. In contrast to them, private and consortium blockchains may implement read restrictions. Due to the limited number of participants in these blockchains, the transactions in them can be tampered. As a result, these blockchains are less immutable than public ones. The number of participants can also affect the blockchain's efficiency. For this reason, public blockchains are in general less efficient than private and consortium blockchains [9].

2.1.2 Bitcoin & Ethereum

The idea of a blockchain was first introduced in a white paper about Bitcoin [1] published on the Internet under the pseudonym of Satoshi Nakamoto in 2008. In contrast to previously proposed digital currencies (*e.g.,* Mint [13] in 1997), Bitcoin uses a peer-to-peer network and, therefore, does not require a trusted third party such as a bank [10].

Besides, Satoshi Nakamoto [1] addresses the problem of double-spending. He tries to prevent malicious users from issuing more than one transaction of the same coin by implementing Proof-of-Work consensus algorithm in Bitcoin. According to Proof-of-Work, users have to perform heavy computations proving that they are valid members of the blockchain's network before verifying a transaction [10]. Another popular blockchain implementing Proof-of-Work is Ethereum. It was introduced by Vitalik Buterin [14] in 2013. In contrast to Bitcoin, Ethereum is Turing-complete. In other words, it supports any kinds of calculations including loops [10]. Moreover, Ethereum's abstract layer allows users to create their own state transition functions, ownership rules and transaction formats, as well as to write smart contracts and decentralised applications [14].

As a blockchain, Ethereum is similar to Bitcoin. However, unlike blocks in Bitcoin, Ethereum blocks contain the most recent state, block number and its difficulty [14]. Besides, Ethereum uses another Proof-of-Work consensus algorithm which is called *Ethash*. This algorithm is memory-heavy and, for this reason, less suitable for mining with application-specific integrated circuits, which has become a big problem for Bitcoin [10, 15]. In the current bachelor's thesis, we will focus on Ethereum and its smart contracts.

2.1.3 Smart Contracts

The concept of a smart contract was known long before the blockchain was first introduced. In 1994 Nick Szabo [16] described a smart contact as "a computerized transaction protocol that executes the terms of a contract". According to him, smart contacts should satisfy common contract conditions, minimize exceptions and help to omit the trusted intermediary. Moreover, by proposing smart contracts, Nick Szabo aimed to lower fraud loss, arbitration and transactional and enforcement costs.

In blockchain context, a smart contract is a piece of code, which can be executed by miners automatically [9]. Smart contracts are heavily used in Ethereum where they are usually written in high-level Turing-complete programming languages such as Solidity. Smart contracts are then compiled down to a low-level stack-based language called *Ethereum Virtual Machine Code* or *EVM Code* [14]. This is the bytecode which is effectively executed by the Ethereum Virtual Machine [15].

2.2 Security Frameworks

In this section, we introduce the related security frameworks used in the bachelor's thesis.

2.2.1 The Confidentiality-Integrity-Availability Triad

The Confidentiality-Integrity-Availability triad (also known as the *CIA triad*) is one of the fundamental concepts of information security. As Figure 2.1 shows, it aggregates three major principles of information security: confidentiality, integrity and availability [17].

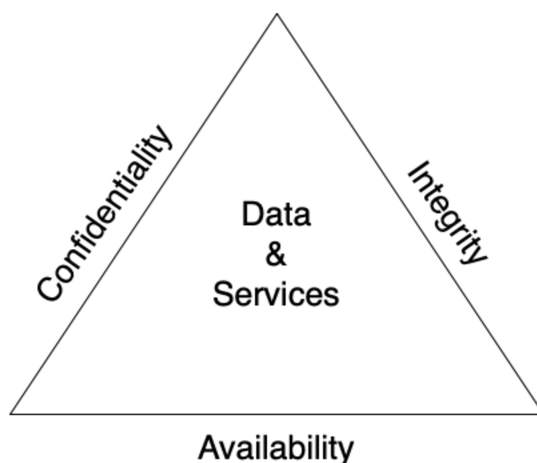


Figure 2.1: The CIA Triad. Adapted from [17]

In this framework, confidentiality stands for restricted access to particular information or functionality. Integrity implies that data is managed and updated correctly and, therefore, remains accurate over time. Finally, availability means that it is possible to access the data or functionality any time needed [17].

2.2.2 NIST Guide for Conducting Risk Assessments

The Guide for Conducting Risk Assessments [18] was developed by the National Institute of Standards and Technology (NIST) of the U.S. Department of Commerce, and published as a NIST Special Publication in 2012. Its target audience is risk management professionals. The framework provides guidelines for each stage of the risk assessment process including the preparation to the assessment, the risk assessment itself, communication of the results and maintenance of the risk assessment.

At the first stage, risk management professionals are suggested to identify the purpose of the risk assessment, define its scope, determine the constraints and assumptions and find necessary information sources. The second stage concerns the conduction of a risk assessment consisting of several steps. First of all, the potential threat sources need to be identified. Based on this information, potential threat events are determined. Then, different vulnerabilities and conditions that can affect the likelihood that threat events occur and have an adverse impact need to be inspected. In the next two steps, risk management professionals estimate the likelihood of the threat events and their potential negative impact. Both of these factors depend on the previously identified threat sources and vulnerabilities as well as on the implemented countermeasures. Finally, the risk of a threat event is determined as a combination of its likelihood and potential impact as demonstrated in Table 2.1 [18].

Table 2.1: Risk Level Assessment Scale. Adapted from [18]

Likelihood	**Level of Impact**				
	Very Low	**Low**	**Moderate**	**High**	**Very High**
Very High	Very Low	Low	Moderate	High	Very High
High	Very Low	Low	Moderate	High	Very High
Moderate	Very Low	Low	Moderate	Moderate	High
Low	Very Low	Low	Low	Low	Moderate
Very Low	Very Low	Very Low	Very Low	Low	Low

The resulted risk level indicates the degree to which the threat event is hazardous for the organisation. The following table describes the expected effect of the threat on the company's operations, assets etc. depending on the risk level [18].

Table 2.2: Expected Effect of Different Risk Levels. Adapted from [18]

Risk Level	**Expected Effect**
Very High	Multiple severe or catastrophic effects
High	Severe or catastrophic effect
Moderate	Serious effect
Low	Limited effect
Very Low	Negligible effect

In general, the NIST Guide for Conducting Risk Assessments supports three risk as-

sessment approaches: quantitative, qualitative and semi-quantitative. The first one uses methods and principles relying on numerical values and is especially useful for cost-benefit analysis. However, its results are often difficult to interpret which makes the communication with the decision-makers complicated. The second one is based on non-numerical categories or levels (*e.g.,* high, low) and is prone to subjectivity because the results often rely heavily on the expert's individual experiences. Finally, the last approach uses, for example, bins (*e.g.,* 0-10, 11-20), scales (*e.g.,* 1-10) and other representative numbers. It combines the benefits of both quantitative and qualitative approaches. The results are more objective than in the qualitative approach and can be compared with each other. Moreover, the decision-makers can easily interpret them [18].

Due to the lack of numerical values which could be used for the risk level determination during the risk assessment, the qualitative risk assessment approach with five levels (very low – low – moderate – high – very high) is employed in this and following sections.

Chapter 3

Related Work

3.1 Security Audit & Vulnerabilities in Smart Contracts

One of the first systematic reviews of Ethereum and Solidity vulnerabilities was published in 2017 by Atzei, Bartoletti, and Cimoli [19]. Overall, they identified and described 12 vulnerabilities dividing them into three categories: Solidity vulnerabilities (call to the unknown, gasless send, exception disorders, type casts, reentrancy and keeping secrets,), vulnerabilities related to Ethereum Virtual Machine (immutable bugs, Ether lost in transfer and stack size limit) and general blockchain vulnerabilities (unpredictable state, generating randomness and time constraints).

In the same year, another classification of vulnerabilities in Ethereum smart contracts was proposed by Alharby and van Moorsel [20]. However, the scope of their study included not only security vulnerabilities such as transaction-ordering dependency, timestamp dependency, mishandled exceptions, reentrancy, criminal smart contracts and the lack of trustworthy data feeds, but also codifying, privacy and performance issues.

Similar to Atzei et al. [19], Praitheeshan et al. [6] distinguish general blockchain vulnerabilities from the ones typical only for Ethereum and Solidity smart contracts. However, they put Ethereum and Solidity vulnerabilities together and added one more group of vulnerabilities – general software security issues. Overall, Praitheeshan et al. distinguish three groups of smart contract vulnerabilities. The first one consists of blockchain related vulnerabilities and includes immutability, sequential execution, complexity, transaction cost and human errors. The second one contains general software security issues (*e.g.,* buffer overflow, command injection, poor usability). Finally, the last group is a list of Ethereum and Solidity related vulnerabilities including reentrancy, transaction-ordering, timestamp dependency, exception handling, call stack limitation, integer overflow and underflow, unchecked and failed *send*, suicidal contracts, unsecured balance, use of *tx.origin*, unrestricted write and transfer, non-validated arguments, greedy and prodigal contracts and gas costly patterns.

The most extensive list of vulnerabilities in Ethereum smart contracts was created by Chen at al. [8] and contains overall 44 security issues (six of them are marked as already eliminated). The vulnerabilities are distinguished by their location in Ethereum's architecture (application layer, data layer, consensus layer, network layer or environment layer). Besides, Chen at al. divided the vulnerabilities into different categories depending on their cause. The resulted four major categories include vulnerabilities related to smart contract programming (*e.g.,* reentrancy, interger overflow and underflow, use of *tx.origin*), to Solidity language and toolchain (*e.g.,* Type casts), to Ethereum design and implementation (*e.g.,* timestamp dependency and generating randomness) and, finally, to human, usability and networking factors (*e.g.,* weak password, broken access control). At the time of writing, this is the most comprehensive overview of Ethereum smart contract vulnerabilities.

Another approach to the classification of vulnerabilities in Ethereum smart contracts was suggested by Dingman et al. [21]. They applied the NIST Bugs Framework [22] to a list of known Ethereum smart contacts vulnerabilities. The NIST Bugs Frameworks is based on the data from Common Weakness Enumeration [23], its clustering Software Fault Pattens, Semantic Templates and other sources and allows unambiguous classification of software weaknesses [22]. In order to map smart contract vulnerabilities to the NIST Bugs Framework, Dingman et al. analysed their cause, attributes and consequences. As a result, the study presents a master list of smart contract vulnerabilities with matching categories and classes from the NIST Bug Framework [21].

Finally, some studies focus on describing the most severe vulnerabilities instead of providing a systematic classification. For example, Luu et al. [7] discuss four vulnerabilities which can be used to manipulate smart contracts and gain profit by malicious actors. These vulnerabilities include transaction-ordering dependence, timestamp dependence, mishandled exceptions and reentrancy. In addition to these four security issues, Dika and Nowostawski [24] describe other severe vulnerabilities such as the use *tx.origin*, call stack depth limitation, external calls, unchecked *send*, DoS with unexpected revert, blockhash usage and gasless send.

3.2 Tools for Automated Security Audit

In general, three major types of automated security analysis can be distinguished: static analysis, dynamic analysis and formal verification. In accordance with the first approach, the programming code is scanned for vulnerable patterns without its execution. In contrast to this method, dynamic analysis is performed in a run-time. This approach simulates the behaviour of an attacker who is trying to find vulnerabilities by inserting malicious code and providing input to the code. Due to this technique, dynamic tools for automated security audit can identify vulnerabilities missed by static tools. Finally, formal verification methods rely on mathematical formal methods and theorems for the programming code validation and the prove of vulnerabilities [6].

All the previously described analysis types can deploy different strategies. For example, the static analysis can be performed on the bytecode using symbolic execution, control

flow graph construction, pattern recognition and decompilation or direct on the Solidity code by rule-based analysis and compilation. The dynamic analysis always executes the bytecode. The possible strategies include the run-time execution trace, transaction graph construction, symbolic analysis and true positives and false positives validation. Besides, the formal verification analyses the bytecode with the help of theorem provers and program logics construction and the Solidity code by translating it to a formal language [6].

The first tool for automated security analysis of smart contracts was presented in the paper by Luu et al. [7] in 2016 and is called Oyente. It performs static analysis and deploys a symbolic execution strategy. Symbolic execution (also called *abstract interpretation*) was introduced by Cousot and Cousot [25] in 1977. This strategy regards variables as symbolic expressions and checks if path conditions are satisfiable. As a result, Oyente is able to detect four types of vulnerabilities: transaction ordering, timestamp dependency, mishandled exceptions and reentrancy. When the tool was run on the 19'366 existing at that time Ethereum smart contracts, at least one vulnerability was found in 8'833 contracts which is about 46% of the total number [7].

Another tool using the symbolic execution is Mythril. It was developed by ConsenSys and presented by Bernhard Mueller [26] at the HITB Security Conference in Amsterdam in 2018. Mythril supports a security analysis of smart contracts not only in Ethereum, but also in Tron, Quorum, Vechain, Roostock, Hedera, etc. [27]. The tool scans the bytecode and is able to detect a wide range of vulnerabilities including write to arbitrary storage location, arbitrary jump with a variable of function type, delegate call, weak randomness, deprecated opcodes, unprotected Ether withdrawal, exception handling, reentrancy, integer overflow or underflow, DoS with failed call, suicidal contracts and unchecked return value. Unlike Oyente, Mythril does not have a Web Graphical User Interface (Web GUI) and can be only run on the command line [28].

However, there is also a professional version of Mythril called MythX. In contrast to Mythril, it is not free of charge. Besides, it covers a wider range of vulnerabilities than Mythril [27]. The list of 37 supported security issues can be found in the Smart Contract Weakness Classification Repository [29] created and maintained by MythX team. Moreover, MythX can be integrated directly into developer tools such as Remix and Truffle which allows to perform security analysis continuously during the whole lifecycle of the project [30].

Another tool for automated security audit of Ethereum smart contracts combines symbolic execution with formal verification [31]. Securify was developed by ETH Zurich and its start-up ChainSecurity [32]. Its first version has a Web GUI [32] and detects 18 different vulnerabilities including reentrancy, transaction-ordering dependency, exception handling and arguments validation [31]. However, in January 2020, a second version of Securify [33] was announced [34]. At the time of writing, it supports 38 vulnerabilities and is available only on the command line [33]. The new list of vulnerabilities is primarily based on the Smart Contract Weakness Classification Register [29] maintained by MythX team and ConsenSys. Besides, Security 2.0 analyses Solidity code and not bytecode as the previous version. Moreover, according to the developers, it is more precise and scalable than the original tool [34].

Automated formal verification is also offered by a built-in Solidity static analysis tool in Remix IDE [24]. At the time of writing, its latest version (0.10.1) was able to identify seven security issues, five gas-related issues, one ERC20 issue and eight miscellaneous vulnerabilities. Thus, the tool detects overall 21 different vulnerabilities [35].

In contrast to previously discussed tools, MAIAN [36] performs dynamic analysis. It deploys symbolic analysis and concrete validation of true and false positives. Instead of supporting a wide range of security issues, MAIAN focuses on three vulnerabilities: prodigal, greedy and suicidal contracts [37].

Another dynamic tool for automated security audit of smart contracts was developed by Trail of Bits and introduced by Mossberg at al. [38] in 2019. Manticore [39] relies on symbolic analysis of bytecode. It does not have a Web GUI but can be run from the command line and with Python Application Programming Interface (API) [39]. According to the information on the command line during the execution (see Figure X), Manticore detects 12 vulnerabilities including integer overflow, reentrancy, delegatecall and suicidal contracts.

Finally, some frameworks such as F* Framework [40], formalisation with Isabelle/HOL proof assistant [41] and FEther [42] implement formal verification analysis. These frameworks do not search for vulnerabilities as previously described tools but define correctness and safety properties for smart contracts and, then, prove them. Besides, the frameworks are semi-automated which means that a lot of manual work is required for their set-up [6]. For this reason, we decided not to discuss them in detail in the current bachelor's thesis and to focus on static and dynamic tools for automated security audit of smart contracts. The overview of these tools is presented in Table 3.1.

Table 3.1: Comparison of Tools for Automated Security Audit

	Oyente	**Mythril**	**MythX**	**Securify**	**Securify 2.0**	**Remix**	**MAIAN**	**Manticore**
Analysis Type	Static	Static	Static	Static	Static	Static	Dynamic	Dynamic
Strategy	Symbolic execution	Symbolic execution	Symbolic execution	Formal verification + Symbolic execution	Formal verification + Symbolic execution	Formal verification	Symbolic analysis + Concrete validation of true/false positives	Symbolic analysis
Number of Vulnerabilities in Scope	4	12	37	18	38	21	3	12
Command Line Interface	✓	✓	✓		✓	✓	✓	✓
Web Graphical User Interface	✓		✓	✓		✓	✓	
Free of Charge Usage	✓	✓		✓	✓	✓	✓	✓

Chapter 4

Blockchain Signaling System

The current bachelor's thesis aims to perform a security analysis of the Blockchain Signaling System (BloSS) created by Rodrigues et al. [2] in 2017 and then improved and extended by Andreas Gruhler et al. [3], Dominik Bünzli [43] and Spasen Trendafilov [5]. The main goal of BloSS is to enable collaborative defence against DDoS attacks. Its latest version contains overall four smart contracts: Protocol, Register, Migrations and Enums. The first smart contract is the main one in BloSS. It distributes the incentives between the participants, evaluates their reputation and manages the signaling process in general. The Register contract was added by Trendafilov [5]. It stores the information about Protocol contracts, targets and mitigators [5]. Finally, the Migrations contract updates the data about migrations, and Enums contains all possible states of the signaling process (Request, Approve etc.) and participants' ratings (Dissatisfied, Positive etc.). In the following, we provide a simplified description of the signaling process.

The process is initialised when a target which is a victim of a DDoS attack asks a mitigator for defence. By raising this request, the target changes the process state from Request to Approve. The initialisation request contains the information about the network and terms of the deal (the minimal reward, the deadline interval for prove generation etc.) [5].

In the second process step, the chosen mitigator considers these conditions and decides if the request should be accepted. In case of the negative answer from the mitigator, the state is changed to Abort. The target can select another mitigator or change the terms of the deal and ask the same mitigator again [5].

If the mitigator accepts the target's conditions, the state is updated to Funding. As soon as the target sends at least the minimal reward defined by the initialisation to the Protocol, the state is changed to uploadProof, and the funds get locked by the Protocol. The mitigator needs to upload the proof of work which was done in order to mitigate the attack. If the deadline interval for uploading the proof is missed, the mitigator gets a reputation of a lazy network participant and the state is switched to Abort. Otherwise, the process of rating estimation of the target and the mitigator begins. Unless both of them are dissatisfied, the signaling process' new state is Complete. As a result, the Protocol releases the blocked funds and transfers them as a reward to the mitigator or

as compensation to the target depending on whether the deadline was met and on the estimated rating of the participants [5].

However, if both target and mitigator are dissatisfied, the state is changed from uploadProof to Escalate. In this scenario, the funds remain locked by the Protocol, and the target and mitigator have to find an agreement on the payment manually [5].

By using this reputation estimation approach together with the incentive mechanism, BloSS aims to discourage targets from free-riding and mitigators from false-reporting. In a long-term scenario, dishonest behaviour of a participant results in poor reputation [3].

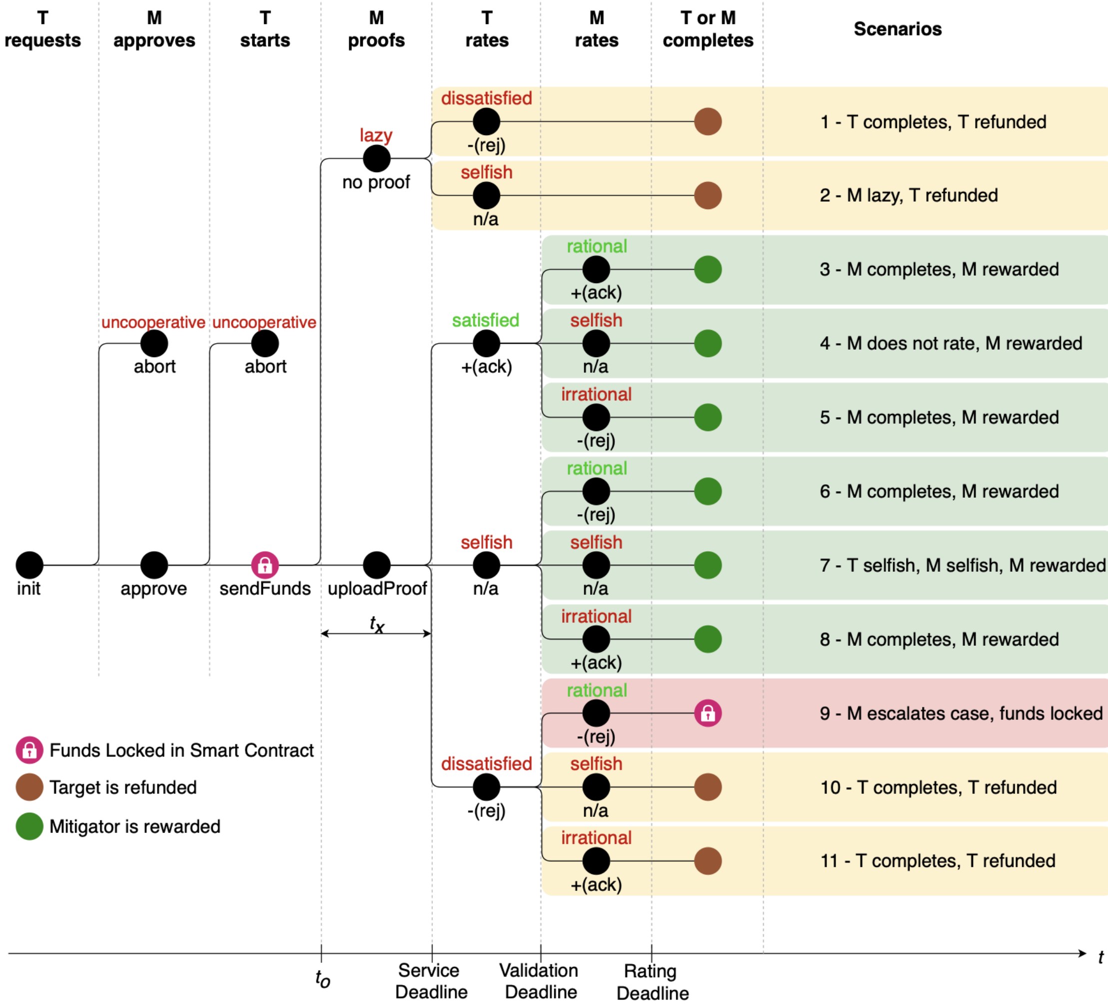


Figure 4.1: Signaling Process. From [5]

Chapter 5

Methodology

5.1 General Approach

The current project consisted of four major steps. First of all, we wanted to learn about the functionality and behaviour of BloSS. For this reason, we deployed the smart contracts on the Rinkeby testnet in two different ways (with Ganache and with Geth and full node on the command line) and interacted with them in Remix. Thus, we were able to gain a deep understanding of the functionality of BloSS.

In the same step, we performed a literature research on the theoretical background of blockchain, Ethereum and smart contracts, security frameworks such as NIST Guide for Conducting Risk Assessments and the CIA triad as well as on vulnerability types in Ethereum smart contracts and tools for automated security audit. As a result, we came up with two lists of common vulnerabilities based on three most recent studies ([8, 24, 6]). The first list comprises six vulnerabilities related to blockchain in general: immutability, transparency, sequential execution, complexity, transaction cost and human errors. The second one lists 41 vulnerabilities typical for Ethereum and Solidity smart contracts: reentrancy, transaction ordering, block timestamp dependency, blockhash usage, exception handling, call stack depth limitation, integer overflow and underflow, unchecked and failed send, destroyable and suicidal contracts, unsecured balance, use of origin, unrestricted write, unrestricted transfer, non-validated arguments, greedy contracts, prodigal contracts, overspent gas, gasless send, external calls, DoS with unexpected revert, DoS with unbounded operations, DoS with block stuffing, delegatecall injection, Ether lost to orphan address, manipulated balance, outdated compiler version, upgradeable contracts, erroneous visibility, secrecy failure, insufficient signature information, type casts, short address, under-priced opcodes, generating randomness, outsourceable puzzle, 51% hashrate, fixed consensus termination, rewards for uncle blocks, unlimited nodes creation, public peer selection and sole block synchronization.

In the second step, we used five tools for security audit of Ethereum smart contracts in order to get the first impression about the number of vulnerabilities in BloSS. More information about this step can be found in Section 5.2.

Then, we performed the security audit of BloSS. The smart contracts were reviewed method by method, as we were looking for vulnerabilities identified in the first step. Besides, we used the NIST Guide for Conducting Risk Assessments [18] to determine the risk level of each finding based on its likelihood and impact. Finally, we described possible measures which can help to eliminate or reduce the risk of the found vulnerabilities.

In the last step, we analysed and validated security tools findings and applied the list of 41 identified vulnerabilities for their classification. Besides, we compared these findings with the results of the security audit.

In the next subsections, we discuss the second and the third steps in more detail.

5.2 Tools for Automated Security Audit

Initially, we planned to use all free of charge security audit tools described in Section 3.2: Oyente, Mythril, Securify, Securify 2.0, Remix, MAIAN and Manticore. However, due to outdated dependencies in Oyente and MAIAN, we were not able to test them on BloSS. According to error messages and reported issues in GitHub repository [44], the latest Solidity version supported by Oyente is 0.4.19, while most of the contracts in BloSS require 0.5.8. Oyente's Web GUI [45] has even older dependencies. The last supported Solidity version is 0.4.17.

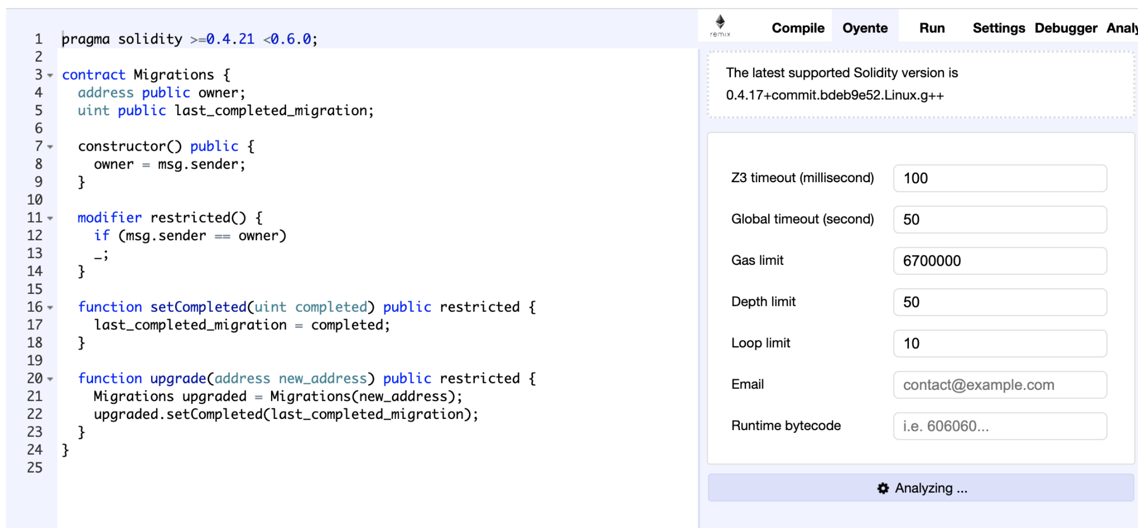


Figure 5.1: Oyente's Web GUI

There is a similar problem with MAIAN. It has a dependency on an outdated Web3 version. Changing the version does not solve the problem because of numerous dependency issues resulting from it. This issue was reported in MAIAN's GitHub repository [36] in 2018 but is still unsolved.

As a result, we excluded these tools from our selection. The findings of the remaining five tools are presented in Section 6.1.

It is also important to note that Securify and Securify 2.0 do not support the import of contracts. Therefore, we had to adopt the code of Protocol and Register for them by out-commenting the import statement for Enums and inserting the code of Enums at the end of the contract.

5.3 Security Analysis of Smart Contracts

The security audit of BloSS smart contracts consisted of three steps. Firstly, BloSS was reviewed regarding six general blockchain vulnerabilities. Then, we applied CIA triad to it. Finally, the smart contracts were analysed method by method with regard to 41 Ethereum and Solidity vulnerabilities identified before. In the following, we describe these vulnerabilities and define criteria used for their detection.

Reentrancy. Reentrancy occurs if one contract hands over the control to another contract, the second contract can call back into the first one several times before the first initiated interaction is completed [24]. There are two types of reentrancy:

- Single function reentrancy [46]:
    1. A *call*, *send* or *transfer* function which can hand over a control to an external contract is executed.
    2. The external contract has a fallback function.
    3. After the *call*, *send* or *transfer* function was executed, the state is updated.
- Cross-function reentrancy [46]: similar to single function reentrancy, happens when two different functions or contracts share the same state.

Transaction ordering. The state of a contract in which a transaction is executed depends on the transaction order determined by the miners of the block and cannot be predicted reliably [6].

Block timestamp dependency. Users can generate block timestamps which differ up to 900 seconds from other users' block timestamps [6]. Often functions rely on the starting time (*StartTime*), current time (*now*) and ending time (*EndTime*), which depend on *block.timestamp* [6].

Blockhash usage. Similar to block timestamp dependency. Malicious users can manipulate the outcome by changing the blockhash. Example: Using *blockhash* for generating randomness [24].

Exception handling. No proper exception handling. It is impossible to check the return value after a function call [6].

Call stack depth limitation. The call stack depth limit is hard-coded to 1024 frames. Every time a *call* or *send* function calls another contract, the call stack depth is increased by one [8].

Integer overflow and underflow. The result of an arithmetic operation is outside of the range of a Solidity data type [8].

Unchecked and failed *send*. The *send* function may fail if the gas limit is exceeded or if there is not enough Ethers on the balance. However, the function does not have a built-in error handling [6].

Destroyable and suicidal contracts. A smart contract that can be terminated by an anonymous *suicide* or *kill* function. This function is usually executed by the owner in case of an attack or an emergency situation [6].

Unsecured balance. The balance of a smart contract is unsecured. Possible reasons include improper access control for *balance* and *constructor* and updating the balance after sending the money [6].

Use of *tx.origin*. *tx.origin* returns the account address initiating the transaction [6].

Unrestricted write. A write-operation to the storage which does not have any restricting conditions [6].

Unrestricted transfer. By default, the *call* function allows to transfer money between any users and smart contracts [6].

Non-validated arguments. The arguments which are not checked before passing to a method [6].

Greedy contracts. If an external library contract is terminated, the contracts calling this library become greedy as they cannot access the library and transfer the funds anymore [6]. Besides, some contracts can receive Ethers but lack the instructions for sending them out (*e.g.*, *send*, *transfer*, *call*), or the instructions are unreachable [37].

Prodigal contracts. The sending function can be called by any user and can be used to send funds to any address chosen by the sender [6].

Overspent gas. Many patterns in Solidity smart contracts are very expensive in terms of gas required to spend for their execution [6]. Gas-costly programming patterns include: dead code, opaque predicate, expensive operations in a loop, constant outcome of a loop, loop fusion, repeated computations in a loop and comparison with unilateral outcome in a loop [47].

Gasless send. A transaction fails because not enough gas is provided. Example: an expensive function which requires a lot of gas to execute [24].

External calls. A call of an external contract. Pushing data to an external contract is in general more dangerous than pulling data [24].

DoS with unexpected revert. The vulnerability occurs when a conditional statement *if*, *for* or *while* depends on an external call [24].

DoS with unbounded operations. Due to the improper programming of unbounded operations (*e.g.,* a loop over a large array), the amount of gas required for contract execution may exceed the gas limit of the block [8].

DoS with block stuffing. Because of the greedy mining incentive mechanism, an attacker can offer a high *gasPrice* and, thus, motivate miners to prioritize his or her transactions over others [8].

***Delegatecall* injection**. Ethereum allows to embed a callee's bytecode into the caller contract using the *delegatecall* function. As a result, the state variables in the caller contract can be modified by the bytecode of the callee contract [8].

Ether lost to orphan address. When money is transferred, Ethereum only checks if the recipient's address is no longer than 160 bit but does not check if the address exists. If money is sent to a non-existing (orphan) address, Ethereum registers the address automatically. However, the address does not have any associated user or contract account. Thus, it is impossible to withdraw the transferred money [8].

Manipulated balance. The vulnerability occurs if the contract's control-flow relies on *this.balance* or *address(this).balance*, as they can be manipulated by an attacker [8].

Outdated compiler version. An outdated compiler may contain unfixed bugs [8].

Upgradeable contracts. In order to solve the problem of immutability in blockchain, developers can split smart contracts into two parts (a *proxy contract* which remains immutable when added to blockchain and a *logic contract* which can be updated by the developer) or use a *registry contract* for recording the updates [8].

Erroneous visibility. Functions which should not be called from external contracts are sometimes erroneously marked as *public* or *external*. As a result, they can be called directly by attackers [8].

Secrecy failure. Due to transparency of the blockchain, marking functions and variables as *private* does not guarantee data secrecy [8].

Insufficient signature information. Instead of using multiple transactions, a user sends money to multiple recipients with a proxy contract. The user can send digitally signed messages off-chain to recipients letting them withdraw the money. The proxy contract validates the digital signature to check if the transaction is approved. If the digital signature does not contain due information (*e.g.,* nonce and proxy contact address), a recipient can use the signed message several times and withdraw additional funds [8].

Type casts. When a function in an external contract is called using an address argument, the Solidity compiler checks if the function is declared in the contract but does not check if the address argument conforms to the contract's address. If there is another contract with the same declaration and a function named as in the first contract, the function in the wrong contract may be executed by mistake [8].

Short address. In contract-invocation transactions such as *transfer*, the selector and arguments are automatically encoded. The first four bytes are reserved for the callee function and the rest stands for arguments in chunks of 32 bytes. That means that if the last byte of one argument is missed, two hexadecimal zeros are added to the end of the last argument [8].

Under-priced opcodes. Some contacts contain many opcodes which have a low gas price but consume a lot of computing resources [8]. For example, *balance*, *extcodecopy*, *extcodesize*, *sload* and *suicide* are considered under-priced [48].

Generating randomness. A seed such as *block.number*, *block.timestamp*, *block.difficulty* and *blockhash* used for generating randomness can be manipulated by miners [8].

Outsourceable puzzle. Ethereum's Proof-of-Work puzzle makes a solution only partially sequential. Therefore, it is possible to divide a Proof-of-Work task into several parts and outsource them [8].

51% hashrate. Due to the Proof-of-Work consensus mechanism, attackers can take over the blockchain if they have at least 51% of the mining power [8].

Fixed consensus termination. Ethereums's consensus protocol uses deterministic termination to achieve a probabilistic agreement. In other words, if a block is followed by a fixed number of blocks n, it will most likely remain on the blockchain. When all the transactions in the block are commited and the next n blocks are added to the blockchain, the consensus for the block is terminated. However, in the reality, the probability of agreement can be affected by external factors such as a communication delay as, in an asynchronous network, a deterministic protocol cannot guarantee agreement, termination and validity at the same time [8].

Rewards for uncle blocks. A stale block referenced by a regular block is called an *uncle* block. In Ethereum not only regular blocks on the main chain, but also uncle blocks are rewarded [8].

Unlimited nodes creation. The node generation is weakly restricted. Malicious users can create an unlimited number if nodes (even with the same IP address) and use them to monopolize the connections to the victim's nodes [8].

Public peer selection. When a node is trying to locate a target, it queries 16 nodes in the bucket which are close to the target and asks each of them for 16 closest to the target neighbours. This process iterates until the target is identified. During this process the IDs of different nodes are provided to the querying node. The mapping of a node ID to the buckets in the routing table is public and can be exploited by attackers [8].

Sole block synchronization. A node may miss the synchronization with another one. If the second node is malicious, it can delay the synchronization on purpose. As it is only possible to synchronize with one node at a time in Ethereum, the first node becomes stalled and has to reject any subsequent blocks [8].

Chapter 6

Findings

6.1 Tools for Automated Security Audit

As a part of the current bachelor's thesis, we used five tools for automated security audit of Ethereum smart contracts. Table 6.1 demonstrates the number of their findings for different smart contracts in BloSS.

Table 6.1: Number of Findings per Security Audit Tool

	Mythril	**Securify**	**Securify 2.0**	**Remix**	**Manticore**
Enums	0	0	0	0	0
Migrations	1	2	8	1	0
Register	2	5	9	5	N/A
Protocol	3	18	N/A	64	N/A
Total	**6**	**25**	**17**	**70**	**0**

Protocol provides the main functionality of BloSS and is, therefore, the most complex smart contract among those evaluated. This explains why the tools found the highest number of vulnerabilities and bugs there. However, two tools were not able to perform the analysis of Protocol. Manticore could not analyse Protocol and Register contracts and broke with an error message about bus error 10 (see the error messages in the screenshot for Register and the the screenshot for Protocol in our GitHub repository). Searching in the Internet for similar behaviour produced no successful results. However, the reason for this failure could be that we run the tools on MacOS, while Manticore was developed for Linux and has only experimental support for MacOS [39].

Securify 2.0 successfully performed the analysis of Register contract after we added Enums contract directly to the code, as described in Section 5.2. However, even after performing the same operation on Protocol, we have got an error related to Souffle logic programming language, which is required for the execution (see the error message in the screenshot stored in our GitHub repository). At the time of writing, it is a known but not yet solved

problem in Securify 2.0. There are several open issues in the GitHub repository describing similar behaviour [33].

Thus, Manticore and Securify 2.0 identified less vulnerabilities than they probably would if they were able to analyse all four contracts. As a result, Remix has the biggest number of findings, and the original version of Securify is on the second place.

In the next step, we categorised all these findings by vulnerability types described in 5.3. Table 6.2 presents an overview of the findings. It is important to note that our definition

Table 6.2: Classification of the Security Audit Tools Findings

	Mythril	**Securify**	**Securify 2.0**	**Remix**	**Manticore**
Reentrancy		2			
Transaction ordering		2			
Block timestamp dependency	1			7	
Exception handling			1	34	
Unrestricted Write	2	14	3		
Non-validated arguments		1	4		
Greedy contract		1			
Overspent gas				25	
External call	3	3			
Erroneous visibility			6		
Division		2			
Uninitialised state variable			1		
Solidity naming convention violation			1		
Complex Solidity version pragma statement			1		
Variables with similar names				1	
Bytes and string length				1	
Tool's internal error during the audit				2	
Total	**6**	**25**	**17**	**70**	**0**

of a vulnerability type does not always completely match the one implemented in the tools. For example, Mythril identifies two external call vulnerabilities in the line 42 of the following snippet:

```
40     reg = Register(RegisterAddress);
41     // cast mitigator address from address to payable address
42     Mitigator = address(uint160(reg.getMitigator(address(this), _name)))
           ;
```

As per Mythril's findings report (see the screenshot for the first finding and the screenshot for second one in our GitHub repository), the first vulnerability occurs because the output of the external call is read. The second is caused by writing this output to a variable. However, according to the definition of external call in Section 5.3, we consider them to refer to the same external call and, therefore, identify only one external call vulnerability in this piece of code.

Besides, the last seven vulnerabilities in the table do not match any of vulnerability types defined in Section 5.3. Securify allows to check if a contract used division in any calculations. Its second version examines state variables, the compliance with Solidity naming convention and Solidity compiler's version statements. Finally, Remix reviews variable names and the usage of bytes and string length. Moreover, there are two messages about an internal error included into the security report for Protocol contract.

Finally, Table 6.2 contains all the findings including false positives. An overview of the number of false positives and true positives per tool is presented in Discussion. The detailed analysis of the findings can be found in Appendix B, while the screenshots of the actual reports are stored in our GitHub repository.

6.2 Security Analysis of Smart Contracts

First of all, we analysed BloSS in terms of general blockchain vulnerabilities. Due to irreversible nature of blockchain's transactions, deployed smart contracts and transactions become immutable after deployment. This property brings both advantages and disadvantages. On one side, hackers cannot modify contracts. On the other side, developers themselves are not able to change contracts if they are already deployed. Thus, in order to fix a bug or improve a contract, they would have to terminate it and create a new one. For this reason, testing and security audit play a crucial role in the development of smart contracts [6]. As this is also relevant for BloSS, we consider it highly important to perform a security analysis of its smart contracts before deploying them on the main network.

Another important blockchain property is sequential execution. The order of smart contracts execution is determined by a consensus mechanism and is the same for all users. As a result, only a limited number of contracts can be executed per second. This leads to a performance bottleneck and allows attackers to stall the network by using a contract which would take a lot of time to execute. Besides, this property makes blockchain solutions unscalable for a large number of transactions per second [6]. In case of BloSS, sequential execution can cause a delay when a proof of work is uploaded which could lead to a missed deadline. Thus, it can potentially influence the result of the signaling process and affect the rating of the participants. However, as BloSS was created for use in a consortium, with a limited number of participants and transactions, we estimate a limited effect of this vulnerability.

Complexity and human errors are other typical problems for smart contracts. The technology is still relatively new, and many developers do not have sufficient knowledge and

experience in this area yet. As a result, developing smart contracts is error prone [6]. These issues are relevant also in case of BloSS, which encourages us to perform the security audit of its contracts.

Another general blockchain issue is that a fee needs to be paid for every transaction execution on a blockchain [6]. This can potentially result in high transaction costs for users and developers. In BloSS, the target has to bear relatively high costs for transaction execution and contract deployment in comparison to mitigators [5]. However, if the contracts are only implemented and used on a test network, participants do not have to pay real-world money for the transaction execution.

Finally, in the context of the CIA triad, blockchain ensures two principles of information security by design: integrity and availability [49]. However, the transparency resulted from storing a copy of the whole ledger by every user [11] leads to lack of confidentiality [49]. Due to consortium network, BloSS is able to limit the access to the data [2]. Nevertheless, storing the unencrypted data about participants in Register is highly unsafe [5].

Then, we analysed the smart contracts regarding the Solidity and Ethereum vulnerabilities discussed in Section 5.3. The majority of the vulnerabilities were not found in the audited smart contracts. For instance, there are no dependencies on *blockhash*, *tx.origin* or attempts to generate randomness, which means that blockhash, use of *tx.origin* and generating randomness do not affect the contracts. Due to multiple state checks (*e.g.*, requirement that the current state is *Enums.State.APPROVE* in line 73 of the Protocol contract), contracts are also resistant against transaction ordering.

Table 6.3 contains an overview of security issues related to Solidity and Ethereum vulnerabilities, and provides a number of findings for each.

Table 6.3: Ethereum and Solidity Vulnerabilities

Vulnerability	**Resulted Security Issues**	**Number of Findings**
Reentrancy	The external contract can retrieve multiple refunds and empty the balance of the contract [24].	1
Transaction ordering	Front running: users try to execute their transaction first by offering higher gas [8].	0
Block timestamp dependency	Malicious users can manipulate the control flow by changing block timestamps [6]. It can be dangerous if critical components of a contract depend on block timestamps [24].	7
Blockhash usage	It can be dangerous if critical components of a contract depend on the blockhash [24].	0
Exception handling	Developers cannot handle errors properly which may result in vulnerable contracts [6].	0

Table 6.3: Ethereum and Solidity Vulnerabilities

Vulnerability	**Resulted Security Issues**	**Number of Findings**
Call stack depth limitation	Malicious users can make a contract call itself 1023 times. Then, they call a victim's contract and reach the depth limit of the EVM's stack. As a result, all further external calls by the victim's contracts fail [8].	0
Integer overflow and underflow	If a value is outside the range of the data type, its value is reset to 0 or to the highest possible value in the range. This vulnerability can be used by malicious actors to manipulate balances and steal Ethers [6].	1
Unchecked and failed *send*	If no error handling is implemented by the developer, the balance of the sender will be updated, although the Ethers were not sent to the receiver [6].	0
Destroyable and suicidal contracts	If there are no restrictions regarding the users who can execute the suicidal function, any user or smart contract can destroy the suicidal contract irreversibly [6].	0
Unsecured balance	The balance is exposed to possible attacks [6].	0
Use of *tx.origin*	If *tx.origin* is used for authorization, it can be easily compromised by phishing [8].	0
Unrestricted write	Malicious users can use this vulnerability to exploit the contract. For example, they can assign the ownership to themselves [6].	4
Unrestricted transfer	If no restrictions are implemented, any user can invoke the *call* function which makes the contract vulnerable [6].	0
Non-validated arguments	Passing non-validated arguments can result in malicious actions during the method execution [6].	10
Greedy contracts	Greedy smart contracts freeze their balance [6].	0

Table 6.3: Ethereum and Solidity Vulnerabilities

Vulnerability	**Resulted Security Issues**	**Number of Findings**
Prodigal contracts	The smart contracts may send funds to unknown users [6].	0
Overspent gas	Overpriced patterns result in poor usability and wasted Ethers [6].	0
Gasless send	If there is no proper error message, it may be difficult to identify the reason for the transaction fail [24].	0
External calls	Malicious code may be executed in the external contract [24].	1
DoS with unexpected revert	There may be a failure in the external call. Moreover, a callee may revert an operation intentionally. As a result, the transaction fails [8].	5
DoS with unbounded operations	The gas required for contact execution exceeds the gas limit of the block, and the contract execution fails [8].	0
DoS with block stuffing	Only the attacker's transactions are added to new blocks [8].	N/A
Delegatecall injection	A callee contract can modify the state variables of the caller contract [8].	0
Ether lost to orphan address	The transferred money is lost [8].	0
Manipulated balance	A malicious user can manipulate the outcome by changing the balance and, as a result, obtain the money [8].	0
Outdated compiler version	The compiled smart contact may be vulnerable [8].	N/A
Upgradeable contracts	A malicious developer can create a malicious logic contract which can interact with the immutable proxy contract [8].	0
Erroneous visibility	An attacker can get unauthorized access to a function [8].	29

Table 6.3: Ethereum and Solidity Vulnerabilities

Vulnerability	Resulted Security Issues	Number of Findings
Secrecy failure	The value of state and private variables can be extracted from transactional data and used by attackers for malicious activities [8].	N/A
Insufficient signature information	A malicious recipient can use a signed message multiple times and, thus, withdraw additional funds [8].	N/A
Type casts	Attackers can execute their contracts instead of the right ones [8].	0
Short address	A function can be called with wrong arguments (*e.g.,* wrong address and wrong amount of funds) [8].	0
Under-priced opcodes	A lot of computing resources are wasted by contract execution. This vulnerability can be exploited in DoS attacks [8].	0
Generating randomness	The outcome can be manipulated by malicious miners [8].	0
Outsourceable puzzle	Malicious miners can outsource Proof-of-Work subtasks and, therefore, benefit from parallel computing [8].	N/A
51% hashrate	The attackers with a majority of mining power can reverse transactions and perform double-spending [8].	N/A
Fixed consensus termination	The probability of agreement can be affected by different factors [8].	N/A
Rewards for uncle blocks	Uncle-rewarding mechanism incentivises selfish mining [8].	N/A
Unlimited nodes creation	The victim becomes isolated from the peers [8].	N/A
Public peer selection	An attacker can identify the victim's buckets by the corresponding node ID and insert malicious node IDs to the victim's routing table [8].	N/A

Table 6.3: Ethereum and Solidity Vulnerabilities

Vulnerability	**Resulted Security Issues**	**Number of Findings**
Sole block synchronization	This vulnerability facilitates double-spending and DoS attacks [8].	N/A

Some vulnerabilities are general and cannot be identified by analysing smart contracts. In such cases, we marked them with *N/A* in the *Number of Findings* column. An example of this kind of vulnerabilities is DoS with block stuffing, which is relevant to any contract in Ethereum due to the greedy incentive mechanism. Also, the usage of the outdated compiler cannot be identified during the security audit. We suggest using the Solidity compiler declared in the contracts as it was used for tests and can guarantee the stable execution.

Our findings on Solidity and Ethereum vulnerabilities are listed in the Table 6.4. First of all, there is a risk of reentrancy in the Protocol contract. The following snippet demonstrates reentrancy on a single function *transfer* in *endProcess()*:

```
148     if(owner!=address(0)){
149       owner.transfer(address(this).balance);
150     }
151     EndTime = now;
```

In this example, a fallback function *transfer* is used before the state change in line 151. The *transfer* function used is considered to be resistant to reentrancy attacks. However, after the EIP 1884 was included into the Istanbul hard fork, the function became vulnerable and is not recommended to used anymore [50, 51]. There are several possible solutions to this problem. For instance, we can use the checks-effects-interaction pattern [8, 24, 6, 46, 50, 51]. In other words, the change of the state and/or the update of the balance have to be done before the *transfer* is executed:

```
148     EndTime = now;
149     if(owner!=address(0)){
150       owner.transfer(address(this).balance);
151     }
```

Another solution would be using OpenZeppelin's ReentrancyGuard [52], which allows checking calls for reentrancy and rejecting such calls [51]. In order to use it, we need to inherit from *ReentrancyGuard* and then declare the *endProcess()* as *nonReentrant*.

The third opportunity is a Mutex lock on the state. This method assures that only the owner can change the state [8, 6]. Finally, ConsenSys Diligence does not recommend to use *transfer* at all after the Istanbul hard fork and suggests using *call()* instead [50, 51].

The second vulnerability is block timestamp dependency. Overall, seven cases were identified. In all of them, there is a dependency on *now* which is an alias of *block.timestamp* and, therefore, should be avoided [53]. The easiest way to fix this issue is to use *block.number* instead of *now* [6].

It is important to note that in two cases *now* is assigned to a variable which is never used for any control flow decisions. For this reason, we estimate the potential impact and the risk level as very low. In five other cases, there is a conditional dependence on *now* or a variable relying on it. For example, by manipulating *now* in the following snippet an attacker can influence the rating of the mitigator.

```
133     if(now > Deadline){
134       MitigatorRating = Enums.Rating.NOT_AVAILABLE;
135       return endProcess();
136     }
```

We consider that in these cases the likelihood of the threat event is moderate but the impact is high. The resulting risk level is moderate.

The next found vulnerability is integer overflow in the line 202 of the Protocol contract:

```
201   function setNewDeadline() private{
202     Deadline = now + DeadlineInterval * 1 seconds;
203   }
```

The variable *Deadline* has *uint256* type. This means that it can be only an integer between 0 and 4’294’967’295 ($2^{256} - 1$) [6].

At the moment, there are no restrictions for the variable *DeadlineInterval*:

```
45    function init(uint _DeadlineInterval,uint256 _OfferedFunds,string
         memory _ListOfAddresses) public {
46      require(msg.sender==Target,"[init] sender is not required actor");
47      require(Mitigator!=address(0),"[init] mitigator is not set.");
48      require(CurrentState==Enums.State.REQUEST ||
49        CurrentState==Enums.State.COMPLETE ||
50        CurrentState==Enums.State.ABORT, "[init] State is not appropriate"
            );
51      Target = msg.sender;
52      DeadlineInterval = _DeadlineInterval;
53      OfferedFunds = _OfferedFunds;
```

As a *uint* (alias of *uint256* according to Solidity Documentation [54]), *DeadlineInterval* has the same range as *uint256*. Therefore, under the assumption that *now* is always greater than 0, setting the variable *DeadlineInterval* to its maximum value 4’294’967’295 leads to an integer overflow for *Deadline*. As a result, the variable *Deadline* is erroneously assigned value 0, which means that there is no time left for proof generation.

A Solidity math library *SafeMath.sol* [55] for arithmetic calculations provides a solution to the problem of integer overflow and underflow [8, 6]. We consider this vulnerability to have a moderate risk level because, although the likelihood of the threat occurrence is high, the integer overflow affects only the deadline but not, for instance, the amount of transferred funds. For this reason, the impact of this vulnerability is moderate. According to the NIST Guide for Guide for Conducting Risk Assessments [18], a moderate level of impact and a high likelihood of a threat event result in an overall moderate risk level.

Four low-risk unrestricted write cases were identified. In the following example from the Register contract, any user can set a mitigator address:

```
25    function setMitigator(string memory _name, address _Mitigator)
         public {
26        if(mitigators[_name].isAdded == false){
27            mitigators[_name].Protocol = address(0);
28            mitigators[_name].Mitigator = _Mitigator;
29            mitigators[_name].isAdded = true;
30        }else{
31            emit LogNotValid('Mitigator already registered.');
32        }
33    }
```

In the same contract any user is able to set a protocol address to the mapping of a mitigator:

```
36    function getMitigator(address _protocol, string memory _name)public
         returns(address){
37        require(mitigators[_name].Mitigator != msg.sender, "[
             getMitigator] Protocol is not allowed to be a Mitigator.");
38        if(mitigators[_name].isAdded){
39            mitigators[_name].Protocol = _protocol;
40            return mitigators[_name].Mitigator;
41        }else{
42            emit LogNotValid('Mitigator not registered.');
43            return address(0);
44        }
45    }
```

Proper write restrictions in these cases can help to improve the security of the smart contracts.

Another common vulnerability in the smart contracts are non-validated arguments. We identified 10 cases in BloSS contracts. For instance, in the following example from Protocol, the argument *_Proof* is not validated. As a result, a user can pass any string argument to the function. This can not only lead to unexpected effects during the contract execution, but also allow attackers to upload malicious code instead of the proof.

```
92  function uploadProof(string memory _Proof) public {
93    require(CurrentState==Enums.State.PROOF,"[uploadProof] State is not
         appropriate");
94    // when lazy
95    if(now > Deadline){
96      CurrentState = Enums.State.RATE_T;
97      setNewDeadline();
98      return;
99    }
100   require(msg.sender==Mitigator,"[uploadProof] sender is not required
         actor");
101   Proof = _Proof;
102   CurrentState = Enums.State.RATE_T;
103   setNewDeadline();
104 }
```

In this example, we consider the vulnerability to have a moderate risk level. Both estimated likelihood and potential impact are moderate, as there are only few operations

with the data from this argument. In some other cases, the evaluated risk level is low, because the possible impact of exposure is considered low. All these vulnerabilities can be eliminated by sanitising arguments before using them in further operations and calculations.

The sixth vulnerability type found during the security audit is external call. The following snippet demonstrates how the Register contract is called from the Protocol contract:

```
40      reg = Register(RegisterAddress);
41      // cast mitigator address from address to payable address
42      Mitigator = address(uint160(reg.getMitigator(address(this), _name)))
            ;
```

In this case, the data is both pushed to the external contract and then pulled from it. In general, it is recommended to avoid external calls but when not possible, proper error handling and return value checks need to be implemented [24].

DoS with unexpected revert vulnerability is the next identified type with overall five findings. The reason for this vulnerability is a conditional dependence of the Protocol contract on the Enums contract. For example, in the following snippet the control flow relies on the ratings in Enums:

```
157       if(TargetRating==Enums.Rating.POSITIVE){
158         return satisfied();
159       }else if(TargetRating==Enums.Rating.DISSATISFIED){
160         return dissatisfied();
161       }else{
162         return selfish();
163       }
```

Although there can be a failure in the external call, no checks for return values are implemented. However, we consider this a low-risk vulnerability as the Enums contract does not have *revert* or *throw* functions in place and, therefore, cannot revert operations intentionally.

Finally, 39 cases of erroneous visibility were found during the audit. In all of them, functions are marked *public* although they are not used internally by the contract. Marking them as *external* functions would help to improve efficiency [56]. However, as no enhancement in terms of unauthorized access is required, we consider these vulnerabilities to have low risk.

Table 6.4 presents the details of all 57 findings.

Table 6.4: Ethereum and Solidity Vulnerabilities Findings

ID	**Vulnerability**	**Finding**	**Measures**	**Risk Level**
ESV1	Reentrancy	Reentrancy on a single function *transfer* (line 149) in Protocol contract. The flow control transferred, then the state is changed (*EndTime = now;*).	Possible solutions: 1. Checks-effects-interaction: Updating the state and the balance before executing the *transfer* function [8, 24, 6, 46, 50, 51]. 2. Use OpenZeppelin's ReentrancyGuard [52] to explicitly check calls for reentrancy and reject such calls [51]. 3. Use a Mutex lock on the state to assure that only the owner can change it [8, 6]. 4. Use *call()* instead of *transfer* [50, 51].	High
ESV2	Block timestamp dependency	*StartTime* variable is defined as *now* (line 17) in Protocol. *now* is alias of *block.timestamp* and, therefore, can be manipulated by miners [53].	Use *block.number* instead of *block.timestamp* and *now* [6].	Very Low
ESV3	Block timestamp dependency	Conditional dependency on *now* (lines 95-99) in Protocol. *now* is an alias of *block.timestamp* and, therefore, can be manipulated by miners [53].	Use *block.number* instead of *block.timestamp* and *now* [6].	Moderate

Table 6.4: Ethereum and Solidity Vulnerabilities Findings

ID	Vulnerability	Finding	Measures	Risk Level
ESV4	Block timestamp dependency	Conditional dependency on *now* (lines 110-118) in Protocol. *now* is an alias of *block.timestamp* and, therefore, can be manipulated by miners [53].	Use *block.number* instead of *block.timestamp* and *now* [6].	Moderate
ESV5	Block timestamp dependency	Conditional dependency on *now* (lines 133-136) in Protocol. *now* is an alias of *block.timestamp* and, therefore, can be manipulated by miners [53].	Use *block.number* instead of *block.timestamp* and *now* [6].	Moderate
ESV6	Block timestamp dependency	*EndTime* variable is defined as *now* (line 151) in Protocol. *now* is an alias of *block.timestamp* and, therefore, can be manipulated by miners [53].	Use *block.number* instead of *block.timestamp* and *now* [6].	Very Low
ESV7	Block timestamp dependency	*Deadline* variable relies on *now* (line 202) in Protocol. *now* is an alias of *block.timestamp* and, therefore, can be manipulated by miners [53]. There are several conditional statements in Protocol which rely on *Deadline*.	Use *block.number* instead of *block.timestamp* and *now* [6].	Moderate
ESV8	Block timestamp dependency	Conditional dependency on *now* (lines 263-265) in Protocol. *now* is an alias of *block.timestamp* and, therefore, can be manipulated by miners [53].	Use *block.number* instead of *block.timestamp* and *now* [6].	Moderate

Table 6.4: Ethereum and Solidity Vulnerabilities Findings

ID	Vulnerability	Finding	Measures	Risk Level
ESV9	Integer overflow und underflow	An integer overflow is possible. There is no check whether the result of the calculation stays in the range of the data type *uint256* for *Deadline = now + DeadlineInterval * 1 seconds;* (line 202) in Protocol contract. The variable *Deadline* as *uint256* can be only an integer between 0 and 4'294'967'295 ($2^{256}-1$) [6].	Use a Solidity math library *SafeMath.sol* [55] for arithmetic calculations [8, 6].	Moderate
ESV10	Unrestricted write	No write restrictions for *mitigators[_name].Protocol = address(0); mitigators[_name].Mitigator = _Mitigator; mitigators[_name].isAdded = true;* (lines 27-29) in Register contract. Any user can set a mitigator address.	Add write restrictions.	Low
ESV11	Unrestricted write	No write restrictions for *mitigators[_name].Protocol = _protocol;* (line 39) in Register contract. Any user can set a protocol address to the mapping of a mitigator.	Add write restrictions.	Low
ESV12	Unrestricted write	No write restrictions for *Target = address(uint160(_target));* (line 30) in Protocol contract. Any user can set a target.	Add write restrictions.	Low

Table 6.4: Ethereum and Solidity Vulnerabilities Findings

ID	**Vulnerability**	**Finding**	**Measures**	**Risk Level**
ESV13	Unrestricted write	No write restrictions for line 42 in Protocol contract. Any user can cast a mitigator's address to payable address.	Add write restrictions.	Low
ESV14	Non-validated arguments	The argument *completed* is not validated in the function *setCompleted(uint completed)* (lines 16-18) in Migrations contract. A user can pass any argument which can have an unexpected effect on the contract execution.	Sanitise the argument before using it in calculations.	Low
ESV15	Non-validated arguments	The argument *new_address* is not validated in the function *upgrade(address new_address)* (lines 20-23) in Migrations contract. A user can pass any argument which can have an unexpected effect on the contract execution.	Sanitise the argument before using it in calculations.	Low
ESV16	Non-validated arguments	The argument *_Mitigator* is not validated in the function *setMitigator(string memory _name, address _Mitigator)* (lines 25-33) in Register contract. A user can pass any argument which can have an unexpected effect on the contract execution.	Sanitise the argument before using it in calculations.	Moderate

Table 6.4: Ethereum and Solidity Vulnerabilities Findings

ID	Vulnerability	Finding	Measures	Risk Level
ESV17	Non-validated arguments	The argument *_protocol* is not validated in the function *getMitigator(address _protocol, string memory _name)* (lines 36-45) in Register contract. A user can pass any argument which can have an unexpected effect on the contract execution.	Sanitise the argument before using it in calculations.	Moderate
ESV18	Non-validated arguments	The argument *_target* is not validated in the function *setTarget(address _target)* (lines 29-31) in Protocol contract. A user can pass any argument which can have an unexpected effect on the contract execution.	Sanitise the argument before using it in calculations.	Moderate
ESV19	Non-validated arguments	The argument *_Register* is not validated in the function *setRegister(address _Register)* (lines 33-36) in Protocol contract. A user can pass any argument which can have an unexpected effect on the contract execution.	Sanitise the argument before using it in calculations.	Moderate
ESV20	Non-validated arguments	The argument *_name* is not validated in the function *getSpecificMitigator(string memory _name)* (lines 38-43) in Protocol contract. A user can pass any argument which can have an unexpected effect on the contract execution.	Sanitise the argument before using it in calculations.	Moderate

Table 6.4: Ethereum and Solidity Vulnerabilities Findings

ID	Vulnerability	Finding	Measures	Risk Level
ESV21	Non-validated arguments	The arguments *_DeadlineInterval*, *_OfferedFunds* and *_ListOfAddresses* are not validated in the function *init(uint _DeadlineInterval, uint256 _OfferedFunds, string memory _ListOfAddresses)* (lines 45-57) in Protocol contract. A user can pass any argument which can have an unexpected effect on the contract execution.	Sanitise the argument before using it in calculations.	Moderate
ESV22	Non-validated arguments	The arguments *_DeadlineInterval*, *_OfferedFunds*, *_ListOfAddresses* and *_Mitigator* are not validated in the function *function reInit(uint _DeadlineInterval, uint256 _OfferedFunds, string memory _ListOfAddresses, address _Mitigator)* (lines 59-69) in Protocol contract. A user can pass any argument which can have an unexpected effect on the contract execution.	Sanitise the argument before using it in calculations.	Moderate
ESV23	Non-validated arguments	The argument *_Proof* is not validated in the function *uploadProof(string memory _Proof)* (lines 92-104) in Protocol contract. A user can pass any argument which can have an unexpected effect on the contract execution.	Sanitise the argument before using it in calculations.	Moderate

Table 6.4: Ethereum and Solidity Vulnerabilities Findings

ID	Vulnerability	Finding	Measures	Risk Level
ESV24	External call	External call using *getMitigator()* function (line 42) of the Register contract by the Protocol contract. The data is pushed to the external contract.	If it is impossible to avoid the external call, it is necessary to implement error handling and check the return value [24].	Moderate
ESV25	DoS with unexpected revert	Conditional dependence on the Enums contract in the Protocol contract (lines 157-163).	No specific measures required.	Low
ESV26	DoS with unexpected revert	Conditional dependence on the Enums contract in the Protocol contract (lines 166-173).	No specific measures required.	Low
ESV27	DoS with unexpected revert	Conditional dependence on the Enums contract in the Protocol contract (lines 178-180).	No specific measures required.	Low
ESV28	DoS with unexpected revert	Conditional dependence on the Enums contract in the Protocol contract (lines 184-186).	No specific measures required.	Low
ESV29	DoS with unexpected revert	Conditional dependence on the Enums contract in the Protocol contract (lines 190-194).	No specific measures required.	Low
ESV30	Erroneous visibility	The public function *setCompleted()* (lines 16-18) in Migrations is never called internally. For this reason, it should be marked as external in order to improve efficiency of the contract [56].	Mark the function as external.	Low

Table 6.4: Ethereum and Solidity Vulnerabilities Findings

ID	Vulnerability	Finding	Measures	Risk Level
ESV31	Erroneous visibility	The public function *upgrade()* (lines 20-23) in Migrations is never called internally. For this reason, it should be marked as external in order to improve efficiency of the contract [56].	Mark the function as external.	Low
ESV32	Erroneous visibility	The public function *setMitigator()* (lines 25-33) in Register is never called internally. For this reason, it should be marked as external in order to improve efficiency of the contract [56].	Mark the function as external.	Low
ESV33	Erroneous visibility	The public function *getMitigator()* (lines 36-45) in Register is never called internally. For this reason, it should be marked as external in order to improve efficiency of the contract [56].	Mark the function as external.	Low
ESV34	Erroneous visibility	The public function *getProtocol()* (lines 48-55) in Register is never called internally. For this reason, it should be marked as external in order to improve efficiency of the contract [56].	Mark the function as external.	Low

Table 6.4: Ethereum and Solidity Vulnerabilities Findings

ID	Vulnerability	Finding	Measures	Risk Level
ESV35	Erroneous visibility	The public function *getCreator()* (lines 58-60) in Register is never called internally. For this reason, it should be marked as external in order to improve efficiency of the contract [56].	Mark the function as external.	Low
ESV36	Erroneous visibility	The public function *setTarget()* (lines 29-31) in Protocol is never called internally. For this reason, it should be marked as external in order to improve efficiency of the contract [56].	Mark the function as external.	Low
ESV37	Erroneous visibility	The public function *setRegister()* (lines 33-36) in Protocol is never called internally. For this reason, it should be marked as external in order to improve efficiency of the contract [56].	Mark the function as external.	Low
ESV38	Erroneous visibility	The public function *getSpecificMitigator()* (lines 38-43) in Protocol is never called internally. For this reason, it should be marked as external in order to improve efficiency of the contract [56].	Mark the function as external.	Low
ESV39	Erroneous visibility	The public function *init()* (lines 45-57) in Protocol is never called internally. For this reason, it should be marked as external in order to improve efficiency of the contract [56].	Mark the function as external.	Low

Table 6.4: Ethereum and Solidity Vulnerabilities Findings

ID	Vulnerability	Finding	Measures	Risk Level
ESV40	Erroneous visibility	The public function *reInit()* (lines 59-69) in Protocol is never called internally. For this reason, it should be marked as external in order to improve efficiency of the contract [56].	Mark the function as external.	Low
ESV41	Erroneous visibility	The public function *approve()* (lines 71-79) in Protocol is never called internally. For this reason, it should be marked as external in order to improve efficiency of the contract [56].	Mark the function as external.	Low
ESV42	Erroneous visibility	The public function *sendFunds()* (lines 81-90) in Protocol is never called internally. For this reason, it should be marked as external in order to improve efficiency of the contract [56].	Mark the function as external.	Low
ESV43	Erroneous visibility	The public function *uploadProof()* (lines 92-104) in Protocol is never called internally. For this reason, it should be marked as external in order to improve efficiency of the contract [56].	Mark the function as external.	Low

Table 6.4: Ethereum and Solidity Vulnerabilities Findings

ID	Vulnerability	Finding	Measures	Risk Level
ESV44	Erroneous visibility	The public function *ratingByTarget()* (lines 106-128) in Protocol is never called internally. For this reason, it should be marked as external in order to improve efficiency of the contract [56].	Mark the function as external.	Low
ESV45	Erroneous visibility	The public function *ratingByMitigator()* (lines 130-140) in Protocol is never called internally. For this reason, it should be marked as external in order to improve efficiency of the contract [56].	Mark the function as external.	Low
ESV46	Erroneous visibility	The public function *getMitigator()* (lines 205-208) in Protocol is never called internally. For this reason, it should be marked as external in order to improve efficiency of the contract [56].	Mark the function as external.	Low
ESV47	Erroneous visibility	The public function *getTarget()* (lines 210-213) in Protocol is never called internally. For this reason, it should be marked as external in order to improve efficiency of the contract [56].	Mark the function as external.	Low

Table 6.4: Ethereum and Solidity Vulnerabilities Findings

ID	Vulnerability	Finding	Measures	Risk Level
ESV48	Erroneous visibility	The public function *getRegister()* (lines 215-218) in Protocol is never called internally. For this reason, it should be marked as external in order to improve efficiency of the contract [56].	Mark the function as external.	Low
ESV49	Erroneous visibility	The public function *getListOfAddresses()* (lines 220-223) in Protocol is never called internally. For this reason, it should be marked as external in order to improve efficiency of the contract [56].	Mark the function as external.	Low
ESV50	Erroneous visibility	The public function *getProof()* (lines 225-228) in Protocol is never called internally. For this reason, it should be marked as external in order to improve efficiency of the contract [56].	Mark the function as external.	Low
ESV51	Erroneous visibility	The public function *getCurrentState()* (lines 230-233) in Protocol is never called internally. For this reason, it should be marked as external in order to improve efficiency of the contract [56].	Mark the function as external.	Low

Table 6.4: Ethereum and Solidity Vulnerabilities Findings

ID	Vulnerability	Finding	Measures	Risk Level
ESV52	Erroneous visibility	The public function *getTargetRating()* (lines 236-239) in Protocol is never called internally. For this reason, it should be marked as external in order to improve efficiency of the contract [56].	Mark the function as external.	Low
ESV53	Erroneous visibility	The public function *getMitigatorRating()* (lines 241-244) in Protocol is never called internally. For this reason, it should be marked as external in order to improve efficiency of the contract [56].	Mark the function as external.	Low
ESV54	Erroneous visibility	The public function *getStartTime()* (lines 246-249) in Protocol is never called internally. For this reason, it should be marked as external in order to improve efficiency of the contract [56].	Mark the function as external.	Low
ESV55	Erroneous visibility	The public function *getEndTime()* (lines 251-254) in Protocol is never called internally. For this reason, it should be marked as external in order to improve efficiency of the contract [56].	Mark the function as external.	Low

Table 6.4: Ethereum and Solidity Vulnerabilities Findings

ID	Vulnerability	Finding	Measures	Risk Level
ESV56	Erroneous visibility	The public function *getDeadline()* (lines 256-259) in Protocol is never called internally. For this reason, it should be marked as external in order to improve efficiency of the contract [56].	Mark the function as external.	Low
ESV57	Erroneous visibility	The public function *getValidTime()* (lines 261-268) in Protocol is never called internally. For this reason, it should be marked as external in order to improve efficiency of the contract [56].	Mark the function as external.	Low

In order to determine the risk level of our findings, we applied the risk level assessment scale from the NIST Guide for Conducting Risk Assessments [18] described in Section 2.2.2 to them. Table 6.5 presents the allocation of the findings according to the framework.

Table 6.5: Allocation of Findings on the Risk Level Assessment Scale. Adapted from [18]

Likelihood	**Level of Impact**				
	Very Low	**Low**	**Moderate**	**High**	**Very High**
Very High					
High		ESV10-15	ESV9, 24	ESV1	
Moderate			ESV16-23	ESV3-5, 7-8	
Low	ESV2, 6	ESV25-29	ESV30-57		
Very Low					

Chapter 7

Discussion

After the findings of the security audit tools were classified, we performed a manual analysis and validated findings against true and false positives. As a result, we identified overall 85 false positives. Almost half of them belong to the exception handling vulnerability type. In particular, 34 findings by Remix contain a suggestion to consider using *assert(x)* instead of *require(x)*. However, the main purpose of using *require(x)* in BloSS is to check that certain conditions are met during the execution and not to handle internal errors. For this reason, according to Solidity documentation [57], the choice of the function is correct.

Solidity 2.0 has also identified one exception handling vulnerability which we consider to be a false positive. According to the tool, the return value in the line 22 in Migrations needs to be explicitly checked for an error. However, we believe that no special error handling is required in this case because *setCompleted()* only assigns an argument to a public variable and is not error prone.

```
20  function upgrade(address new_address) public restricted {
21    Migrations upgraded = Migrations(new_address);
22    upgraded.setCompleted(last_completed_migration);
23  }
```

In the same line, Mythril and Securify identified an external call. However, this is a recursive call of Migration contract by itself. This contract does not contain any critical functions or data. For this reason, we consider this finding not relevant from a security point of view.

Another vulnerability with a large number of false positives is gas overspent. It was identified only by Remix. In overall 25 cases, Remix suggests adding a gas requirement limit to functions. As neither of the functions contain gas costly patterns discussed in Section 5.3, we consider it unnecessary to add gas limits to them.

There are also eight cases of unrestricted write identified by Securify which we consider to be false positives. For example, Securify claims that there are no write restrictions for the line 55 in Protocol. However, as the following snippet demonstrates, only a target can call this function and, therefore, perform these write operations. Besides, it is only possible to do this in three states of the process and only if a mitigator is selected.

```
45  function init(uint _DeadlineInterval,uint256 _OfferedFunds,string
       memory _ListOfAddresses) public {
46    require(msg.sender==Target,"[init] sender is not required actor");
47    require(Mitigator!=address(0),"[init] mitigator is not set.");
48    require(CurrentState==Enums.State.REQUEST ||
49      CurrentState==Enums.State.COMPLETE ||
50      CurrentState==Enums.State.ABORT, "[init] State is not appropriate"
         );
51    Target = msg.sender;
52    DeadlineInterval = _DeadlineInterval;
53    OfferedFunds = _OfferedFunds;
54    ListOfAddresses = _ListOfAddresses;
55    CurrentState = Enums.State.APPROVE;
56    emit ProcessCreated(msg.sender,address(this));
57  }
```

Besides, Securify identified two cases of division in Protocol and suggests being careful about them because of the integer rounding by division which can influence the computation result. However, no division operations were found during the manual review. Thus, we consider these findings false positives.

Securify also claims that the order of transactions can influence who is the receiver of the funds and the amount of the transferred funds in the following piece of code from Protocol:

```
149       owner.transfer(address(this).balance);
```

However, due to the implemented system of different states (Request, Approve, Funding etc.), it is only possible to perform the transfer in states Rate_T or Rate_M: after the rating evaluation by target and, if necessary, by mitigator is completed. For this reason, the receiver and the amount of funds are not affected by the transaction order but by the participant's ratings and whether the proof of work was uploaded on time. Yet it is important to note that Protocol does not check the content of the uploaded proof.

In the same line of the code, Securify identified an external call with a target which can be potentially manipulated by attackers. However, we do not consider this finding as a vulnerability because the target of the call (*owner*) is provided by the *evaluate()* function and can be only the target or the mitigator.

Moreover, Securify marked Register contract as greedy. According to the tool, the funds can be received by the contract but then get locked because there is no opportunity to extract them. As Register does not support any kind of Ether transfer, we consider this finding to be a false positive.

Finally, Remix has found a vulnerability which is not on our list. According to the tool and to Solidity documentation [58], when a *string* is converted to *bytes*, its length is calculated in bytes and not in characters as it might be expected, which can lead to confusion. However, in the following snippet from Protocol, it is only checked if the proof is empty, which would mean that both length in characters and in bytes would equal zero.

```
114       if(bytes(Proof).length==0){
115         return endProcess();
116       }
```

Thus, although we acknowledge that developers should be careful when converting a *string* to *bytes*, in the current example we do not consider it as a vulnerability.

There are also some other vulnerabilities which did not match any vulnerabilities on our list but were considered to be true positives. For example, Remix finds it confusing that there are two variables in Protocol with similar names (*Target* and *_target*). Securify 2.0 claims that the Solidity pragma version statement in Migrations is too complex:

```
1 pragma solidity >=0.4.21 <0.6.0;
```

Moreover, Securify 2.0 suggests renaming *last_completed_migration* variable in Migrations because it does not comply with Solidity naming convention. As all of these findings are correct, we evaluated them as true positives although they do not correspond to any vulnerability type on our list.

Table 7.1 contains an overview of our results, while the detailed information can be found in Appendix B.

Table 7.1: Overview of True Positives and False Positives in the Findings of the Security Audit Tools

	Mythril	**Securify**	**Securify 2.0**	**Remix**	**Manticore**
True Positives	5	10	16	8	0
False Positives	1	15	1	60	0
Total	**6**	**25**	**17**	**68**	**0**

It is important to note that we removed two findings of Remix from the overview because they were related to an internal error message in Remix and, therefore, could not be considered neither as true positives nor as false positives. As a result, Remix has only 68 instead of 70 findings in Table 7.1.

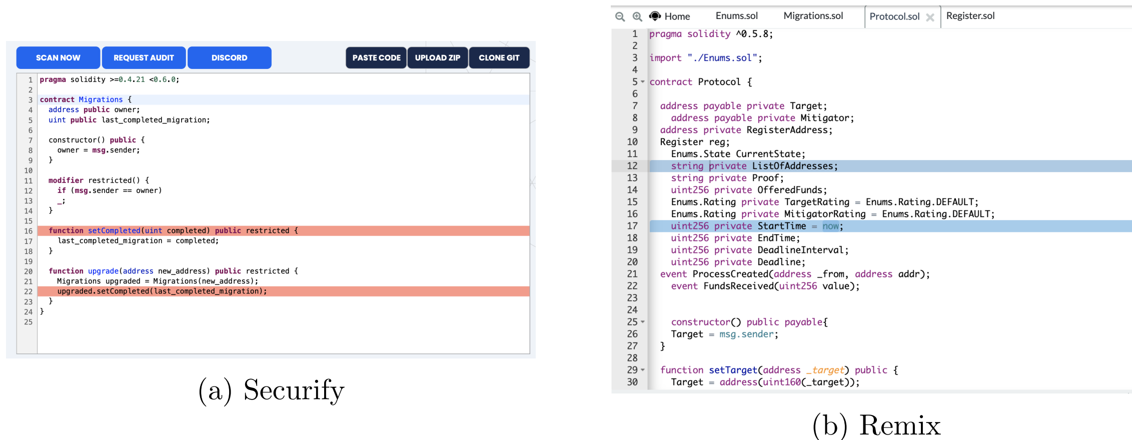
The table demonstrates also that the number of findings does not necessarily correlate with the number of true positives. Even though Remix has by far the highest number of findings, it has a rather low number of true positives. On the other hand, Securify 2.0 with relatively few findings has 16 true positives and only one false positive.

In the next step, we have compared the findings of the security tools with the results of our security audit. In order to do this, we went through all 57 identified vulnerabilities and checked if they were found by the security tools. The results are presented in Table A.1 in Appendix.

Although Securify 2.0 has not analysed Protocol which is the main and the most complex contract in BloSS, it was able to identify the highest number of security audit findings and has, therefore, the lowest number of false negatives. With seven findings, Securify and Remix are together on the second place among the tools.

Besides, according to one of the internal error messages in Remix, it was impossible to perform the check-effect-interaction. Thus, Remix could not check if there is a reentrancy, and it is unclear if it would identify this vulnerability correctly.

Altogether, we would like to stress that there are big differences between the security tools. As discussed in Section 3.2, they use various analysis types and strategies and have different vulnerability types in scope. In the concrete example of BloSS smart contracts, Securify and Securify 2.0 performed the best. Although Remix had the largest number of findings, most of them were evaluated as false positives during the manual security analysis. Mythril demonstrated good results regarding the ration of true and false positives. However, as it focuses only on finding four types of vulnerabilities, it has a high number of false negatives.



(a) Securify

(b) Remix

Figure 7.1: Highlighted Findings

The security tools vary also regarding their usability. Two of them, Securify and Remix, highlight the line with findings in the code which makes it easier to find vulnerable pieces of code. Moreover, Mythril and Securify 2.0 assign a risk level to every finding. Securify marks its findings with color (*e.g.*, red). However, it is not clear if the color indicates the risk level or something else. There is no manual for the tool on the official page which could explain this.

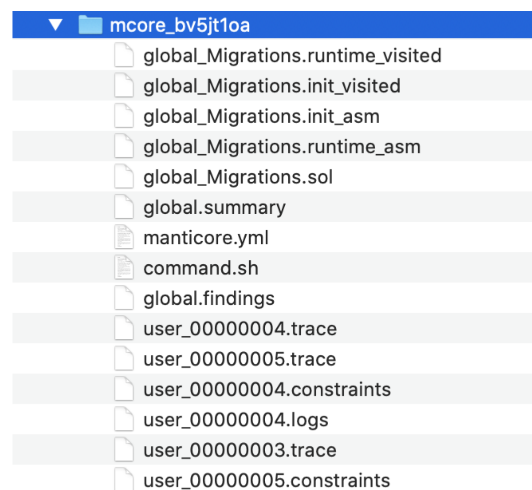


Figure 7.2: Reports Generated by Manticore

Although we do not have any findings from Manticore to review, it is important to note

that it was the only dynamic tool that we tested. Besides, in contrast to other tools, it creates automatically multiple reports in the folder where the project is located.

Regarding the security audit findings, we would like to emphasise that although we identified overall 57 vulnerabilities in the smart contracts, only one of them is considered a high risk issue. Most of them, 39 findings, have a low risk level.

Table 7.2: Number of Findings per Risk Level and Contract

Smart Contract	**Risk Level**					**Total**
	Very Low	**Low**	**Moderate**	**High**	**Very High**	
Enums	0	0	0	0	0	**0**
Migrations	0	4	0	0	0	**4**
Register	0	6	2	0	0	**8**
Protocol	2	29	13	1	0	**45**
Total	**2**	**39**	**15**	**1**	**0**	**57**

Table 7.2 demonstrates a short overview of the findings. As expected, the Protocol contract which is the main and the most complicated contract in BloSS has the highest number of findings.

Chapter 8

Conclusion and Final Considerations

In conclusion, it is important to highlight that this thesis was able to meet the initial goals. By reviewing multiple studies, six general blockchain vulnerabilities and a list of 41 vulnerabilities related to Ethereum and Solidity were identified. In order to get the first impression on the security level of the smart contracts in BloSS, we used five tools for automated security audit: Mythril, Securify, Securify 2.0, Remix and Manticore. Their findings were manually reviewed and true and false positives were identified.

In the next step, we analysed BloSS and its contracts regarding the previously determined general blockchain vulnerabilities, Ethereum and Solidity vulnerabilities and CIA triad. The result of the security audit is a list of 57 findings, where their risk level was estimated based on the NIST Guide for Conducting Risk Assessments. Moreover, we classified the findings of the security tools using our list of vulnerabilities and compared them with the security audit results. As a result, we identified cases in which the findings match each other as well as false negatives induced by security tools.

The comparison of the security tools is also a contribution of this thesis. Altogether, we would like to note that Securify 2.0 has a big potential. At the time of writing, it is able to detect the largest number of vulnerabilities among all analyzed tools. Besides, in the particular case of BloSS, it had the best performance among the tools. Even though it could not analyse Protocol contract due to the unsolved bug mentioned in Discussion, it identified 17 findings. 16 of them were considered true positives, which is the best result both in relative and absolute terms. Moreover, Securify 2.0 had the highest number of matches with the security audit results and, thus, the lowest number of false negatives. Nevertheless, as Securify 2.0 is rather new, it has some unsolved bugs and does not support a Web GUI.

During the current project, we had to overcome several difficulties. First of all, as several studies were used for creating our list of Ethereum and Solidity vulnerabilities, which entailed removing duplicates. The studies often use different names for the same vulnerability. For example, the terms *frozen Ether* and *greedy contract* stand in fact for the same issue. By analysing the descriptions of the vulnerabilities, we tried to figure out whether they belong to the same security issue or not.

A similar problem occurred during the classification of security tools findings. Each of the tools uses its own vulnerability names, which makes it difficult to compare their findings. For example, Securify 2.0 identified several cases of *External Calls of Functions*. However, the analysis of the findings descriptions showed that it was actually erroneous visibility (functions were marked *public* instead of *external*). In this case, the naming can lead to confusion. A standardised and universally acknowledged classification of smart contract vulnerabilities would solve this problem.

Besides, it was often difficult to estimate the risk level of the findings. The evaluation of the potential impact and likelihood of a threat event is rather subjective. In general, we followed a conservative strategy and in uncertain situations assigned rather a higher risk level to a finding than a lower one.

After the subjectivity of risk level estimation, the biggest limitation of the current study is that we have used the security tools before performing our own analysis. Therefore, the findings identified by the tools could influence our perception of the smart contracts and, thus, our findings.

Finally, it is important to note that security audit of smart contracts is still a new and fast developing area. Although there are a lot of studies and proposed approaches, this field lacks standardisation. A universally acknowledged classification of vulnerabilities would make the communication and comparison of different studies easier. At the moment, there is no framework for conducting manual security audit of Ethereum smart contracts. Such a guide would be especially useful for young professionals who do not have a lot of experience in this area yet. Moreover, although security audit tools allow to perform security analysis automatically, they are not error-free, which means that a manual security audit is still necessary.

Bibliography

[1] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System." http://www.bitcoin.org/bitcoin.pdf, 2008.

[2] B. Rodrigues, T. Bocek, and B. Stiller, "Enabling a Cooperative, Multi-domain DDoS Defense by a Blockchain Signaling System (BloSS)," October 2017.

[3] A. Gruhler, B. Rodrigues, and B. Stiller, "A Reputation Scheme for a Blockchain-based Network Cooperative Defense," in *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pp. 71–79, 2019.

[4] S. Mannhart, B. Rodrigues, E. Scheid, S. S. Kanhere, and B. Stiller, "Toward Mitigation-as-a-Service in Cooperative Network Defenses," in *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*, pp. 362–367, 2018.

[5] S. Trendafilov, "Cooperative Signaling." Bachelor's thesis at the Communication Systems Group of the University of Zurich, August 2019.

[6] P. Praitheeshan, L. Pan, J. Yu, J. Liu, and R. Doss, "Security Analysis Methods on Ethereum Smart Contract Vulnerabilities: A Survey," 2019.

[7] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making Smart Contracts Smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, (New York, NY, USA), p. 254–269, Association for Computing Machinery, 2016.

[8] H. Chen, M. Pendleton, L. Njilla, and S. Xu, "A Survey on Ethereum Systems Security: Vulnerabilities, Attacks and Defenses," 2019.

[9] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang, "An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends," in *2017 IEEE International Congress on Big Data (BigData Congress)*, pp. 557–564, 2017.

[10] D. Vujičić, D. Jagodić, and S. Ranđić, "Blockchain Technology, Bitcoin, and Ethereum: A Brief Overview," in *2018 17th International Symposium INFOTEH-JAHORINA (INFOTEH)*, pp. 1–6, 2018.

[11] M. Yang, T. Zhu, K. Liang, W. Zhou, and R. H. Deng, "A Blockchain-Based Location Privacy-Preserving Crowdsensing System," *Future Generation Computer Systems*, vol. 94, pp. 408–418, 2019.

[12] R. Zhang, R. Xue, and L. Liu, "Security and Privacy on Blockchain," 2019.

[13] L. Law, S. Sabett, and J. Solinas, "How to Make a Mint: The Cryptography of Anonymous Electronic Cash," *The American University law review*, vol. 46, p. 6, 1997.

[14] V. Buterin, "Ethereum Whitepaper: A Next-Generation Smart Contract and Decentralized Application Platform." `https://ethereum.org/en/whitepaper/`, 2013.

[15] D. D. Wood, "Ethereum: A Secure Decentralised Generalised Transaction Ledger. Petersburg Version." `https://ethereum.github.io/yellowpaper/paper.pdf`, 2020.

[16] N. Szabo, "Smart Contracts." `https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html`, 1994.

[17] P. Gregory, *CISSP Guide to Security Essentials.* Boston, MA, USA: Cengage Learning, 2nd ed., 2015.

[18] R. M. Blank and P. D. Gallagher, *Guide for Conducting Risk Assessments. NIST Special Publication (SP) 800-30 Revision 1.* Gaithersburg, MD: National Institute of Standards and Technology, 2012.

[19] N. Atzei, M. Bartoletti, and T. Cimoli, "A Survey of Attacks on Ethereum Smart Contracts (SoK)," pp. 164–186, March 2017.

[20] M. Alharby and A. v. Moorsel, "Blockchain Based Smart Contracts: A Systematic Mapping Study," *Computer Science & Information Technology (CS & IT)*, August 2017.

[21] W. Dingman, A. Cohen, N. Ferrara, A. Lynch, P. Jasinski, P. E. Black, and L. Deng, "Defects and Vulnerabilities in Smart Contracts, a Classification using the NIST Bugs Framework," *International Journal of Networked and Distributed Computing*, vol. 7, pp. 121–132, 2019.

[22] NIST, "The Bugs Framework." `https://samate.nist.gov/BF/index.html/`, 2016.

[23] CWE, "Common Weakness Enumeration." `https://cwe.mitre.org/`.

[24] A. Dika and M. Nowostawski, "Security Vulnerabilities in Ethereum Smart Contracts," in *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pp. 955–962, 2018.

[25] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," pp. 238–252, January 1977.

[26] B. Mueller, "Smashing Ethereum Smart Contracts for Fun and Real Profit." https://conference.hitb.org/hitbsecconf2018ams/materials/WHITEPAPERS/WHITEPAPER0-%20Bernhard%20Mueller%20-%20Smashing%20Ethereum%20Smart%20Contracts%20for%20Fun%20and%20ACTUAL%20Profit.pdf/, 2018.

[27] ConsenSys Diligence, "Mythril." https://github.com/ConsenSys/mythril/.

[28] ConsenSys Diligence, "Mythril Documentation." https://mythril-classic.readthedocs.io/en/master/index.html/.

[29] ConsenSys Diligence, "Smart Contract Weakness Classification Registry." https://swcregistry.io/.

[30] ConsenSys Diligence, "MythX." https://mythx.io/about/.

[31] P. Tsankov, A. Dan, D. D. Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical Security Analysis of Smart Contracts," 2018.

[32] ChainSecurity, "Securify." https://securify.chainsecurity.com/.

[33] Secure, Reliable, Intelligent Systems Lab ETH Zurich, "Securify 2.0." https://github.com/eth-sri/securify2/.

[34] P. Tsankov, "Release of Securify v2.0." https://medium.com/chainsecurity/release-of-securify-v2-0-6304a40034f/, January 2020.

[35] Remix, ""Remix IDE Documentation. Solidity Static Analysis"." https://remix-ide.readthedocs.io/en/latest/static_analysis.html/.

[36] I. Nikolic, "MAIAN." https://github.com/ivicanikolicsg/MAIAN/.

[37] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding The Greedy, Prodigal, and Suicidal Contracts at Scale," 2018.

[38] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts," 2019.

[39] Trail of Bits, "Manticore." https://github.com/trailofbits/manticore/.

[40] K. Bhargavan, N. Swamy, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, and T. Sibut-Pinote, "Formal Verification of Smart Contracts: Short Paper," pp. 91–96, October 2016.

[41] S. Amani, M. Bégel, M. Bortin, and M. Staples, "Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL," in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, (New York, NY, USA), p. 66–77, Association for Computing Machinery, 2018.

[42] Z. Yang and H. Lei, "FEther: An Extensible Definitional Interpreter for Smart-Contract Verifications in Coq," *IEEE Access*, vol. 7, p. 37770–37791, 2019.

[43] D. Bünzli, "Cooperative Signaling Protocol." Master Basis Module at the Communication Systems Group of the University of Zurich, February 2019.

[44] X. L. Yu, "Oyente." `https://github.com/melonproject/oyente/`.

[45] X. L. Yu, "Oyente. Web GUI." `https://oyente.melonport.com/#version=soljson-v0.7.0+commit.9e61f92b.js/`.

[46] N. F. Samreen and M. H. Alalfi, "Reentrancy Vulnerability Identification in Ethereum Smart Contracts," in *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pp. 22–29, 2020.

[47] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized Smart Contracts Devour Your Money," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 442–446, 2017.

[48] T. Chen, X. Li, Y. Wang, J. Chen, Z. Li, X. Luo, M. H. Au, and X. Zhang, "An Adaptive Gas Cost Mechanism for Ethereum to Defend Against Under-Priced DoS Attacks," 2017.

[49] F. Fatz, P. Hake, and P. Fettke, "Confidentiality-preserving Validation of Tax Documents on the Blockchain," in *WI 2020: Proceedings der 15. Internationalen Tagung Wirtschaftsinformatik 2020* (N. Gronau, M. Heine, H. Krasnova, and K. Pousttchi, eds.), vol. 1, pp. 1262–1277, Berlin, Germany: GITO, March 2020.

[50] ConsenSys Diligence, "Ethereum Smart Contract Security Best Practices. Don't use Transfer() or Send()." `https://consensys.github.io/smart-contract-best-practices/recommendations/#dont-use-transfer-or-send/`.

[51] S. Marx, "Stop Using Solidity's Transfer() Now." `https://diligence.consensys.net/blog/2019/09/stop-using-soliditys-transfer-now/`, 2019.

[52] OpenZeppelin, "ReentrancyGuard." `https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/ReentrancyGuard.sol`.

[53] Ethereum, "Solidity Documentation. Units and Global Variables." `https://solidity.readthedocs.io/en/v0.6.12/units-and-global-variables.html`.

[54] Ethereum, "Solidity Documentation. Types." `https://solidity.readthedocs.io/en/latest/types.html`.

[55] OpenZeppelin, "SafeMath Library." `https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/math/SafeMath.sol`.

[56] Ethereum, "Solidity Documentation. Contracts." `https://solidity.readthedocs.io/en/v0.7.0/contracts.html`.

[57] Ethereum, "Solidity Documentation. Control Structures." `https://solidity.readthedocs.io/en/latest/control-structures.html`.

[58] Ethereum, “Solidity Documentation. Bytes and Strings as Arrays.” https://solidity.readthedocs.io/en/latest/types.html#bytes-and-strings-as-arrays.

Abbreviations

API	Application Programming Interface
BloSS	Blockchain Signaling System
CIA	Confidentiality-Integrity-Availability
DAO	Decentralised Autonomous Organisation
DDoS	Distributed Denial-of-Service
DoS	Denial-of-Service
EVM	Ethereum Virtual Machine
GUI	Graphical User Interface
IDE	Integrated Development Environment
NIST	National Institute of Standards and Technology
REST	Representational State Transfer

Glossary

Block timestamp dependency Dependence on *block.timestamp* or its alias which can be manipulated by miners.

Distributed Denial-of-Service A target of an attack is flooded with requests from different sources. The aim of the attackers is to overload the target with the requests.

Ether Native cryptocurrency in Ethereum.

Gas A fee required to pay for transaction or contract execution in Ethereum.

Reentrancy If one contract hands over the control to another contract, the second contract can call back into the first one several times before the first initiated interaction is completed [24].

Smart Contract A programming code which automatically executes the terms of an agreement.

Solidity High-level Turing-complete programming language.

Transaction ordering dependence The order of transactions can influence the execution of a smart contract which leads to race conditions.

List of Figures

2.1 The CIA Triad. Adapted from [17] 7

4.1 Signaling Process. From [5] 16

5.1 Oyente's Web GUI 18

7.1 Highlighted Findings 52

7.2 Reports Generated by Manticore 52

List of Tables

2.1 Risk Level Assessment Scale. Adapted from [18] 8
2.2 Expected Effect of Different Risk Levels. Adapted from [18] 8
3.1 Comparison of Tools for Automated Security Audit 14
6.1 Number of Findings per Security Audit Tool 23
6.2 Classification of the Security Audit Tools Findings 24
6.3 Ethereum and Solidity Vulnerabilities 26
6.3 Ethereum and Solidity Vulnerabilities 27
6.3 Ethereum and Solidity Vulnerabilities 28
6.3 Ethereum and Solidity Vulnerabilities 29
6.3 Ethereum and Solidity Vulnerabilities 30
6.4 Ethereum and Solidity Vulnerabilities Findings 34
6.4 Ethereum and Solidity Vulnerabilities Findings 35
6.4 Ethereum and Solidity Vulnerabilities Findings 36
6.4 Ethereum and Solidity Vulnerabilities Findings 37
6.4 Ethereum and Solidity Vulnerabilities Findings 38
6.4 Ethereum and Solidity Vulnerabilities Findings 39
6.4 Ethereum and Solidity Vulnerabilities Findings 40
6.4 Ethereum and Solidity Vulnerabilities Findings 41
6.4 Ethereum and Solidity Vulnerabilities Findings 42
6.4 Ethereum and Solidity Vulnerabilities Findings 43

6.4 Ethereum and Solidity Vulnerabilities Findings 44

6.4 Ethereum and Solidity Vulnerabilities Findings 45

6.4 Ethereum and Solidity Vulnerabilities Findings 46

6.4 Ethereum and Solidity Vulnerabilities Findings 47

6.5 Allocation of Findings on the Risk Level Assessment Scale. Adapted from [18] 47

7.1 Overview of True Positives and False Positives in the Findings of the Security Audit Tools 51

7.2 Number of Findings per Risk Level and Contract 53

A.1 Security Audit Findings Identified by Security Tools 71

B.1 Mythril's Findings 77

B.2 Securify's Findings 77

B.3 Securify 2.0's Findings 78

B.4 Remix's Findings 79

B.5 Links to Securify's Reports 81

Appendix A

Comparison of the Security Audit Findings with the Results of the Security Audit Tools

Table A.1: Security Audit Findings Identified by Security Tools

ID	Vulnerability	Mythril	Securify	Securify 2.0	Remix	Manticore
ESV1	Reetrancy		✓			
ESV2	Block timestamp dependency				✓	
ESV3	Block timestamp dependency				✓	
ESV4	Block timestamp dependency				✓	
ESV5	Block timestamp dependency				✓	
ESV6	Block timestamp dependency				✓	
ES7	Block timestamp dependency				✓	
ESV8	Block timestamp dependency	✓			✓	

ESV9	Integer overflow und underflow					
ESV10	Unrestricted write	✓	✓	✓		
ESV11	Unrestricted write		✓			
ESV12	Unrestricted write		✓			
ESV13	Unrestricted write		✓			
ESV14	Non-validated arguments		✓	✓		
ESV15	Non-validated arguments			✓		
ESV16	Non-validated arguments			✓		
ESV17	Non-validated arguments			✓		
ESV18	Non-validated arguments					
ESV19	Non-validated arguments					
ESV20	Non-validated arguments					
ESV21	Non-validated arguments					
ESV22	Non-validated arguments					
ESV23	Non-validated arguments					
ESV24	External call	✓	✓			

ESV25	DoS with unexpected revert					
ESV26	DoS with unexpected revert					
ESV27	DoS with unexpected revert					
ESV28	DoS with unexpected revert					
ESV29	DoS with unexpected revert					
ESV30	Erroneous visibility			✓		
ESV31	Erroneous visibility			✓		
ESV32	Erroneous visibility			✓		
ESV33	Erroneous visibility			✓		
ESV34	Erroneous visibility			✓		
ESV35	Erroneous visibility			✓		
ESV36	Erroneous visibility					
ESV37	Erroneous visibility					
ESV38	Erroneous visibility					
ESV39	Erroneous visibility					

ESV40	Erroneous visibility					
ESV41	Erroneous visibility					
ESV42	Erroneous visibility					
ESV43	Erroneous visibility					
ESV44	Erroneous visibility					
ESV45	Erroneous visibility					
ESV46	Erroneous visibility					
ESV47	Erroneous visibility					
ESV48	Erroneous visibility					
ESV49	Erroneous visibility					
ESV50	Erroneous visibility					
ESV51	Erroneous visibility					
ESV52	Erroneous visibility					
ESV53	Erroneous visibility					
ESV54	Erroneous visibility					
ESV55	Erroneous visibility					

ESV56	Erroneous visibility					
ESV57	Erroneous visibility					
Total		**3**	**7**	**11**	**7**	**0**

Appendix B

Classification and Analysis of the Security Audit Tools Findings

Table B.1: Mythril's Findings

Contract	Finding's Order in the Contract's Report	Vulnerability Type	True Positive
Migrations	1	External call	
Register	1	Unrestricted write	✓
Register	2	Unrestricted write	✓
Protocol	1	External call	✓
Protocol	2	External call	✓
Protocol	3	Block timestamp dependency	✓

Table B.2: Securify's Findings

Contract	Finding's Order in the Contract's Report	Vulnerability Type	True Positive
Migrations	1	Non-validated arguments	✓
Migrations	2	External call	
Register	1	Unrestricted write	✓
Register	2	Unrestricted write	✓
Register	3	Unrestricted write	✓
Register	4	Unrestricted write	✓
Register	5	Greedy contract	
Protocol	1	Transaction ordering	
Protocol	2	Transaction ordering	
Protocol	3	Reentrancy	✓
Protocol	4	Reentrancy	✓
Protocol	5	Unrestricted write	✓

Protocol	6	Unrestricted write	✓
Protocol	7	Unrestricted write	
Protocol	8	Unrestricted write	
Protocol	9	Unrestricted write	
Protocol	10	Unrestricted write	
Protocol	11	Unrestricted write	
Protocol	12	Unrestricted write	
Protocol	13	Unrestricted write	
Protocol	14	Unrestricted write	
Protocol	15	Division	
Protocol	16	Division	
Protocol	17	External call	✓
Protocol	18	External call	

Table B.3: Securify 2.0's Findings

Contract	**Finding's Order in the Contract's Report**	**Vulnerability Type**	**True Positive**
Migrations	1	Erroneous visibility	✓
Migrations	2	Erroneous visibility	✓
Migrations	3	Non-validated arguments	✓
Migrations	4	Non-validated arguments	✓
Migrations	5	Solidity naming convention violation	✓
Migrations	6	Complex Solidity version pragma statement	✓
Migrations	7	Exception handling	
Migrations	8	Uninitialised state variable	✓
Register	1	Erroneous visibility	✓
Register	2	Erroneous visibility	✓
Register	3	Erroneous visibility	✓
Register	4	Erroneous visibility	✓
Register	5	Non-validated arguments	✓
Register	6	Non-validated arguments	✓
Register	7	Unrestricted write	✓
Register	8	Unrestricted write	✓
Register	9	Unrestricted write	✓

Table B.4: Remix's Findings

Contract	**Finding's Order in the Contract's Report**	**Vulnerability Type**	**True Positive**
Migrations	1	Overspent gas	
Register	1	Overspent gas	
Register	2	Overspent gas	
Register	3	Overspent gas	
Register	4	Exception handling	
Register	5	Exception handling	
Protocol	1	Internal error during the reentrancy check	N/A
Protocol	2	Block timestamp dependency	✓
Protocol	3	Block timestamp dependency	✓
Protocol	4	Block timestamp dependency	✓
Protocol	5	Block timestamp dependency	✓
Protocol	6	Block timestamp dependency	✓
Protocol	7	Block timestamp dependency	✓
Protocol	8	Block timestamp dependency	✓
Protocol	9	Overspent gas	
Protocol	10	Overspent gas	
Protocol	11	Overspent gas	
Protocol	12	Overspent gas	
Protocol	13	Overspent gas	
Protocol	14	Overspent gas	
Protocol	15	Overspent gas	
Protocol	16	Overspent gas	
Protocol	17	Overspent gas	
Protocol	18	Overspent gas	
Protocol	19	Overspent gas	
Protocol	20	Overspent gas	
Protocol	21	Overspent gas	
Protocol	22	Overspent gas	
Protocol	23	Overspent gas	
Protocol	24	Overspent gas	
Protocol	25	Overspent gas	
Protocol	26	Overspent gas	
Protocol	27	Overspent gas	
Protocol	28	Overspent gas	

Protocol	29	Overspent gas	
Protocol	30	Internal error when checking if a constant can be used instead of a function	N/A
Protocol	31	Variables with similar names	✓
Protocol	32	Exception handling	
Protocol	33	Exception handling	
Protocol	34	Exception handling	
Protocol	35	Exception handling	
Protocol	36	Exception handling	
Protocol	37	Exception handling	
Protocol	38	Exception handling	
Protocol	39	Exception handling	
Protocol	40	Exception handling	
Protocol	41	Exception handling	
Protocol	42	Exception handling	
Protocol	43	Exception handling	
Protocol	44	Exception handling	
Protocol	45	Exception handling	
Protocol	46	Exception handling	
Protocol	47	Exception handling	
Protocol	48	Exception handling	
Protocol	49	Exception handling	
Protocol	50	Exception handling	
Protocol	51	Exception handling	
Protocol	52	Exception handling	
Protocol	53	Exception handling	
Protocol	54	Exception handling	
Protocol	55	Exception handling	
Protocol	56	Exception handling	
Protocol	57	Exception handling	
Protocol	58	Exception handling	
Protocol	59	Exception handling	
Protocol	60	Exception handling	
Protocol	61	Exception handling	
Protocol	62	Exception handling	
Protocol	63	Exception handling	
Protocol	64	Bytes and string length	

Table B.5: Links to Securify's Reports

Contract	**Link**
Enums	https://securify.chainsecurity.com/report/8e44004f893ef8734e6a3d1fb34cbd61dcfe7efae39452f09b37d604ac16634f
Migrations	https://securify.chainsecurity.com/report/7c1eaedd743fc5093bd37f1ce135d1b677a53945b2280dc37255fdfd120c57c0
Register	https://securify.chainsecurity.com/report/133d35b1bb453eceba123fee76a73555ab595cc5e15b10e1694f42e379ad9283
Protocol	https://securify.chainsecurity.com/report/93eccf4b6b71bfec4918ab8fece9c49667184ee823387a01c48de5c6cf6bb92d